Dynace

User Manual
November 20, 2024

by Blake McBride

This manual was typeset with the TEX typesetting system developed by Donald Knuth.

Short Contents

Table of Contents

1 Introduction

Dynace (pronounced *dī-ne-sē*) stands for a "DYNAmic C language Extension." It is an object-oriented extension to the C or C++ languages.

Dynace is written in the C language, designed to be as portable as possible and to achieve its goals without changing or adding much to the standard C syntax.

Dynace is a preprocessor, include files, and a library that extends the C or C++ languages with advanced object-oriented capabilities, automatic garbage collection, and multiple threads. Dynace is designed to solve many of the problems associated with C++ while being easier to learn and containing more flexible object-oriented facilities. Dynace is able to add facilities previously only available in languages such as Smalltalk and CLOS without all the overhead normally associated with those environments.

This manual is designed for a programmer who is proficient in the C language but has little or no knowledge of object-oriented concepts. We will start with some object-oriented concepts and then discuss how these concepts are realized with Dynace. We will then continue with a detailed description of Dynace usage. Finally, we will discuss the various classes that are built on top of the Dynace kernel.

1.1 Features

The following table lists some of the key features of Dynace:

- Written in C and designed to maximize portability (currently tested on 32 and 64-bit environments, including Linux, Mac, Windows, and various other systems such as Plan 9, VMS, IoT devices, etc.)
- Very little added to the standard C syntax (very easy to learn for anyone familiar with C)
- May be intermixed and linked with regular C or C++ code (easy to incrementally add OO features to pre-existing apps)
- Multiple Inheritance
- True Dynamic Binding (polymorphism) using Generic Functions
- Fast, Cached Method Lookup
- Full Metaclass System (runtime class representation)
- Enforced Encapsulation – very important for large applications! (No longer place 50\% of your code in include files, as in C++. If something changes, there is no need to recompile everything in Dynace. If a class needs to be changed, you don't need to worry who is looking under the hood, as in C++)
- Modeled after CLOS, providing the most flexible solution without the overhead and runtime penalty, and large memory requirements of an interpreter (as in CLOS and Smalltalk).
- Compiles with a regular C or C++ compiler – not an interpreter – runs FAST! (Doesn't have the runtime overhead and performance penalty experienced with CLOS and Smalltalk)

- Automatic Garbage Collection (the system automatically frees unused objects – you no longer have to keep track of what and when things must be freed)
- Multiple Threads
- Named & Unnamed Pipes
- Named & Unnamed Counting Semaphores
- Any class located anywhere in the class hierarchy may be changed without necessitating the recompilation of any other module so long as the pre-existing method protocols remain unchanged. (No need to recompile the world because you're not sure who is using or subclassing it – a major problem with C++ !!)
- Uniform representation of all objects, including Classes, Instances, Methods, and Generics. This allows easy construction of generic container classes.
- Powerful base class library including fundamental data types and collection classes such as Dictionary and Doubly Linked Lists.

1.2 Reasons to use Dynace

The following lists several reasons to use Dynace:

1. Drastically reduce application complexity and increase its maintainability by enforcing strong encapsulation – no more need to wonder which other modules are modifying your instance variables or what effect changes in one module will have on other modules.
2. Drastically reduce the need for recompilation since all class information is defined in the source file (*.c) instead of header files (*.h). Most changes to a class, including its place in the class hierarchy, instance or class variable declarations, and new or changed methods (member functions), have no compile-time effect on modules that use, subclass, or inherit from the changed class.
3. Easily build generic functionality (like generic container classes) which translates into increased use of existing code since Dynace treats all objects, including classes, in a uniform manner and supports weak and runtime object typing.
4. Dynace is the easiest OO language to learn since it's almost entirely normal C syntax, yet offers features far surpassing all the alternatives such as true dynamic binding using generic functions, multiple inheritance, fully metaclass-based (runtime class representation) from the ground up (like CLOS or Smalltalk), garbage collector (the system automatically frees unused objects), threader, and a powerful class library. Defining classes in Dynace is simple; all of the complexity and housecleaning chores are handled by the Dynace pre-processor (dpp).
5. Dynace applications run fast because they are compiled by your normal C compiler into fast machine code (no interpreter or p-code). Dynace includes a pre-processor which converts Dynace class definition files (which contain mostly straight C code) into plain C source files.
6. It's easy to incrementally add OO facilities into pre-existing C code because Dynace is just more C code.
7. No longer be the victim of a vendor – Dynace comes with full, portable source code.
8. You no longer have to distribute header files containing your proprietary class structure in order for third parties to be able to use and subclass your class. Dynace doesn't need any of this information.
9. The Dynace OO model is much more flexible due to its metaobject base. Now you can actually have greater control over classes as well as instances.
10. Dynace applications are royalty free.

1.3 Dynace vs. Other Languages

1.3.1 A comparison of Dynace and C++

The Dynace object-oriented extension to the C language was created in order to solve many of the problems associated with C++ while retaining more backward compatibility with C than C++ and offering stronger object oriented facilities. This section discusses many of the shortcomings of C++ and how Dynace addresses them, as well as other important facts regarding Dynace.

1.3.1.1 Code In Header Files

C++ greatly encourages the movement of code and class definitions from source files (.c) to header files (.h). Because of its design, C++ requires this in order to support many of its compile-time facilities. However, while enabling many compile-time facilities, it is also the source of many of C++'s shortcomings. Dynace is able to offer similar compile-time facilities (argument checking, ability to subclass a pre-existing class, etc.) while greatly encouraging the movement of code from header files (.h) to source files (.c). In fact, all Dynace class definitions are entirely contained within source files. The consequences of these fundamental design differences will ripple throughout this comparison.

C++ requires class definitions to be placed in header files and included by other source files for three main reasons. The first reason is that when other files need to declare or create new instances of a given class, C++ needs to know the size of the instance at compile time.

The second reason is that C++ needs to know the entire structure of classes at compile time in order to create new subclasses of them. The third reason is that C++ needs to know at compile time the names and locations of all externally accessible instance members (through public, protected, and friend members).

Dynace is able to perform these facilities with an increased level of abstraction and code protection without any class definitions in header files.

1.3.1.2 Frequency Of Re-compiles

One very important consequence of placing class definitions in header files is that even very small changes in a class's definition necessitate the re-compilation of source files that implement the class, files that use the class, and files that subclass that class. In large, real-life applications, this can amount to several hours of time. And for applications in which a lot of development is being done, this may be required every thirty minutes. The net effect of this problem is that compile time quickly becomes the place where most of the development time is spent.

Dynace, on the other hand, places all class definition code in source (.c) files. With Dynace, you can have a massive application, take a class out of the middle of the class hierarchy, and totally modify it, and so long as the pre-existing member functions (methods) retain their interface protocol (take the same arguments), only the one source file that implements that class would have to be re-compiled. Even if the member function's syntax changed, the only thing that would have to be modified and re-compiled were those modules that used the changed member function – something that would have to be done in any

environment. The net effect of Dynace's approach is that only the absolute minimum amount of time is wasted on compiles.

1.3.1.3 Application Maintainability / Object Encapsulation

One of the principal features of the object-oriented paradigm is encapsulation. Encapsulation is the ability to combine code and data into a package such that the only external access provided is through a well-defined interface. There are several important benefits associated with strong encapsulation.

At development time, strong encapsulation helps organize the design and breaks up the code into manageable-sized chunks. At debug time, strong encapsulation significantly limits the amount of code a programmer has to look at in order to locate the problem. After development, strong encapsulation makes code easier to understand for new programmers and limits the amount of code that must be checked and modified when a given module is changed. Strong encapsulation can have a major effect on the development, debug, and enhancement times associated with a project, not to mention the reliability of the code.

C++ supports four types of encapsulation (protection) associated with members of a class: private, protected, public and friends. Of the four types, three of them (public, protected, and friends) are ways to circumvent C++'s encapsulation features. Private members are only private from a narrow standpoint since they are declared in a publicly accessible header file.

Note that from a technical standpoint, there is never a reason for protected, public, or friend members. Anything you could ever need to do can be done with private members and access functions. The net effect is that C++, for no apparent technical reason, does not encourage strong encapsulation, nor does it even have the facilities to support it.

Dynace, on the other hand, enforces strong encapsulation (private members) and places all member definitions in a source (.c) file. With this approach, all the benefits of strong encapsulation may be maximally achieved.

1.3.1.4 C++ Templates

In order to write similar code for many classes, C++ provides templates. Templates are an extremely poor attempt to patch some of the blatant shortcomings of C++. There are two principal problems with templates. First, they suffer from the same compile-time limitations as all the rest of C++, namely, templates only work on classes that were set up at compile-time. Any time you want to add a new class, you must rebuild. The second problem is unnecessary duplication of code. If you compile a template for five classes, you get five copies of the same code!

Dynace has no templates and no need for them. Dynace methods support any class at runtime. There is no need to recompile the existing ones or to duplicate any code. The existing code adapts to new classes at runtime!

1.3.1.5 Virtual Functions

Virtual functions give C++ a great deal of flexibility. However, there is a major problem associated with them. If you purchase a third-party library and the vendor chooses not to implement a particular function as a virtual function and you later need to use it as a virtual function, it can't be done without the source code.

In Dynace, all methods operate like C++ virtual member functions. There is never one you can't use like a virtual function. Dynace implements these methods in a memory-efficient manner, so it's not prohibitive.

1.3.1.6 Learning Curve / Code Complexity

C++ is a large and complex language from a syntactical standpoint, especially when compared to the minimalist C language. There are two problems associated with this complexity. The first is that the learning curve associated with going from a proficient C programmer to a proficient C++ programmer is at least six months for most programmers. The second problem is that code written in C++ tends to be more complex and, therefore, harder for new programmers to understand and maintain. The bottom line is that C++ development tends to be much more expensive than C, and employers end up being much more dependent on a few "experts."

Dynace, on the other hand, uses straight C syntax and only adds a few easy-to-learn constructs. The learning curve associated with Dynace is about one week (although you can actually use it the first day). Pre-existing C code can coexist with Dynace code better than C++. And due to the simpler syntax and strong encapsulation features of Dynace, it is much easier for new programmers to pick up Dynace and the new application. This translates into real dollars saved.

Dynace adds powerful capabilities of object-oriented technology in a simple, idealistic, theoretically pure way. Dynace has taken the best attributes of CLOS, Smalltalk, and Objective–C and implemented them in a way that is natural to a C programmer. A Dynace program looks just like any other C program. Dynace is, therefore, much simpler and easier to master than C++, and the resulting tool is more powerful and expressive.

1.3.1.7 Code Reusability

C++ is a strongly typed language. This means that all type information associated with variables and function arguments must be completely specified at compile time. While attempting to create reusable code a programmer must create routines that are general purpose and can therefore be used in a variety of situations. The problem is that these two points are at odds with each other.

When creating general-purpose (reusable) software components, it is often desirable to have the routine handle a variety of data types, as is often needed in container classes. Strongly typed languages require that the potential data types be known at the compile time of the routine. This means that every time a new data type is used, the "general purpose" routine would have to be modified and recompiled. How many books and articles have you seen that attempt to trick C++ into handling data in a generic fashion? This important feature of good software development (creating reusable software) is an uphill battle with C++.

Dynace, on the other hand, is a weakly typed language. This means that Dynace treats all objects (including classes) in a generic fashion. Creating generic (reusable) routines like container classes is trivial in Dynace. Dynace is also able to validate an object or its type at runtime.

1.3.1.8 Strong vs. Weakly Typed Languages

Strongly typed languages (such as C++) require that the programmer specify exactly what data types a particular routine may handle. Weakly typed languages (such as Dynace, CLOS, and Smalltalk) allow a programmer to define a process in abstract terms without specifying any particular data types. It's kind of like the difference between speaking literally and speaking abstractly.

If teaching a child were anything like programming a computer and you could only teach him literally, you would have to explain to him that if he had two dogs and someone gave him another, he would then have three dogs. If he has two apples and someone gives him another, he'll have three apples. If he has two trains and someone gives him another he'll have three trains, and so on. However, if you could teach your child in abstract terms, you could simply tell him that if he has two of something and he receives another, he'll then have three of them.

The advocates of strongly typed languages like to point out that their language can uncover an incorrect argument to a function at compile time, before the program is even run. The reality is that passing the incorrect data type to a function is just one of many, many possible types of errors a given program may contain. Why restrict a language's capabilities so severely to uncover just one type of error?

In fact, when programming in C++, it is quite easy to create a situation where the correct type is being passed (a pointer to some structure) but when the code gets run, it hangs the system because it wasn't initialized and the compiler can't check for that. So C++, as well as other strongly typed languages, is quite good at performing static, compile-time type checking at the expense of a flexible language. And then when it comes to dynamic (runtime) checking, you're on your own.

While Dynace does check arguments at compile time, it does not attempt to validate each argument beyond whether it's an object or a particular C data type. In other words, all Dynace objects are treated opaquely. Not only can Dynace verify the type at runtime (essentially the same test C++ performs at compile time), but it will also validate an uninitialized pointer! If there is an attempt to use an uninitialized object or an incorrect type of object, Dynace prints an error message. No hung system.

Weakly typed languages, such as Dynace, provide an extremely more flexible and powerful means of expressing computer algorithms, and they do this with an increase in error detection and handling as compared to strongly typed languages such as C++.

Weak typing is a much stronger approach to implementing the object oriented model. Weak typing allows much more powerful, flexible, and general-purpose solutions to be developed. Since strong typing requires the types of data a routine handles to be known at compile time, at the very least, the module will have to be recompiled if it is to handle a new type of object, if not entirely recoded. Weak typing allows the development of truly general-purpose, reusable software components.

1.3.1.9 Generic Container Classes

The tremendous difficulty involved in tricking C++ into implementing generic container classes, such as dictionaries, linked lists, and bags, typifies the biggest problem with C++:

lack of runtime flexibility. How many books and computer magazine articles have to be written in order to show how to trick C++ into accomplishing something that should be fundamental and trivial? It lies at the heart of the value of object-oriented concepts.

The properties of C++ that make it so inflexible are its strong data typing and lack of runtime class representation.

Creating generic container classes is fundamental and trivial in Dynace. This is a very important difference.

1.3.1.10 Runtime class representation

C++ has no runtime representation of classes and no metaclasses. In fact, a C++ class is not an object at all. In short, C++ is a half hearted attempt to add object-oriented capabilities to the C language. Dynace implements all object-oriented concepts in a pure, flexible fashion. In fact, in Dynace, classes are objects just like any other.

Since C++ has no runtime class representation its compiler must know the entire (supposed to be private) structure of every class the module might use. This causes two very significant problems. First of all, it encourages the programmer to put a great deal of code in the header files. The more code put in the header the greater the chance that something in the header will have to be changed when modifying a class. If a header file changes all programs that are dependent on this header file must be recompiled. This is because the "new" operator must know the class size at compile time (even if all the elements are private!). What this ends up causing, in a real-life application where there are many classes in a complex hierarchy, is that the entire application must be totally recompiled every time there is the smallest modification (in order to play it safe). This one factor can multiply the time it takes to develop an application.

The second problem with not having a runtime class representation is a lack of flexibility and a lack of the ability to create truly general purpose routines. Since C++ does all its type checking at compile time it makes sense that at runtime, it can only handle the types it was compiled for. Therefore, if you compile some general-purpose routines into a library (or, worse yet, buy a third party's library without source) and later wish to use its existing functionality but for a class it was not compiled for, you would have to make some sort of code changes and re-compile the library (if you have source).

Since Dynace has a runtime class representation, it doesn't have any of these problems. All code and structure definitions for a class reside in the .c file. There is no need to put anything in a header. If a class changes, it has no compile-time effect on any other modules. As long as its original interface remains the same (although it can be added to), not a single line of code in any other module would have to be changed, no other modules would have to be recompiled, and no libraries rebuilt.

Again, since Dynace has a runtime representation, truly general-purpose routines can be built without requiring the old code to be recompiled. You can build a general-purpose routine, put it in a library, and then two years later create a totally new class and run an instance of the new class against the old code in the library without ever having to recompile the old code.

1.3.1.11 Efficiency

Dynace is not an interpretive language. Dynace programs are compiled with a standard C compiler. The majority of the code is just standard compiled C code with no performance penalty. The only place Dynace incurs a runtime cost is at the point of method dispatch. Since C++ also incurs a runtime cost when using virtual functions, there is not much of a difference between the performance of Dynace programs when compared to C++. In addition, Dynace gives the ability to locally cache a method pointer in tight loops to totally dispense with the runtime overhead.

1.3.1.12 Compatibility With C

C++ is somewhat compatible with C; however, Dynace is entirely compatible with C. Code that uses Dynace classes is just plain C code. Code that defines new Dynace classes consists of standard C with a few added constructs to define classes and methods. This then goes through a Dynace preprocessor, which emits straight C.

The two main points about this are that Dynace is more compatible with C than C++, and it's easier to incrementally add object-oriented facilities to a pre-existing C application with Dynace.

1.3.1.13 Garbage Collection

While C++ will automatically dispose of some types of objects (automatic variables), it cannot automatically dispose of those objects allocated from the heap. In a complex application, most objects would be allocated from the heap, requiring explicit disposal in C++. Dynace's garbage collector handles all object reclamation automatically.

1.3.1.14 Class Library

C++ environments come with an absolute bare minimum of fundamental classes. Of course, there are many C++ class libraries available; however, given the fundamental architecture of C++, these libraries are typically incompatible and difficult to understand, use, and extend.

Dynace comes with a complete set of fundamental classes, including classes to represent all the basic C types, a variety of container classes (sets, dictionaries, linked lists, associations, etc.), multi-dimensional dynamic arrays, threads, pipes and semaphores.

1.3.1.15 Conclusion

Dynace adds powerful object-oriented capabilities in a simple, idealistic, theoretically pure way. Dynace has taken the best attributes of CLOS, Smalltalk, and Objective–C and implemented them in a way that is natural to a C programmer. Dynace has no special syntax and is easy to learn. A Dynace program looks just like any other C program.

Dynace offers effective solutions to a variety of problems associated with software development, including reducing development time by segregating and organizing various application components, providing a powerful class library, making pre-written components maximally reusable, reducing language and application learning curves for new programmers, and making an application easy to debug, maintain, and enhance.

1.3.2 A comparison of Dynace and Smalltalk

Dynace is a very dynamic and flexible language, just like Smalltalk. Dynace and Smalltalk share the following features:

- Both are weakly typed languages and therefore allow for the development of powerful routines that provide truly generic and reusable software components.
- Both provide true dynamic binding for all messages.
- Both provide tight encapsulation of objects for easy program maintenance.
- Both provide a similar class hierarchy, including the Object, Behavior, Class, and MetaClass classes.
- Both provide a complete metaclass system that allows all objects in the system to be treated uniformly.
- Both provide the ability to perform automatic reclamation of unneeded objects (garbage collection).
- Both provide the ability to create, synchronize and pass information between multiple independent processes (threads).

The following lists some of the differences between Dynace and Smalltalk:

- Dynace uses a standard C syntax that is easy for C programmers to learn.
- Dynace may be freely mixed with regular C code and may also link with pre-existing C libraries.
- Dynace uses a standard C compiler and generates stand-alone native executable code. Smalltalk typically generates code for a virtual machine that must be included or bound with an application.
- Since Dynace generates native code, it runs many times faster than Smalltalk and requires much, much less RAM (just a little more than a typical C program).
- Dynace is written in standard C and comes with full source code. It is therefore very portable and drastically reduces your dependence on a vendor.
- Dynace supports multiple inheritance.

1.4 Obtaining The System

The entire system is available at `https://github.com/blakemcbride/Dynace`

1.5 Contents

Once the system has been installed, there will exist a series of directories under the `Dynace` root directory. The system will contain the following directories:

lib	This is the location of all the Dynace libraries. You may wish to add this directory to the list of paths your linker searches.
include	This is the location of the include files necessary to compile Dynace applications. You may need to add this directory to the path your compiler uses to search for include files.
bin	Executable files necessary for development with the Dynace system. This directory should be added to your normal search path for executable programs.
examples	Example programs used to learn & demonstrate the Dynace object-oriented extension to C.
docs	Miscellaneous documentation files.
manual	The Dynace manual.
kernel	Complete source for the Dynace kernel.
class	Source for all of the Dynace base classes.
threads	Complete source for the multi-threader, pipes, and semaphores.
generics	Files necessary to build the system generics files from scratch.
dpp	Complete source for the `dpp` utility.
utils	Complete source for the utility programs.

The only files that are absolutely necessary for a developer are those located in the `lib`, `include`, and `bin` directories.

1.6 Learning The System

This manual contains a description and rationale of object-oriented concepts, a detailed description of the Dynace system, and a complete reference to all Dynace classes, including examples. The example programs included with Dynace provide a step-by-step tutorial to getting started.

A user of the Dynace system will need to know the C language. However, no previous experience with object-oriented concepts is necessary.

The best approach to learning Dynace would be to start by reading chapters 1 (Introduction) and 2 (Concepts). Then read sections 3.1 and 3.2 (Using Dynace Classes, including their sub-sections) while working through the example programs. When the example programs start creating their own classes, follow with section 3.3 (Defining Dynace Classes, including its sub-sections). Ultimately, it is best to read the entire manual.

If you are somewhat familiar with object-oriented concepts and wish to dive right in and get a feel for the system, you may go directly to the example programs after reading this chapter.

While working through the examples, you may refer to the index (located in the back of the manual) to locate information on all Dynace methods, macros, variables, functions, and data types.

It is highly recommended that a programmer already familiar with object-oriented concepts still read the entire *Concepts* chapter because of its particular relevance to the approach taken in the Dynace system.

1.7 Quick Start & Tutorial

The Dynace system includes a series of examples that serve both as a Quick Start and a Tutorial. Each example is contained in its entirety in an independent directory. This is done to illustrate the exact files and steps necessary to create a single application.

The example programs are contained in subdirectories under the *examples* directory and are named *examNN* (where the *NN* is a two-digit number). These numbers are significant in that they describe the correct order in which the examples should be followed. Each example depends on knowledge built up in previous examples, which is not repeated.

Each example contains a *readme* file that describes information relating to the purpose of the example and build instructions. These files should be read first. Each example also includes makefiles. The source for the examples is also included.

The example programs are not intended as a substitute for the manual. They are simply meant to augment the manual and accelerate the initial learning curve.

The following is a list of the enclosed examples:

01 Illustrates the steps necessary to compile and link a program that incorporates and correctly initializes Dynace.

02 Illustrates the creation, use, and disposal of a simple object.

03 Illustrates the creation, use, and disposal of more simple objects.

04 Illustrates the creation, use, and disposal of an instance of the LinkObject class.

05 Illustrates the use of the LinkObjectSequence class to enumerate through the elements of a linked list.

06 Illustrates the creation, use, and disposal of an instance of the StringDictionary class.

07 Illustrates the process of enumerating through the elements of a set or dictionary using the SetSequence class.

08 Illustrates how Dynace handles errors.

09 Illustrates the value of and initialization procedure for the automatic garbage collector.

10 Illustrates the creation and initialization of a new class.

11 Illustrates the creation of a new method and generic function.

12 Illustrates additional points about methods and generics.

13 Illustrates the independence one instance has from another.

14 Illustrates the use of class variables / methods using gNew / gDispose.

15 Illustrates the initialization of instance variables with gNew.

16 Another illustration of the initialization of instance variables with gNew.

17 Illustrates subclassing.

20 Illustrates the use of threads.

21 Illustrates the use of the BTree classes.

30 Illustrates the process of getting information from the system.

31 Illustrates how generic functions are first-class C objects.

32 Illustrates how to locally cache a method lookup and avoid the runtime cost.

33 Illustrates most of the methods associated with the String class.

34 Illustrates some of the numeric and date formatting abilities.

See the file `examples/list.txt` for the most up-to-date list of example programs.

1.8 Manual Organization

This manual serves as both a user manual and a complete reference manual to the Dynace system. It makes no assumptions about the user's knowledge of object-oriented programming. It does, however, assume a working knowledge of the C language.

Chapter 1 (Introduction) covers background material needed to orient a new user.

Chapter 2 (Concepts) covers object-oriented concepts both in an abstract sense and as they relate to the Dynace system. It does this without introducing very much syntax or other mechanics.

A full understanding of these concepts is crucial to the effective use of the Dynace system. Even if you have experience with other object-oriented languages, it is important to understand these concepts as they are implemented within Dynace. Note, however, that it is not necessary to know everything before effective use of Dynace can be accomplished. Full understanding will come with the use of the system.

Chapter 3 (Mechanics) introduces the exact syntax and procedures necessary to use the Dynace system. It does much of this with examples. Although this chapter does not make explicit reference to the Dynace example programs, they should be reviewed while reading this chapter.

Chapter 4 (Kernel Reference) provides a detailed reference to all classes, methods, macros, functions, and data types associated with the Dynace kernel.

Chapter 5 (Class Reference) provides a detailed reference to all classes and methods associated with the class library included with the Dynace system.

The index (located in the back) provides a complete alphabetical listing of all classes, methods, macros, functions, and data types described in Chapters 4 and 5.

1.9 Contact Information

Email blake@mcbridemail.com

Discussion There is a Dynace discussion group at
https://github.com/blakemcbride/Dynace/discussions

1.10 Use, Copyrights & Trademarks

1.11 Credits

The Dynace Object Oriented Extension to C and its associated documentation were written by Blake McBride (blake@mcbridemail.com).

Credit and great appreciation are given to several sources for the ideas and inspiration behind many of Dynace's facilities as follows:

John Wainwright for the development and sharing of ideas concerning early releases of his Objects In C (OIC) system. He is an extremely talented individual with many unique and innovative approaches to software development issues. I am proud to call him my friend.

Many thanks to Robert Nielsen for his many great ideas and motivation. Without him, Dynace would not be what it is.

Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow for their book "The Art of the Metaobject Protocol" and their CLOS system.

Adele Goldberg and David Robson for their books on Smalltalk-80 and the Xerox Palo Alto Research Center for providing access to much powerful and innovative technology.

And finally, Dr. Brad Cox for pioneering object-oriented extensions to C with his Objective–C system.

2 Concepts

Over time, the methods used in the development of computer software have evolved from a *Goto* model, to a *Structured Programming* model, and finally the *Object Oriented model*. In the beginning, the `goto` model was able to accommodate all the needs of the programmers. However, as the size of programs became larger and more complex, the goto code began to look like spaghetti. No one, except perhaps the original author, was able to understand, and therefore maintain, a complex program that was jumping all over the place. The solution to this problem was the Structured Programming model.

The Structured Programming model got rid of `goto` statements and replaced them with the more structured `for` and `while` statements. With this model, the programmer knows the beginning and ending location of the loop or conditional, as well as what the specific conditions are for executing the contained code. No more spaghetti.

Structured Programming also stressed the use of subroutines. This allowed blocks of code to be packaged in a form that could be more easily understood as a small, well-defined unit. It also enabled, to some degree, the ability to create reusable code. It should be noted that while packaging blocks of code as subroutines greatly increased the programmers' ability to manage large programs, it also comes at a cost. Calling blocks of code as subroutines is substantially more expensive, in terms of computer cycles, than a `goto` statement. As time has progressed, however, it has generally been accepted that the benefits of subroutines far exceed the costs.

The Structured Programming model worked well for many years. However, as computer programs became even larger, the programmers' ability to manage larger, more complex projects became increasingly difficult. The ability to use common subroutines across varying applications was very limited. It became very difficult to manage the relationship between all the varying data structures and various subroutines.

Three things were needed to solve the problems of the Structured Programming model. First, there needed to be a way of isolating or packaging related groups of code and data, so that if a modification had to be made, or a bug fixed, only a specific group of code and data had to be understood and modified. This would allow a programmer to change a system, or add new functionality, without first having to understand all the intricacies of the entire system.

Second, any changes made to the related group of code and data should have minimal effect on any other code or data. This way, if a programmer changes a module, he shouldn't then have to search every place it's used to see what additional changes have to be made in order to accommodate the module change.

The third problem is that of maximizing reusability. For years, programmers have been writing the same linked list routines, the same sort routines, and the same hash table routines over and over. The ability to create general-purpose routines, which could be used in many places, over and over, is of immense value. After the creation of a good general purpose library, applications development time would be reduced considerably, and code reliability increased considerably. In addition, any enhancements made to the general-purpose library would have an automatic benefit to all applications that use the library.

All three of the above issues are addressed in the Object Oriented model. The first two are addressed by the *encapsulation, data abstraction*, and *loose coupling* attributes of the Object Oriented model. The third issue is addressed by the *inheritance* and *dynamic binding* aspects of the Object Oriented model. All these issues, as well as others, will be addressed by the remainder of this manual.

2.1 Object Oriented

The Object Oriented model of programming is embodied by three fundamental concepts, a bit of new terminology, and a few new programming mechanisms used to implement the new concepts. The three fundamental concepts are *encapsulation, polymorphism, and inheritance*. These concepts will be introduced first, followed by the mechanisms. The terminology will be introduced as it is encountered.

2.1.1 Encapsulation

Encapsulation allows the programmer to tightly bind a data structure with the routines that access it into an item called an *object*. Encapsulation reduces the ways in which outside data or procedures may affect a given object. Outside elements may only affect a given object by a limited number of well-defined methods. Using a limited number of ways to affect an object causes *loose coupling*. A well defined method of effecting an object is a *protocol*.

A limited example of this idea can be achieved, for example, by creating a unique structure declaration inside a particular C source file or module, as opposed to inside an include file. By doing this, only the functions in the same source file as the structure declaration may access elements of the structure, since those are the only routines that have any knowledge of its structure. Anything outside the defining module must call a non-static function, located within the module, in order to affect the data structure. These non-static functions would, in effect, comprise the module's *protocol*. The module would have *loose coupling* because the only way outside routines may affect the internal data structure, or state of the object, is, again, through the non-static functions. External routines have no access to the static or local routines and have no direct access to the data structure itself.

Further, a very important form of *data abstraction* is achieved through encapsulation. Data abstraction refers to the ability to change the structure of data without having to change any external routines that, indirectly, use the data structure. As in the previous example, the data structure, as well as the entire implementation of the module may be changed entirely, and so long as the pre-existing protocol and module functionality remain intact, all other modules that use this module will continue to work without *any* modifications. Major implementation details may change and be made more efficient, and major additions to the module's functionality may be made without any effect on external, pre-existing code.

As opposed to the above example, the form of encapsulation implemented in most of the more advanced object-oriented models, as well as Dynace, is much tighter. External routines not only have no access to the elements of the data structure, but also have no direct access to *any* of the routines that may affect the data structure! The data structure and associated routines are tightly packaged into a unique object. The only way to affect the object is to request the object itself to make any modifications to itself or perform any operation. The meaning and methods to achieve this will become increasingly clear as we progress.

2.1.2 Polymorphism And Overloading

In a normal C program, you may only have one function with any particular name. For example, if you had a non-static function `fun1()` in one module, you couldn't have another

non-static function `fun1()` in another module. The two modules wouldn't link. If the program did link, and a third module called `fun1()`, the system wouldn't know which one you wanted to execute.

Polymorphism or *overloading* refers to the ability of the system to determine, based on context, which particular routine to execute. The context normally refers to the arguments to the routine. So, for example, in our previous example, you would be able to link two modules, each with non-local `fun1()`s. If a third module called `fun1()` the system would know which `fun1()` to execute based on the types of the arguments passed to `fun1()`. This ability to determine which specific routine to execute among several possible options is called *dynamic binding*.

At first, it may seem unclear what value polymorphism may have. The following example will show that polymorphism is an extremely valuable tool when trying to create general-purpose, reusable software components.

Let's say you had a *box* object. That is, a data structure representing a box (height and width), and a group of routines that can create, modify, draw, and destroy a particular box data structure. Now we want to create a routine to draw an arbitrary number of a particular box. We would probably write the routine as follows:

```
void    draw_n(box b, int n)
{
        while (n-- > 0)
                draw(b);
}
```

If you later wanted to modify `draw_n()` to work with circles, as well as boxes, you would normally have to modify `draw_n()` as follows:

```
void    draw_n2(void *b, int n, int type)
{
        if (type == BOX_TYPE)
                while (n-- > 0)
                        draw_box((box) b);
        else if (type == CIRCLE_TYPE)
                while (n-- > 0)
                        draw_circle((circle) b);
}
```

The first problem with the `draw_n2()` solution is that `draw_n()` had to be changed to handle the additional type. The second problem is that the syntax has also been changed. This means that everywhere `draw_n()` is called the syntax would have to be adjusted. The third problem is that now the programmer has to keep track of which type `b` really is. All this work is necessary just to add a little additional functionality. You can imagine what would be involved in making these types of changes to real, complex applications.

The facility of polymorphism alleviates the need for *any* of these changes. As an example, in Dynace, the code could have originally been written as follows:

```
void    draw_n3(object b, int n)
{
        while (n-- > 0)
                draw(b);
}
```

If you then created a `circle` object, with all the appropriate supporting functions, and passed it to `draw_n3()`, it would work fine! No code changes anywhere! You wouldn't even have to recompile `draw_n3()`! You could later pass `draw_n3()` a `triangle` or a `rectangle` and again, no code changes, and no recompilation of `draw_n3()`.

2.1.2.1 Early vs. Late Binding

In the previous example of `draw_n3()` one should understand that there is a `draw()` function defined for the `square`, one for the `circle`, one for the `triangle`, and so on. What's going on is that the system is automatically determining which `draw()` to call based on the type of `b`, which is kept automatically.

Some object-oriented implementations have what's called *early binding*. Early binding is when the relationship between the function call and what's actually called is determined when the program is being compiled. The advantage of early binding is speed. Since the determination is being performed at compile time, there is no runtime cost. It is just as fast as a normal function call.

There are, however, some serious disadvantages of early binding. First of all, in order for the compiler to make the determination, it must know the type of the argument at compile time. This means that the procedure could only work for *one* type of object. This is entirely inadequate for the above example. Using early binding, one would define a function as in `draw_n()`. This would allow multiple definitions for `draw()`, however, it would only work for a `box`. Therefore, early binding provides only a partial solution to the problems polymorphism can solve.

Late binding refers to a system that determines which function to call at runtime. Late binding requires that the system knows, at runtime, the type of any object. This information is typically kept automatically by the system, as it is in Dynace. Late binding has the advantage that it gives all the benefits of polymorphism. Therefore, one could define a very general-purpose `draw_n3()` type of function within a system that supports late binding. The only disadvantage normally associated with late binding is speed. The system must determine which `draw()` function, for example, to execute at runtime.

Dynace normally uses late binding and has a caching facility that minimizes the runtime overhead associated with late binding. Dynace also contains a method for reducing the runtime overhead to virtually zero. This method may be used in tight loops. Dynace can also use early binding; however, this is discouraged.

2.1.3 Inheritance

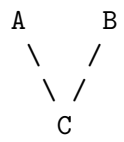
Inheritance allows you to create objects that are like other objects or groups of objects, but have some differences. Inheritance is an extremely powerful tool. It enables a programmer

to create a new module by using pre-existing modules. The only parts the programmer has to write are the parts that make the new module unique. In Dynace, all this can be achieved without any access to implementation details or source code of the pre-existing modules.

Inheritance operates in a hierarchical form. Thus, if object `B` inherits from object `A`, and a new object `C` is created which inherits from object `B`, you will have the following graphical relationship:

This diagram shows that object `C` has all the data and functionality locally defined as part of object `C`, as well as all the data and functionality of objects `B` and `A`. Object `B` has all the data and functionality locally defined as part of object `B`, as well as all the data and functionality of object `A`, but not object `C`. Object `A` has only the immediate data and functionality that are locally defined.

This idea may be extended in a multiple inheritance form pictured below:



In this example object `A` has only its locally defined data and functionality. Object `B` also has only its locally defined data and functionality. Object `C`, however, contains its locally defined data and functionality, as well as the data and functionality of both object `A` and `B`.

This idea of single and multiple inheritance may be extended arbitrarily in both depth and width, whatever makes sense.

2.2 Dynace

Dynace is an object-oriented extension to the C language. It implements all of the features discussed in this manual. Dynace is written in the C language and designed to be portable, efficient, powerful, and simple to use. Dynace was designed after an in-depth study of Smalltalk–80, the Common Lisp Object System (CLOS), Objective–C, C++, as well as object-oriented principles in general. Dynace was designed with the following goals:

- Implement as much power and flexibility, available in languages such as CLOS and Smalltalk–80 as possible.
- Create a language that was as simple to use as possible, like Objective–C.
- Create a language that stuck as close as possible to the standard C syntax.
- Create a language that was as efficient as possible, such as C++ and Objective–C.

Dynace succeeds in the satisfaction of these specific goals better than any of the other languages above.

The remainder of this manual will deal with more specific aspects of Dynace, although the vast majority of these concepts are very closely related to most of the above languages.

2.2.1 Object

Dynace adds to the C language only *one* new data type, the *object*. An *object* is a package consisting of data and the procedures necessary to create, destroy, modify, and otherwise affect the data. An object always knows what type it is. Dynace operates on objects. An object can be thought of as a fundamental data type. In fact, `object`, much like `int` or `long`, is a fundamental type that may be used in declarations. All other elements of Dynace, such as classes, metaclasses, instances, methods, and generic functions, are regular objects, just like any other object you may create in an application. In this respect, *all* objects are treated exactly the same throughout Dynace. The exact same code which processes the pre-defined Dynace objects, which allow Dynace to operate, processes the user-added objects.

Variables containing objects may be declared as follows:

```
object    a, b, c;
```

In this example three variables, `a, b,` and `c`, which may refer to objects, have been defined.

2.2.2 Classes & Instances

In Dynace, *each* and *every* object refers to an object, called a *class*, which defines that object. One class object may define any number of objects. Each object that is defined by a particular class object has the same data structure and performs the same functions as all the other objects that are directly defined by that class object. The only thing that differentiates one object from another, when they are both defined by the same class object, is the values contained within their similar data structures.

The objects that are defined by a particular class object are called *instance* objects of that class object. Class objects have names. For simplicity's sake, we will call a class object

with a name `X`, `X` class. An instance object of `X` class is called an instance of class `X`. In general terms, we can speak directly about classes and instances.

Now that we have some terminology under our belt, I can restate exactly what classes and instances are and what their relationship to each other is. One very important thing to keep in mind, and the main reason all these terminology gyrations were necessary, is that classes and instances are both objects. There is nothing more special about one than the other. There is no code in Dynace that treats classes any differently from instances.

A class keeps track of the structure of the data located in its instances. A class also keeps track of the functions the instances are able to perform. Each instance has only a single class that defines it, and it always knows which class it is. There may be an arbitrary number of instances of a class, however.

2.2.2.1 Superclasses

Each class may have zero or more superclasses. Classes normally have at least one superclass, though. The data contained in an instance will be the sum of the data defined directly in the instance's class, and the data defined in all the superclasses of the instance's class. In addition, the instance will also contain the data defined in its superclass's superclasses. This continues until there are no more superclasses.

The instance's functionality follows the same path as its data.

If class `A` is a superclass of class `B`, then class `B` is called a *subclass* of class `A`.

2.2.2.2 Instance Variables

The data contained within a particular instance are called its *instance variables*. The values contained within an instance's instance variables are unique to that individual instance. Of course every instance of one class will always have the same instance variable structure, though. You can think of instance variables as a C structure definition. Each instance of a particular class contains the same structure. It just contains different data within that structure.

When dealing with an instance, only the procedures that are part of the instance's class are able to access the instance variables. This ensures encapsulation and data abstraction.

2.2.2.3 Class Variables

Each class may also contain data. This data, which is located in the class, is called *class variables*. The data in class variables is always unique to a particular class. Class variables often serve as data that is common among all the instances of the class. With class variables you can keep track of how many instances of the class there are, or sums of their instance variables. You can even keep a linked list of a class's instances if you like.

Like instance variables, class variables may only be accessed by specific procedures that are bound to the class.

2.2.3 Metaclasses

As was discussed before, all things in Dynace are *objects*, and are treated the same. If this is true, you may be wondering why we are discussing classes and instances as if they are two

different things. The fact is, they are not! In fact, it so happens that, like instances, classes are also instances; instances of another class. The classes whose instances are themselves classes are called *metaclasses*. And as you might imagine, metaclasses are also instances of other classes. This is a circular process that goes on without beginning or end.

In this sense Dynace is quite interesting. Every part of Dynace, and all of its functionality, is defined by some other class. Fortunately, though, Dynace hides much of its complexity. You will rarely, if ever, have to concern yourself with any metaclasses or any aspects of its recursive design and operation. There is one important thing to remember, though: every class is an instance of exactly one class (the metaclass) and has zero or more superclasses.

2.2.4 Methods

As was discussed earlier, the only procedures that may access instance variables are the procedures associated with the class of the instance. These procedures are called *methods*. Methods are used to effect all aspects of an instance's variables. In fact, methods are the only things that can create or destroy instances.

There is one interesting thing about methods, though. Like instance variables, there is not normally a way to evoke a method. Methods, like instance variables, are tightly bound to the class, and there is no easy external access to the method. Thus, methods, like instance variables, are encapsulated and abstracted.

After reading about instance variables and methods being so tightly bound and inaccessible, external to the object, one may begin to wonder how one could ever effect an object. Be patient. The next few sections will wrap up all the ideas.

These features are very important. They are among the key elements that allow the object-oriented model to be such a powerful tool.

2.2.4.1 Instance Methods

Instance methods are methods that typically operate on instance variables. Instance methods are conceptually located in the class of an instance.

2.2.4.2 Class Methods

Class methods are methods that typically operate on class variables. Class methods are conceptually located in the class of the class, or the metaclass.

2.2.5 Generic Functions

Given how tightly bound and inaccessible instance (or class) variables and methods are to an object, one might wonder just how anything external to the object can cause any change in the object. The answer to this is *generic functions*.

Generic functions act as special C functions that take one or more arguments. What a generic function does is check the class of the first argument to the generic function for a method that has been associated with this generic function. If one is found, it is invoked. If one is not found, the generic function will search all the superclasses of the class of the first argument. This superclass search will continue until a method is found or there are no more superclasses to check. If no method is found, an error is signaled.

Although this search process sounds quite involved and time-consuming, it really is not. Dynace employs a caching scheme, so there is almost never an actual search.

As you can see, this generic function mechanism gives Dynace quite a bit of its power and flexibility. Through generic functions, most of the dynamic binding and method inheritance facilities are implemented.

Dynamic binding is achieved, for example, through the fact that class `A` could have a method called `M1`, as can class `B`. The generic function will invoke the correct method based on the class of its first argument. If the first argument to the generic function is an instance of class `A` then class `A`'s `M1` method will be invoked. Likewise, if the first argument to the generic function is an instance of class `B` then class `B`'s `M1` method will be invoked. The two methods could perform entirely different functions.

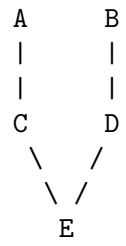
The method inheritance facility of Dynace is realized by the fact that generic functions search the superclasses until an appropriate method is found. It will keep searching up the tree until a class that implements the necessary operation is found.

All the arguments passed to a generic function are passed directly to the appropriate method, just as if the method were called directly. Therefore, since one implementation of an `M1` method may take a different number of arguments (or different types) than another `M1` method, generic functions take different arguments (or different types) depending on the method that will eventually be invoked. In addition, the return value of a generic function is whatever the method returns.

The process of invoking a generic function may also be referred to as sending a *message* to the object that is the first argument to the generic function.

2.2.5.1 Method Lookup

The order with which generic functions search for an appropriate method is important to understand. The first place searched is always the class of the object in the first argument to the generic function. If the method is not found, the first superclass (the one listed first when the class was created) is searched and then its superclasses. After checking the entire branch, if the method is still not found, then the next superclass to the first class will be searched. This process will continue until a method is found or there are no more superclasses. If no method is found, an error is signaled. The search method, therefore, is a depth-first search. For example the following class structure:



would be searched in the following order `E--C--A--D--B`. Note that the order of `C` and `D` is determined by the order they are listed when class `E` is created.

One important point: If the first argument to the generic function is an instance, the class of the instance will be searched first. And since instance methods are kept in classes, an *instance* method will be evoked. However, if the first argument to the generic function is a class, the class of the class, or the *metaclass*, will be searched first. And since class methods are kept in metaclasses, a *class* method will be evoked.

So to summarize, if the first argument to a generic function is an instance, then an instance method will be evoked. Likewise, if the first argument to a generic function is a class, then a class method will be found.

2.2.6 Memory Management and Garbage Collection

All objects in Dynace are dynamic entities that are allocated from the C heap. When a new object is created, it is allocated from the heap, but when an object is no longer needed, it must be returned to the heap, lest we use up all the memory. This can be done in Dynace explicitly via one of the disposal generics. However, as an application gets larger and more complex, it becomes more and more difficult to know exactly when an object is no longer needed. Also, there is often a larger than desired level of complexity added to an application when trying to keep track of when an object should be disposed of and being sure it is.

Garbage collection is a system used to free the programmer of most concerns about when an object should be freed and making sure it is done. The garbage collector works by marking all the objects that your program is using and then going through memory and freeing any object that is not marked. This process happens automatically and without any programmer intervention. The algorithms used in Dynace assure smooth, reliable, and efficient operation of an application without ever having to worry about freeing any objects.

2.2.7 Threads

Dynace provides two different methods of multi-thread support. One is the Dynace-provided thread-sharing mechanism where all of the application threads share a single operating system thread. The second capability is support for true multi-OS threads so that your application can take advantage of multiple CPUs.

In a normal C program, when one function calls another the first function waits until the second function completes before it may continue. The program follows a single line, or *thread*, of execution. In a multi-threading language, a function may call a second function, and while the second function executes, the first function is able to continue running. Thus, two threads would be running. Either thread could even start more threads.

Threads share the same global variables and opened files just like the functions of a normal single-threading system. Each thread, however, has its own stack space for local variables and function calls.

The advantage of having multiple threads is that it adds the ability to create applications that make more efficient use of user and computer CPU time. While a user is updating information in an application, the same application program could be printing a report. Any time an application has to perform an operation that the user would normally have to wait for, such as report printing or other processing of information, it could continue in a thread while the user is able to perform other data entry/query operations in another thread.

Dynace supports multiple threads in a semi-preemptive, round-robin, priority-based system. Dynace uses the system timer to assure that each thread gets equal access to CPU time. Semi-preemptive operation is achieved by the fact that Dynace performs automatic thread switching whenever a generic function is called (if the thread's time is up). Each thread has a priority associated with it. Higher priority threads get first access to CPU time, while threads of equal priority share CPU time equally.

Dynace supports threads being put on hold, released, priority changed, waiting for other threads, getting thread return values, and a blocking `getkey` function. Threads on hold or otherwise not running require no CPU time.

2.2.8 Pipes

Once there are multiple threads running, there arises a need to coordinate and communicate among the multiple threads. *Pipes* are a stream-based facility that allows one thread to pass information to another using a `puts/gets` type of interface. *Named pipes* allow one thread to create a pipe and give it a name. Later, if another thread wants to use the pipe all it needs to know is the name. It can get access to the pipe object via the name.

Dynace supports named and anonymous pipes with blocking and non-blocking I/O.

2.2.9 Semaphores

Thread coordination is achieved with *semaphores*. Semaphores allow the programmer to control the number of threads that may have simultaneous access to arbitrary system resources or blocks of code.

Semaphores may be thought of as a global variable that is initialized to zero. When one thread wants access to a resource, it first increments the global variable. Later, if a different thread wants access to the same resource, it first checks the value of the global variable. If it is greater than zero, it means that another thread is utilizing the resource and that the second thread should wait until the first thread is done. When the first thread is done with the resource, it decrements the global variable. This allows the second thread to start up and use the resource.

Real semaphores, as opposed to global variables, automate the creation and destruction of semaphores, the maintenance of the semaphore count, and the holding and automatic release of threads.

Dynace supports named and anonymous counting semaphores. Dynace semaphores are implemented in such a fashion that threads that are waiting for a semaphore require no CPU time!

2.2.10 Dynace Kernel & Class Library

Dynace consists of two separable parts. One part is called the *kernel*, and the other is the *class library*. The kernel contains all the raw functionality of Dynace. It contains all the features that allow the creation and functionality of classes, instances, methods, generics, and garbage collection.

The class library provides a set of general purpose classes that are built on top of the Dynace kernel. These classes provide basic facilities to create object representations of all

the basic C language data types. It also provides other general-use classes to handle object collection and ordering.

The Dynace kernel is entirely functional without the class library.

3 Mechanics

This chapter describes the actual steps necessary to build and use Dynace. First the steps necessary to configure, compile, and link Dynace and Dynace applications are described; then the steps necessary to use existing classes are described. Once this is understood, the creation of new classes is described. Finally, Dynace customization and special techniques are discussed.

3.1 Setup, Configuration & Build Procedures

This section, in concert with the examples, describes the tools and actual steps necessary to build Dynace and create Dynace application programs. The best method of learning is to go through the example programs located under the `\DYNACE\EXAMPLES` directory in the order in which they appear.

3.1.1 Build Utilities

All the utilities used to build Dynace applications are contained within the Dynace package and the command line utilities (compiler, linker, etc.) that come with your compiler. Dynace utilizes special utilities and build procedures designed to intelligently minimize the need for re-compilation. These procedures are too complex for the IDEs that come with the various compilers. For this reason the IDE environments that come with your compiler are not used. Instead, under Windows, the Microsoft-supplied *nmake* utility is used. Under all other Unix-like operating systems such as Linux and macOS, GNU Make is utilized. Having said that, however, you may still use the IDE for editing and debugging. You may also be able to find a way to integrate the supplied makefiles and procedures into your favorite IDE.

3.1.2 Building Dynace From Scratch

The procedure used to build Dynace from scratch is fully described in `\DYNACE\DOCS\BUILD.txt`. See that file for build instructions.

New users will want to read all the files in the `docs` directory.

3.1.3 Examples Setup

All examples are buildable under all supported environments. The only difference is the makefile used.

3.1.4 Example Files

This section documents the files that are contained in each example program.

README This file describes the objective of the example. It should be read first.

MAIN.C This file is the complete source code for the example program. It is fully commented and describes all aspects of the current example that are unique.

In addition to the above, the following build/system-specific make files are used.

MAKEFILE.MSC
NMAKE makefile for Microsoft Visual C 32 & 64 bit

MAKEFILE
: GNU MAKE makefile for Unix, Linux, Mac, BSD, etc. systems

Once an application is built, the only file needed to run it is `MAIN.EXE`.

3.1.5 Building The Examples

In order to build with the NMAKE command line utility execute the following command.

```
nmake -f makefile.msc
```

Alternatively, you can use the following command to create a debug version of the example:

```
nmake -f makefile.msc  DEBUG=1
```

3.1.6 Debugging

When `dpp` strategy 1 is selected Dynace uses a small amount of assembler. This is not true for the default strategy of 2; however, when strategy 1 is used, there are a couple of procedures which, when followed, will ease the debugging process.

First, compile your application with debugging information. Then, when you wish to debug the application, set a breakpoint at the beginning of any methods you wish to debug. You may then step through your program as normal and when you enter a generic, the system will stop at the methods in which you placed the breakpoints.

Once you are fully into the method, you may access the instance's variables through the "`iv`" structure pointer and the class variables through the "`cv`" structure pointer.

3.1.7 Building Your Own Application

The best method of building your own application would be to start with one of the example programs and proceed from there. This way, all the compiler, linker and other options will be preset. The best examples to start with would be one of the ones that define a new class. Their makefiles are designed to handle class files in a very convenient and automated way.

3.2 Using Dynace Classes

Dynace usage can be divided into two categories: *class user* or *class creation*. All a class user has to do to use Dynace classes is include a single file in the source code and call a function to initialize the system. After these two easy steps, the class user has full access to the entire set of existing classes. A class user need not learn any special syntax or procedures. Learning to use the existing classes is the same as learning to use any other library of routines.

A *class creator* will have to understand all aspects of being a class user first. In addition, a class creator will have to understand the steps necessary to define classes, methods, and generics. These steps are well defined and simple to use.

This section describes the actual steps necessary to *use* a pre-existing class. The next section will discuss the steps necessary to create new classes.

3.2.1 External Naming Conventions

External naming conventions refer to those names that the programmer defines, such as a new class name, an instance variable name, or a method or generic name. By contrast, internal naming conventions refer to those names that are generated by the Dynace preprocessor (`dpp`). These internal names are often made up of modifications or concatenations of the external names.

In Dynace, generic function names, by convention only, always begin with a lowercase "`g`" or "`v`", followed by an uppercase letter, and then optionally followed by other letters (of any case), numbers, or underscores. This convention allows a user to immediately know which functions may be used polymorphically or overloaded.

Generics that begin with "`g`", by convention only, have compile time (as well as runtime) argument checking turned on and often, but not always, have a fixed number of arguments.

Generics that begin with "`v`", by convention only, have compile time argument checking turned off and are often, but not always, associated with generics that take a variable number of arguments. The use of generics with argument checking turned off allows them to be associated with methods with arbitrary argument signatures. It is up to the system and the programmer to ensure that the correct arguments are passed. Dynace provides ample tools necessary to ensure correct arguments at runtime. In fact, the runtime checking facilities are much more reliable and foolproof than compile-time argument checking. The only disadvantage is that you don't know a problem exists until you run the system. If, however, an error is discovered, it can be handled in many different ways and never crashes the system.

Dynace uses the "`g`" type of generic as much as possible. However, some methods, such as indexing into a variable dimension array or `printf`-like functionality, are implemented as "`v`" generics.

It's all right if several methods (associated with different classes) have the same name because methods are implemented as static C functions. So there is no name conflict.

Classes normally begin with a capital letter. Instance variables, by convention only, begin with "`i`" and are followed by an uppercase letter. Likewise, class variables, by convention only, begin with "`c`" and are followed by an uppercase letter.

In the index the notation will specify a generic (minus the normal leading "`g`" or "`v`"), two colons, and its associated class. For example, `ChangeValue::Number` specifies a generic called `gChangeValue` which is associated with the `Number` class.

The above naming scheme is used by the Dynace system; however, it is not enforced. You may use any naming scheme you wish.

3.2.2 Internal Naming Conventions

External naming conventions refer to those names that the programmer defines, such as a new class name, an instance variable name, or a method or generic name. By contrast, internal naming conventions refer to those names which are generated by the Dynace preprocessor (`dpp`). These internal names are often made up of modifications or concatenations of the external names.

The following table lists the internal name structure used by Dynace. The associated source external names for example purposes are the class `String` and the generic `gNew`.

`String` a macro which evaluates to the class object but also automatically initializes the class if it hasn't already been initialized.

`String_c` the actual variable which contains the class object (`NULL` if the class hasn't been initialized yet).

`String_t` a typedef defining a String class (always object), used for documentation purposes only.

`String_cv`
pointer to the actual locally defined class variable structure.

`String_cv_t`
a typedef which defines the locally defined class variable structure.

`String_iv_t`
a typedef which defines the locally defined instance variable structure.

`String_initialize`
the name of the class initialization function generated by `dpp`.

`gNew_t` a typedef defining a pointer to a function with the return type and argument signature of the generic.

`gNew_g` a Dynace object representing the generic.

`String_im_gNew`
default instance method name for methods with argument checking turned on (imeth).

`String_cm_gNew`
default class method name for methods with argument checking turned on (cmeth).

`String_ivm_gNew`
: default instance method name for methods with argument checking turned off (ivmeth).

`String_cvm_gNew`
: default class method name for methods with argument checking turned off (cvmeth).

3.2.3 Class Hierarchy

The only class in Dynace that has no superclasses is called `Object`. All classes in Dynace eventually inherit from the class `Object`. The Object class defines those features that are common to all objects. The class `Behavior` inherits from Object and defines the functionality common to classes and metaclasses. The class `Class` inherits from Behavior and describes the functionality particular to all classes. The class `MetaClass` also inherits from Behavior and defines the operations particular to metaclasses. There is a class called `Dynace` that inherits from Object and is used to effect various aspects of the Dynace kernel. There are also classes called `Method` and `GenericFunction`, which both inherit from Object, and perform the expected functions.

The above class hierarchy can be depicted as follows:

```
Object
     Behavior
          Class
          MetaClass
     Dynace
     Method
     GenericFunction
```

These classes comprise the entire Dynace kernel. Other facilities of Dynace, such as threads, pipes, and dictionaries, are supplied as Dynace classes. Although an in-depth understanding of each of the above classes is unnecessary, it is important to fully understand the relationship between the inheritance of the above classes and the indentation of the diagram.

3.2.4 Example Code

The following example illustrates a complete program that uses Dynace. Each line is explained in the following subsections.

```
#include "generics.h"

main(int argc, char *argv[])
{
        /*  declare some variables which will hold objects  */
        object  name, pi, age;

        /*  initialize the Dynace kernel and class library  */
        InitDynace(&argc);

        /*  optional:  start automatic garbage collector  */
        gSetMemoryBufferArea(Dynace, 40000L);


        /*  set name to an instance of the String class  */
        name = gNewWithStr(String, "Tom Swift");

        /*  set pi to an instance of the DoubleFloat class  */
        pi   = gNewWithDouble(DoubleFloat, 3.14159265358979);

        /*  set age to an instance of the ShortInteger class  */
        age  = gNewWithInt(ShortInteger, 32);

        /*  tell some objects to print themselves  */
        gPrint(name, stdoutStream);
        gPrint(age, stdoutStream);

        /*  change the age  */
        gChangeShortValue(age, 33);



        /*  manually dispose of the objects (not necessary if the
            garbage collector is enabled)  */
        gDispose(name);
        gDispose(pi);
        gDispose(age);
}
```

3.2.5 Include File

Any source module that uses any Dynace objects must include `generics.h`. This may be accomplished with the following line:

```
#include "generics.h"
```

3.2.6 Declarations

All objects (classes, instances, methods...) in Dynace are *objects*. The declaration used in Dynace is `object`. Therefore, if you wanted to declare three objects in Dynace you could use the following:

```
object   a, b, c;
```

This declaration may be used anywhere you can define any other type of variable in the C language.

3.2.7 Initializing Dynace

The first thing that must be done prior to using any classes in Dynace is to initialize the Dynace kernel and system classes.

This initialization *must* be called from the `main()` function. The initialization function *must* also be passed the address of the `argc` argument to the `main()` function. This information is required by the Dynace garbage collector.

Use of the following is required to initialize the Dynace system:

```
InitDynace(&argc);
```

3.2.8 Generic Functions

In Dynace, generic function names, by convention only, always begin with a lower case "`g`" or "`v`", followed by an upper case letter, and then optionally followed by other letters (of any case), numbers, or underscores. This convention allows a user to immediately know which functions may be used polymorphically or overloaded.

Generic functions are used like any other C language function. However, generic functions always have at least one argument. This first argument is the object that will be performing the requested method. Any remaining arguments will be passed to the appropriate method.

When using variable argument generics (those beginning with "`v`") the number and types of arguments may be different from one call to a particular generic function to another. This is because different methods, with different argument requirements, may be called.

3.2.9 Creating Objects

By convention, the generic function used to create new objects is called `gNew()` or `vNew()`. The first argument to either should be the class of the new object (or instance) to be created. Any possible remaining arguments (in the case of `vNew`) will be the ones required by a particular class.

If compile-time argument checking is used, then the `gNew` function will possibly be renamed to avoid argument conflicts.

The following are examples of some instances being created:

```
object   name, pi, age;

name = gNewWithStr(String, "Tom Swift");
pi   = gNewWithDouble(DoubleFloat, 3.14159265358979);
age  = gNewWithInt(ShortInteger, 32);
```

There are a number of interesting things about this example. First of all, the syntax is normal C language syntax. Second, the three objects created, each of a different type, had the same declaration. Note also the standard naming of the generic function `gNewXXXX()`.

3.2.10 Using Objects

Once an object is created, it may be used as arguments to generic functions. For example:

```
object   myInt;

myInt = gNewWithInt(ShortInteger, 42);
gPrint(myInt, stdoutStream);
gChangeShortValue(myInt, 68);
gPrint(myInt, stdoutStream);
```

When the `gPrint()` generic function is called, as in the above example, Dynace checks the class of `myInt` and finds that it is of type `ShortInteger`. It then finds a method that is associated with the `gPrint()` generic function / `ShortInteger` class combination and invokes it. This association is defined by the person who wrote the `ShortInteger` class.

3.2.11 Disposing of Objects

All objects in Dynace are allocated from the heap. Therefore, when an object is no longer needed, it must be returned to the heap. If this is not done you would eventually run out of heap space. Dynace has two methods of accomplishing this task: disposal generics and the garbage collector.

3.2.11.1 Disposal Generics

One method Dynace uses to release unused objects is through an explicit call to a disposal generic. There are two common disposal generics. One is called `gDispose` and the other is `gDeepDispose`.

`gDispose` is typically used to dispose of a single object and any associated allocated space it may reference.

The `gDeepDispose` generic, in addition to performing the same functions as the `gDispose` generic, is typically used to dispose of a whole group of related objects at the same time. This method will typically recursively call the `gDeepDispose` generic on all objects being referenced within the object being disposed.

Both `gDispose` and `gDeepDispose` always return `NULL`. This is done in order to allow the disposal generic to null out the variable that holds the object in the same expression that disposes of it.

The following is an example of the `gDispose()` generic:

```
object    age;

age = gNewWithInt(ShortInteger, 32);
gPrint(age, stdoutStream);
age = gDispose(age);
```

3.2.11.2 Garbage Collection

The second, and preferred, method Dynace has for disposing of objects is through a *garbage collector*. The garbage collector is an integral part of Dynace that constantly monitors the applications memory use. Whenever certain conditions arise, such as low memory conditions, the garbage collector automatically and transparently kicks in. The garbage collector marks all objects that are directly or indirectly accessible via the *automatic* variables in the program. It also marks any objects that are referenced by global or static variables which have been registered with Dynace. Dynace also automatically marks all classes, methods, and generic function objects. After this marking phase, the Dynace garbage collector goes through all objects and frees those that are not marked.

The simple explanation is that by using the Dynace garbage collector, you never have to worry about freeing any objects that are no longer needed. The garbage collection process requires no additional coding on the developer's part. It runs entirely automatically and transparently to the application program.

A few things to remember, though: the Dynace garbage collector may be used with the `gDispose` generic function. The garbage collector is off by default and must be turned on (as shown below). Only those objects that are referenced by automatic variables, and those globals and statics that have been registered with the garbage collector (via the gRegisterMemory generic), will be marked. Therefore, any objects that are only referenced by unregistered globals or statics will be disposed of by the garbage collector.

Note that under Win32, Dynace has special code and can automatically detect and mark unregistered globals or statics. Therefore manual registration of these memory locations is not needed under Win32.

The garbage collector may be turned on with the following line (after the Dynace system has been initialized):

```
gSetMemoryBufferArea(Dynace, 40000L);
```

Dynace offers an alternative garbage collection scheme that has important tradeoffs. See the sub-section on the Boehm Garbage Collector under the Dynace Customization & Special Techniques section.

3.2.12 Generics Files

In any given program, only a single source file should contain all the generic functions used throughout the entire application. This includes the generics used by the Dynace kernel, the class library, and any application-specific generics. The file used to define all the generics is called `generics.c`. There is also a file called `generics.h` which provides the external declarations for all the classes and generics. Both of these files are automatically generated by a program called `dpp`.

See the section documenting the `dpp` program for further details.

3.2.13 Compiling & Linking

This section describes the necessary tools and steps to create an application that uses the Dynace system.

3.2.13.1 Files

In order to compile and link programs that use the Dynace kernel and class library you will need the following files:

`dynl.h` This file is included with the Dynace system and is located in the include directory.

`generics.h`
: If your application creates its own classes, you will need a custom `generics.h` file located in your application directory. This file is automatically generated by the `dpp` program. Otherwise, the `generics.h` file included with Dynace may be used.

`generics.c`
: This is a file generated by `dpp` from the application-specific `generics.h` file. It should be created and kept in the application's source directory. It is only needed if your application creates its own classes.

`dpp.exe` This program is used to generate the `generics.c` and `generics.h` files in the application's source directory. `dpp.exe` should be kept somewhere in your normal path for executable programs.

`*.lib` The Dynace library files are located under the `lib` directory. These libraries are specific to each supported compiler and whether they are for 16 or 32-bit versions of the compiler. For 16-bit compilers, only the large memory model is supported.

3.2.13.2 Compiling & Linking

The best way to compile and link your application is to start with one of the makefiles included with the example programs included with Dynace. There are specific makefiles for each supported compiler.

3.2.14 Learning Dynace Classes

Once the basics of using Dynace objects and the mechanics of creating a program are understood, a study of the Dynace library, including what facilities it has to offer and how to use it, is strongly recommended. Nothing will get you up and running with the system faster. The true value of Dynace cannot be appreciated until a full understanding of the Dynace class library and what facilities it has to offer, as well as an understanding of the simplicity and power offered by creating your own classes, has been achieved.

3.3 Defining Dynace Classes

This section describes the syntax and procedure necessary to create Dynace classes. It is recommended that all the information in the previous sections be understood prior to continuing in this section. It is also recommended that some experience with the class library be aquired.

Each new class in Dynace is defined in its own unique source file. Multiple classes should never be defined in a single source file. However, it is possible to define a large class using several source files. This process is illustrated in one of the example programs.

Dynace uses the static function ability of the C language to encapsulate the methods. This is why it is best to define a class in a single source file.

Classes are defined in class definition files which have a "`.d`" file extension instead of the normal "`.c`" file extension. These class definition files may contain normal C code as well as the additional syntax elements described in the next subsection.

When building the application, class definition files are processed by the Dynace preprocessor (`dpp`) which produces corresponding C source files (with a "`.c`" file extension) that may be compiled with your normal C compiler. At the same time, `dpp` also performs argument checking by comparing arguments to methods and their associated generics. `dpp` also uses this information to produce the appropriate generic files.

The following subsections describe the exact syntax necessary to define your own Dynace classes.

3.3.1 Class Definition File Syntax

Class definition files may contain normal C code without restriction. In addition, the following syntax elements may be used to define a class, methods, and generics.

```
Legend:

        Items between [ ] are optional.
        ... indicates any number (including 0) of the preceding.
        | indicates a choice.
        Identifiers in caps should be replaced by the appropriate name.
        Lines beginning with // are comments.
        Lower case identifiers and other symbols are typed as-is.
```

```
// Class definition

[CLASS-SCOPE]  defclass  CLASS-NAME
        [[:]  SUPERCLASS-NAME  [[,] SUPERCLASS-NAME]...]
[{
        [[instance:]
                INSTANCE-VARIABLE-DECLARATIONS]
        [class:
                CLASS-VARIABLE-DECLARATIONS]
        [init:
                SPECIAL-CLASS-INITIALIZATION-FUNCTION-NAME ;]
}];


CLASS-SCOPE = public | private

VARIABLE-DECLARATION = [TYPE]  variable ;

TYPE defaults to object
```

```
//  Public method definition

[public]  METHOD-TYPE [RETURN-TYPE]  GENERIC [,GENERIC]...
        [,<GENERIC>]... [: METHOD-NAME]  ( ARGUMENT-LIST )
{
        METHOD-BODY
}

METHOD-TYPE = imeth | cmeth | ivmeth | cvmeth

RETURN-TYPE defaults to object

Undeclared method names default to CLASS_Xm_GENERIC

ARGUMENT-LIST - types default to object,
                first arg (object self) defaults
```

```
//  Private method definition

private METHOD-TYPE [RETURN-TYPE]  METHOD-NAME
        ( ARGUMENT-LIST )
{
        METHOD-BODY
}

METHOD-TYPE = imeth | cmeth | ivmeth | cvmeth

RETURN-TYPE defaults to object

ARGUMENT-LIST - types default to object,
                first arg (object self) defaults
```

```
//  Super syntax

GENERIC(super [ARG]  [, ARGN]... )
```

```
//  Comments

/*  COMMENT  */
```

```
//  COMMENT-TO-END-OF-LINE
```

3.3.1.1 Class Definition

This syntax element allows for a central location for all elements associated with the new class. The new class's name, an optional list of superclasses, instance variables, class variables, and class initialization function are all defined by this structure.

All variables default to type `object` if not otherwise declared.

Since most elements of a class definition are optional, the minimum that is required is as follows:

```
defclass   MyClass;
```

This defines a class called `MyClass` that has no specific class or instance variables and has a single superclass (`Object`).

The next example defines a class that is a subclass of `SomeClass`.

```
defclass   MyClass : SomeClass;
```

The next example defines a class that is a subclass of `SomeClass` and `SomeOtherClass`.

```
defclass   MyClass : SomeClass, SomeOtherClass;
```

The next example defines a class that has a couple of specific instance variables associated with it. This class defaults to the single `Object` superclass.

```
defclass   MyClass  {
        int     iAge;
        object  iName;
};
```

The next example is the same as the previous except that since variables default to the `object` type, this type declaration may be omitted.

```
defclass   MyClass  {
        int     iAge;
        iName;
};
```

The next example adds to the previous example by adding a class variable. Note that since there is no type explicitly declared, it defaults to `object`.

```
defclass   MyClass  {
        int     iAge;
        iName;
 class:
        cPeople;
};
```

The next example adds to the previous example with the addition of a specific superclass (as shown above).

```
defclass   MyClass  :  SomeClass  {
        int     iAge;
        iName;
 class:
        cPeople;
};
```

The next example adds to the previous example with the addition of a class initialization function. This feature is rarely needed but allows a specific function to be executed by Dynace when the class is being initialized. This is typically used to initialize class variables.

```
defclass   MyClass  :  SomeClass  {
        int     iAge;
        iName;
 class:
        cPeople;
 init:  my_init_function;
};
```

The next example illustrates the fact that the various sections may be in any order; however, if the instance section is not first, it must be explicitly declared.

```
defclass   MyClass  :  SomeClass  {
 init:  my_init_function;
 class:
        cPeople;
 instance:
        int     iAge;
        iName;
};
```

The default scope of class definitions is `private`. Therefore the previous example could just as well have been written as follows.

```
private  defclass   MyClass  :  SomeClass  {
 init:  my_init_function;
 class:
        cPeople;
 instance:
        int     iAge;
        iName;
};
```

Use of the `private` keyword is strongly discouraged. It was only added to complement the `public` scoping which is also strongly discouraged but needed in certain circumstances.

The `public defclass` declaration is used to cause the Dynace preprocessor (`dpp`) to create additional files that contain the class and instance variable structures. This is the same as the `-iv` and `-cv dpp` options. See the `dpp` section for further explanation of this feature.

3.3.1.2 Public Method Definition

Public methods are only accessible through generic functions. The body of public methods is identical to normal C functions. The main difference between a C function definition and a public method is that a method of associating a method with generics and determining the method type is required.

The following table lists the available method types:

`imeth` This indicates an instance method with compile time argument checking enabled. Use of this option requires that all methods associated with a specific generic function have the exact same argument signature (even if they take a variable number of arguments ("...")).

`cmeth` This indicates a class method with compile-time argument checking enabled. Use of this option requires that all methods associated with a specific generic function have the exact same argument signature (even if they take a variable number of arguments ("...")).

`ivmeth` This indicates an instance method with compile-time argument checking turned off. Use of this feature allows methods with totally arbitrary argument signatures to be associated with the same generic function.

`cvmeth` This indicates a class method with compile-time argument checking turned off. Use of this feature allows methods with totally arbitrary argument signatures to be associated with the same generic function.

The following example illustrates a minimum method declaration. Note that a generic name is specified instead of the method's name. This is done since all external access to a method is through its associated generic; the actual method name is rarely important. Dynace will default a method's name to be the associated generic name prefixed with the class name, a code indicating the method type, and the generic name. Each part is separated with an underscore. So, in the following example the actual method name will be `MyClass_im_gMyGeneric`. You can, however, explicitly name a generic. This will be shown in other examples.

```
public  imeth   object  gMyGeneric(object self)
{
        return self;
}
```

Since *all* methods take `object self` as their first argument, Dynace defaults to this information. Therefore, the above example may also be written as follows.

```
public  imeth   object  gMyGeneric()
{
        return self;
}
```

If the method took additional arguments, the first argument may optionally be skipped. For example:

```
public  imeth   object  gSomeGeneric(object self, int age, char *name)
```

may also be written as:

```
public  imeth   object  gSomeGeneric(int age, char *name)
```

In addition, all argument types default to `object`; therefore, any arguments that are of type object may exclude their type declaration. For example:

```
public  imeth   object  gSomeGeneric(object self, int age, object name)
```

may also be written as:

```
public  imeth   object  gSomeGeneric(int age, name)
```

Also, since `public` and the return type of `object` are Dynace's defaults the first example may also be written as follows. Explicit use of the `public` declaration is provided in order to be consistent with the `private` declaration and is not especially recommended.

```
imeth   gMyGeneric()
{
        return self;
}
```

It is often convenient to associate one method with several generics. This can be done as follows.

```
imeth   gMyGeneric, gGeneric2, gGeneric3 ()
{
        return self;
}
```

Explicit method naming is sometimes needed if it is to be directly evoked within the same source file. This can be done as follows.

```
imeth   gMyGeneric : SomeMethod ()
{
        return self;
}
```

Explicit method naming can also be used with multiple generic associations as follows.

```
imeth   gMyGeneric, gAnotherGeneric : SomeMethod ()
{
        return self;
}
```

The following example is a duplicate of the previous example, except that it returns a `char *`.

```
imeth   char * gMyGeneric, gAnotherGeneric : SomeMethod ()
{
        return "String";
}
```

The following example illustrates the syntax used to associate a single method with generics that have argument checking turned on (imeth's) and generics that have them

turned off (ivmeth's). This syntax is only recognized by the `dpp` program when its `-X` option is enabled; otherwise this special syntax is just ignored.

```
imeth   gMyGeneric, <vAnotherGeneric> ()
{
        return "String";
}
```

To use this feature the method must be declared as an `imeth` or `cmeth`. You then surround the generic name, which is to be treated as an `ivmeth` or `cvmeth`, in angled brackets, as shown. All the previously described syntactical elements may be used without restriction when this option is used. And again, generic names in angled brackets will be ignored by the system unless the `-X dpp` option is used.

3.3.1.3 Variable Arguments

Dynace methods may be defined to take a variable number of arguments (e.g. "..."). If all methods with the same name have the same number and type of fixed arguments then the method should be declared as an `imeth` or `cmeth`. However, if the fixed arguments are not the same, then `ivmeth` or `cvmeth` should be used.

The method argument list should end in "`...`". However, internal to the method, the variable `_rest_` will automatically be initialized to the variable argument list (`va_list`). Normal `va_arg` may be used to obtain the arguments. `va_end` should not be used on `_rest_`.

`ivmeth` or `cvmeth` should be used only when the fixed argument list is different among methods with the same name.

3.3.1.4 Private Method Definition

Private methods are methods that are not associated with any generic and are therefore only available for use within the file they are declared in - similar to static functions. However, like public methods, private methods always take `object self` as their first argument.

Private methods are mainly used as a way of creating method-like functionality in cases where they are only used internally and no dynamic binding is needed. They are more efficient.

The following declares a typical private method. Unlike public methods, private method declarations name the method and have no generics associated with them. Once declared, private methods may be used like regular functions.

```
private imeth   object  MyMethod(object self)
{
        return self;
}
```

Like public methods, private methods default to their return type and argument list. Therefore, the above declaration may also be written as follows.

```
private imeth   MyMethod()
{
        return self;
}
```

3.3.1.5 Super Messages

Super messages are a way of evoking methods associated with a superclass of an object, as opposed to the method that is directly associated with the class. Typically, this is used to allow a method to wrap functionality around functionality already provided by a superclass. It is often used in the creation of new instances.

Typically, if you wanted to evoke the functionality associated with an instance, you would use the following syntax.

```
gSomeGeneric(someInstance, someArg);
```

However, if you wanted to instead evoke the method associated with the superclass if your instance, you would use the following syntax.

```
gSomeGeneric(super  someInstance, someArg);
```

As a more complete example, let's say you have a class containing an instance variable that you wish to be initialized to some value whenever a new instance is created. The following illustrates the point.

```
defclass  MyClass  {
        iSomeObject;
};

private imeth init()
{
        iSomeObject = gNew(String);
        return self;
}

cmeth  gNew()
{
        object  obj = gNew(super self);  // line 1
        return init(obj);
}
```

In this example the instance variable is being initialized to a null string object for every new instance created. Remember that since this is a class method, `self` will be a class object. Since only one of the kernel classes knows how to actually create a new (blank) object, the `gNew` method must invoke the one above it in order to finally get to the one that can actually create the bare new object. Once the new object is obtained, it may be initialized and returned.

Note that you couldn't have just called `gNew(self)` because that would call the same method you're in to be called recursively and indefinitely.

In addition, since sending the `super` message to `self` is the most common use of `super`, the `self` is actually optional. Therefore, line 1 above could be written as follows:

```
object  obj = gNew(super);  // line 1
```

3.3.1.6 Comments

In addition to normal C comments, the `//` is used to signal comments to the end of the line.

3.3.2 Example Code

The following example illustrates a complete program that defines a new Dynace class. Each line is explained in the following sub-sections.

```
defclass  MyNewClass  {
        char    *iName;
        int     iAge;
        iSomeOtherObject;
 class:
        long    cTotalInstances;
        int     cAverageAge;
};

private imeth init(char *name, int age)
{
        iName = (char *) malloc(strlen(name)+1);
        strcpy(iName, name);
        iAge = age;
        return self;
}

cmeth  gNewInstance(char *name, int age)
{
        return init(gNew(super), name, age);
}

imeth   object    gDispose()
{
        free(iName);
        return gDispose(super);
}

imeth   object   gDeepDispose()
{
        free(iName);
        if (iSomeOtherObject)
                gDeepDispose(iSomeOtherObject);
        return gDispose(super);
}

imeth   gSetObject(obj)
{
        iSomeOtherObject = obj;
        return self;
}

imeth   gPrint(FILE *fp)
{
        fprintf(fp, "%s is %d years old\n", iName, iAge);
        return self;
}
```

3.3.3 Class Definition

The first step in defining a new class would be to create a `.d` file with a `defclass` specification. This is discussed in detail in the section that describes the `defclass` syntax.

Note the naming convention used. Class names begin with an upper case letter and are followed by upper and lower case letters. Instance variables start with an "`i`" and are followed by an uppercase letter. Likewise, class variables start with a "`c`" and are followed with an upper case letter.

This naming convention is just that – a convention. Dynace does not enforce or necessarily encourage this convention. However, since Dynace allows direct access to instance and class variables from within methods, some naming convention is desirable; otherwise it would be difficult to distinguish instance variables from locally declared variables.

3.3.4 Defining Methods

Once the `defclass` specification is given, the methods may be declared. The syntax for this, as well as several examples, is discussed in the section that describes method syntax.

It is important to understand that `self` will always refer to the instance object that called the generic, if it is an instance method, and `self` will always refer to the class object if this is a class method.

The naming convention used is that generics with a fixed number and type of arguments in which compile-time argument checking is enabled, start with "`g`" and are followed by an uppercase letter.

Generics without compile-time argument checking enabled may be passed arbitrary arguments. These generics start with "`v`" and are followed by an uppercase letter.

This naming convention is not enforced or necessarily encouraged by Dynace. However, it is intended to help a programmer distinguish normal C functions from various points where dynamic binding is occurring.

3.3.4.1 Accessing Instance And Class Variables

An object's instance and class variables may only be accessed from within a routine located in the source file that defines the class. There are several ways available to access these variables. This subsection will discuss all the ways from the most frequently used methods to the least.

The most frequent use of instance variables is from within public or private instance (not class) methods. These methods may directly access the variables as they would access any local variable, by just using them by name.

Class variables are also directly accessible by name in *all* methods or regular C functions within the module.

There are four places where instance variable access gets a little tricky. Dynace, however, provides mechanisms to conveniently access instance variables in all circumstances.

The first difficulty is when attempting to access a newly created instance's instance variables from within a class method that has just created the new object. This is a common

problem within `New` generics. There are two solutions to this problem. One solution is clean but a little less efficient than a more dirty solution. Understanding the dirty solution, however, is helpful when debugging code (regardless of which solution is used).

The clean solution is depicted in the following example.

```
private imeth  init(char *name, int age)
{
        iName = (char *) malloc(strlen(name)+1);
        strcpy(iName, name);
        iAge = age;
        return self;
}

cmeth   gNewInstance(char *name, int age)
{
        return init(gNew(super), name, age);
}
```

In the above example, `gNewInstance` first calls `gNew` to create a new instance. It then passes the new instance, along with the initialization arguments to an instance method (`imeth`) because instance methods can access the new instance's instance variables. Since the `init` method returns the object passed, `gNewInstance` can simply return it.

One interesting point about the above example is that private methods, unlike public ones, must either be defined or declared prior to use.

With the dirty solution it is important to understand that Dynace provides all instance variable access through a structure pointed to by a locally declared pointer named `iv` (which stands for "instance variables"). In the case of instance methods, this procedure is automatically handled by the Dynace preprocessor. However it must be done explicitly when needed, in class methods.

Dynace also declares a typedef, which is a structure mirroring the instance variable definition contained within the `defclass` statement. Finally, Dynace provides macros for obtaining a pointer to the instance variables associated with a given object.

The following example depicts the solution to the above-mentioned problem.

```
cmeth   gNewInstance(char *name, int age)
{
        object  obj;                    /*  line 1  */
        ivType  *iv;                    /*  line 2  */

        obj = gNew(super);              /*  line 3  */
        iv = ivPtr(obj);                /*  line 4  */
        iName = (char *) malloc(strlen(name)+1);
        strcpy(iName, name);
        iAge = age;
        return obj;
}
```

This example demonstrates all the above procedures. Line 1 declares a variable that will hold the new instance object to be created. Line 2 declares the `iv` variable which Dynace needs in order to access the object's instance variables.

Line 3 calls the super `gNew` in order to create a new and uninitialized instance object and assign it to `obj`. Line 4 gives Dynace access to the instance variables associated with the `obj` variable.

After line 4, the code may directly access the instance variables as would be done in an instance method.

The second problem associated with accessing instance variables has to do with normal C functions contained within the same module. Since C functions don't necessarily take the self argument, it is impossible for Dynace to know how to access the instance variables. However, a simple method of achieving this is provided.

As stated above, Dynace uses the `iv` variable in order to access instance variables; therefore, all that has to be done is to pass the `iv` pointer to the C function. For example:

```
static  void    MyFunc(ivType *iv, int age)     /*  Line 1  */
{
        iAge = age;                             /*  Line 2  */
}

imeth   gChange(int age, char *name)
{
        MyFunc(iv, age);                        /*  Line 3  */
        iName = name;
        return self;
}
```

In the above example, a normal C function is given access to the instance variables by passing it the noted `iv` pointer. Since Dynace automatically gives instance methods access to an `iv` pointer, it may be simply passed to the function (as shown in line 3).

Line 1 shows how the `iv` variable would be declared. It doesn't necessarily have to be the first argument. Line 2 shows how the instance variables are now directly accessible within the function.

The third problem associated with accessing instance variables has to do with accessing the instance variables of more than one object within the same module at the same time. The problem can be resolved (in a relatively inefficient, but clean manner) with instance variable accessing methods. However, Dynace provides a very efficient way to accommodate this situation.

Since Dynace gives a method of accessing the instance variable structure pointer associated with an object (`ivPtr` macro), these pointers may then be used as normal C structures. For example:

```
imeth   gCopy()
{
        object  obj2;                   /*  Line 1  */
        ivType  iv2;                    /*  Line 2  */

        obj2 = gNew(super MyClass);     /*  Line 3  */
        iv2  = ivPtr(obj2);             /*  Line 4  */
        iv2->iAge = iAge;               /*  Line 5  */
        return obj2;                    /*  Line 6  */
}
```

The above example defines an instance method that produces a new instance of the class and initializes it to have the same age as the instance object passed. Line 1 declares the variable that will hold the new object being created.

Line 2 declares a pointer that will point to the instance variables contained within `obj2`. Note that you couldn't use the name `iv` because that would conflict with Dynace's use of that variable relative to the implied `self` argument.

Line 3 creates a new blank instance of the class. Line 4 gives access to the instance variables contained within the new object.

Line 5 is the neat part. The left side of the assignment references an instance variable associated with `obj2`, and the right side references an instance variable associated with the object passed (self). This line could have also been written as follows:

```
        iv2->iAge = iv->iAge;
```

Dynace automatically creates the `iv` structure pointer and associates it with `self`. The key point here is that all direct instance variable usage is always relative to the `iv` variable.

The final difficulty associated with instance variable access has to do with the C preprocessor. The following example will cause problems.

```
#define ASSIGN  iAge = age

imeth   gChangeAge(int age)
{
        ASSIGN;
        return self;
}
```

The above example will cause problems because Dynace will scan the method and determine that since it can't find any instance variable names referenced in the method, there is no need to gain access to them. Therefore, Dynace will not generate the `iv` variable for this method and you will receive a compile error.

This problem may be corrected by explicitly requesting Dynace to give access to the instance variables by use of the `accessIVs` macro. The following example corrects the previous example as discussed.

```
#define ASSIGN  iAge = age

imeth   gChangeAge(int age)
{
        accessIVs;
        ASSIGN;
        return self;
}
```

This macro must be placed in the variable declaration portion of the instance method's definition. The `accessIVs` macro actually declares and initializes a variable called `iv` from the `self` argument to the method. This `iv` variable will point to a normal C language structure that will look like the structure you defined when defining the instance variables.

3.3.4.2 Determining the Class of an Object

There is a macro in Dynace that will return the class of any object. This macro is called `ClassOf`. The following example illustrates its usage:

```
object  o1, o2;

o1 = ClassOf(o2);
```

Now `o1` will contain the class of `o2`.

3.3.4.3 Sending Super Messages

The process of evoking a method that is located in a superclass is sometimes necessary when executing a method within a particular class. Normally, when a generic function is evoked, the search for an applicable method starts at the class of the object that is the first argument to the generic. If no method is found, the superclasses are then searched. When sending a super message, the superclasses of the object are searched first. If no method is found, then the superclasses of the superclasses are searched, and so on. The class of the object is bypassed.

An example will clarify this concept. Let's say you have a class defined as follows:

```
defclass  MyClass  {
        int     iA;
};

imeth   gPrint()
{
        printf("iA = %d\n", iA);
        return self;
}
```

This is a simple class that just holds the integer `iA` and has a method to print it. Now let's define a second class that inherits from the above class. The second class will look as follows:

```
defclass  AnotherClass : MyClass  {
        int     iB;
};

imeth   gPrint()
{
        gPrint(super self);
        printf("iB = %d\n", iB);
        return self;
}
```

Not that both class's print methods share the same generic function, `gPrint`.

Since the second class inherits from the first class, instances of the second class will actually contain both the integer `iB` and also the integer `iA` defined in the first class. Since the second class's print method has no direct access to the `iA` instance variable, because of strict encapsulation, there is no way for it to print the `iA` instance variable's value. When the `self` argument to the `gPrint` generic function is of the second class's type, the second class's print method would always be invoked. There needs to be a way to invoke the print method associated with the first class from the print method in the second class.

In the second class's print method, the super syntax shown would accomplish the desired effect. This syntax works like invoking the `gPrint` method on an instance of the current class, except that it starts its search with the superclasses.

The end result of all this is that when a `gPrint` generic function is called with an instance of the second class, the second class's print method will be invoked. Then the second class's print method will invoke the first class's print method, printing the instance variable `iA`. When the first class's print method is through, it returns to the second class's print method, which called it. After that, the second class's print method prints the `iB` instance variable and returns.

This super message concept is very important to understand. It is one of the key features that allow you to take a pre-existing class and modify its behavior to suit your needs without having to rewrite the pre-existing code.

3.3.5 Class Initialization

During startup, when your application executes `InitDynace`, Dynace initializes the Dynace kernel, the generic objects, and the Dynace kernel classes. The remaining classes get initialized when they are used. If you use a class that is a subclass of other classes, Dynace automatically initializes the parent classes first.

Explicit class initialization may be accomplished by simply referencing the class. The following example, which should exist inside a function, explicitly initializes the `MyClass` class.

```
MyClass;
```

3.3.5.1 Class Initialization Function

Sometimes it is desirable to perform some additional procedures at the time the class is initialized. For this case it is possible to have Dynace evoke class-specific initialization functions declared by the programmer. Most often this feature is used to initialize class variables.

A class-specific initialization should be a static function that take no arguments and returns no results. This function will get automatically evoked by Dynace immediately after the creation of the class it's associated with. Therefore, it will have full access to all the class variables.

In order to inform Dynace of the existence of a class initialization function, it should be declared in the `defclass` statement as described in the `defclass` specification detailed in a previous section.

3.3.5.2 An Example

The following code fragment depicts a class initialization function being used.

```
static  void    class_init(void);

defclass  MyClass  {
        int     iVar;
 class:
        object  cVar;
 init:  class_init;
};

static  void    class_init()
{
        cVar = gNew(Set);
}
```

3.3.6 What Gets Linked

Regardless of what classes exist in a particular library or what classes are mentioned in the `generics.h` or `generics.c` files, Dynace is designed in such a fashion that at link time only the classes which your application actually uses will get linked in.

3.3.7 The gNew Class Method

Most classes will have an instance creation method. This is the method that is responsible for creating and initializing instances of the new class, also called the *constructor*. If no local instance variable initializing is necessary, then no constructor method needs to be defined. One of the superclasses, or the Dynace kernel, will automatically create and initialize the instance. In the absence of any explicitly initialized instance variables in a new instance, the Dynace kernel will zero out all instance variables of a new object.

The `gNew` class method works like any other class method. It receives, as its first argument, the object representing the class. If the constructor needs to take additional arguments you can either use the `vNew` generic (which has no compile-time argument checking so the method can have any signature) or you can use a different generic name (such as `gNewInstance`) which will have the desired argument signature.

The first thing the `gNew` method normally does is create an instance of the class. This can be done by sending a super message to `gNew`. It must then pass the new object to an instance method so that the instance variables can be accessed and initialized.

```
defclass  MyClass  {
        char    *iName;
        int     iAge;
};

private imeth  init(char *name, int age)
{
        iName = (char *) malloc(strlen(name)+1);
        strcpy(iName, name);
        iAge = age;
        return self;
}

cmeth  gNewInstance(char *name, int age)
{
        return init(gNew(super), name, age);
}
```

Notice how the above example was also able to allocate additional storage for `iName`.

3.3.8 The Dispose Instance Method

Any class whose instances must do something more when being disposed of than just freeing themselves, must define an instance disposal method. This method, called the *destructor*, is usually associated with the `gDispose` and `gDeepDispose` generic functions.

In the example in the previous section, one of the instance variables (`iName`) contained a pointer to allocated storage. If the instance was freed without regard for this allocated storage, the system would soon run out of heap space. Even the garbage collector couldn't help. So what we do is define a destructor that will free the instance appropriately. The following code will illustrate this procedure:

```
imeth   object    gDispose()
{
        free(iName);
        return gDispose(super);
}
```

There are several things that may be noted from this example.

First, the method returns what is returned from the `gDispose` super call. By convention the disposal generics return `NULL`. This is often used to null out a variable that references an object which is being disposed of and hence no longer valid.

Second, the call to free frees the storage allocated and assigned to the `iName` instance variable. It does not free the entire instance.

Third, the call to the super method is used to request any superclasses to perform their freeing. Eventually, the super call will terminate at the Dynace `Object` class, which will do the actual freeing of the entire instance.

As discussed before, there are two ways to dispose of an object. The first, is by explicitly evoking the `gDispose` generic function on an object. The second, and preferred, way is to allow the garbage collector to dispose of the object when necessary.

If no special processes need to be performed to free an object, no destructor need be defined. Dynace has a default disposal procedure that simply disposes of the individual object.

3.3.9 Creating Generic Files

The `generics.h` and `generics.c` files are created by the `dpp` program. `dpp` reads in the pre-defined classes and generics from a pre-existing `generics.h` file as well as user added classes and generic information from class definition (`.d`) files. `dpp` is able to parse the necessary files and merge the information in order to obtain sufficient information to generate the correct generic files for the application. See the section describing `dpp` for a complete description of it.

3.4 Dynace Pre-Processor (DPP)

The `dpp` program is the heart of the compile-time functionality of Dynace. `Dpp` is capable of reading in pre-existing generics declaration files (normally `generics.h`) and class definition files (`*.d` files) and using this information to perform five specific tasks.

First, `dpp` is able to assimilate all the information read in and produce a merged generics declaration file (normally `generics.h`), which can be included by the application. This file includes all necessary declarations for the entire application, including class and generic declarations and system includes.

Second, `dpp` is also able to use the information read in to produce an application-specific `generics.c` file. This is the file that contains all the actual generics as well as the generic and class initialization procedures.

Third, `dpp` is able to read in a class definition file (`*.d` file) and produce a normal C file (`*.c`) for compilation by your compiler.

Fourth, `dpp` will validate the argument and return type declarations between methods and pre-existing generics to make sure there are no inconsistencies. This will only be done for methods which have argument checking turned on. If an inconsistency is encountered, `dpp` issues an error message but does not stop its processing. It will, however, optionally return an error code.

Fifth, it provides facilities so that the application-specific `generics.c` file only needs to be generated and compiled when absolutely necessary. It does this by comparing any pre-existing `generics.h` file with one it's about to create. If they are the same (i.e., no new generics or classes have actually been added), then instead of generating the file, the `generics.h`, `generics.c` and `generics.obj` files are just touched.

3.4.1 Controlling DPP

`Dpp` operates by first reading in all input files (regardless of their order on the command line) and then using this information to check the input consistency and produce any requested output files.

`Dpp` is non-interactive; all operational requests and parameters are given at the command line. However, these command line arguments may also be located in files which `dpp` can read as an extension of the command line. These files, called command files, may contain any number of parameters in the same format as the normal command line. Parameters may be spread across any number of lines because `dpp` ignores new lines, tabs and multiple spaces located in one of these command files.

Command files are identified by a "@" followed by a file name with no intervening spaces. Command files and normal command line parameters may be freely intermixed. For example, the following is quite legal.

```
dpp -g @cmdfile -h
```

Unlike the DOS standard, `dpp` only supports a hyphen or dash as an option switch. All options are case sensitive.

A complete list of options may be obtained by evoking `dpp` without any arguments. It will display a complete list of all options and exit.

3.4.2 DPP Status Messages

During `dpp` operation, it will display the name of each file it is reading or writing, in addition to any warning or error messages. At the end of its run, `dpp` will display a status message similar to the following.

```
Classes  = 48, 3, 51
Generics = 193, 15, 208
```

The first line indicates that `dpp` read in 48 class definitions via generic header files (i.e. `generics.h`). It then encountered 3 new class definitions while processing source files (i.e. `*.c, *.d`). This totaled 51 classes.

The second line indicates the same thing for generics.

3.4.3 Input Options (-g & -G)

Input options are those that cause `dpp` to read a file.

3.4.3.1 Generic Declaration Input

It is important that `dpp` have complete access to all related class and generic declarations when processing any files. This is the only way `dpp` can have sufficient information to adequately verify system consistency and produce complete generic files.

Typically, the single `generics.h` file associated with the application is all that is needed. The main exception to this rule is when initially building the application-specific `generics.h` file or when wanting to build a new, clean one. In that case, you should read in all the `generics.h` files associated with all the subsystems you intend to use.

As shipped, Dynace only includes two such files in the include directory. One called `generics.h` includes all the classes and generics defined by the Dynace kernel and class library. A second, called `wingens.h`, includes all that's in the `generics.h` file plus all the classes and generics associated with the Dynace Window Development System. You would use one or the other, not both.

The command line argument used for this purpose is `-g`. Immediately following this option may be a list of zero or more, space-separated, generic file names to be read in. There is always a space after the `-g`. If no file names follow the option, the single `generics.h` file is assumed.

There is also a similar `-G` option which works the same except that it won't return an error code if one of the files can't be found.

3.4.3.2 Source File Input (-s & -p)

`Dpp` includes two options for reading in source files. One command is for reading only, and the other is for reading and pre-processing.

The `-s` option is for reading only. It reads in the specified files and uses this information for consistency checks and producing complete generic files. It is capable of reading in the new class definition files (`*.d`) as well as the old Dynace 2.x class definition files (`*.c`).

The `-p` option is for reading and pre-processing class definition files (`*.d`), thus producing associated C source files (`*.c`) with the same file name prefix. The information read in is also used as in the `-s` option; therefore, it would never be necessary to include the same file for processing under the `-s` and `-p` options.

In addition to recognizing `*.d` files as class definition files, `dpp` will likewise recognize `*.dd, *.n, *.nn, *.i`, and `*.ii` in upper or lower case. Input file names ending in `.dd, .nn`, or `.ii` will produce corresponding `.cc` or `.cpp` files depending on whether you are on Unix or DOS respectively.

3.4.4 Output Options

Output options are those options that cause `dpp` to create a file.

3.4.4.1 Generic Declaration File Creation (-h)

Once all the pre-existing generic declaration files (typically `generics.h`) and class definition files have been read in, `dpp` is then able to produce a merged class and generic declaration file for inclusion by application code.

The `-h` option is used for this purpose. The `-h` option may, optionally, be followed by a space and file name. If the file name is given, `dpp` will use that file for the generic class definition file. Otherwise, the default of `generics.h` will be used.

3.4.4.2 Generic Source File Creation (-c)

Use of the `-c` option causes `dpp` to generate the application-specific C source file which contains all the generics used by the application as well as generic and class initialization code. Typically, the application-specific generics declaration file (`generics.h`) contains all the information needed for the creation of this file.

The `-c` option may, optionally, be followed by a space and file name. If the file name is given, `dpp` will use that file for the generic file. Otherwise, the default of `generics.c` will be used.

If the `generics.c` file gets too large for your compiler environment see the Splitting the Generics File (-M) `dpp` option.

3.4.4.3 Processing Class Definition Files (-p)

`Dpp` is capable, after reading in the appropriate generics declaration file, of reading a class definition file, verifying its consistency with the system, and generating a corresponding C source file for compilation.

The `-p` option is for reading and pre-processing class definition files (`*.d`), thus producing associated C source files (`*.c`) with the same file name prefix. The information read in is also used as in the `-s` option; therefore, it would never be necessary to include the same file for processing under the `-s` and `-p` options.

Input files ending in `.dd` will produce corresponding `.cpp` or `.cc` files, depending on whether you are on Unix or DOS.

3.4.4.4 Java Interface Files (-j)

Dynace has the ability to call Java code and have Java code call Dynace generics. The `-j` option to `dpp` causes `dpp` to generate files used to interface with Java. This option also takes an optional package name argument. The files generated are named `object.java`, `DynaceClassRef.java`, and `LoadDynClasses.c`. These Java interface files, in concert with the Dynace / Java interface DLL named `JavaDynace.dll`, make the interface possible.

3.4.4.5 Scheme Interface Files (-L1 & -L2)

`Dpp` has the ability to generate interface code to allow Dynace generic functions to be accessible from within a Scheme language environment. Once the `generics.h` file is read in (via the -g option), these options cause `dpp` to generate the necessary interface files for two supported Scheme environments, LibScheme and MzScheme. This interface file may be named after the option and will default to `scminter.c`. This file must be compiled and linked with your application.

MzScheme is available at `http://www.plt-scheme.org`

3.4.5 Misc. Options

`Dpp` supports a variety of other options documented in this section. Some options, however, are not documented. This is because some features are either experimental and not ready for use or are only used to support old versions of Dynace, which are no longer supported in any other fashion.

3.4.5.1 Preventing Unnecessary Compiles (-t)

Generic files (both .c and .h) change whenever classes or generics are added or deleted, or arguments to existing generics change. Whenever a class definition file is changed, it must be pre-processed to produce a compilable C file. Also, when class definition files change, it may or may not be the case that new classes or generics have been added or existing generics have changed argument types. In fact, except for a new development project, adding classes and generics is relatively rare when compared to other code changes.

Given this fact, `dpp` is able to determine whether a significant change has occurred or not. It does this by comparing the pre-existing application-specific generics declaration file (`generics.h`) with one it's about to create. If they are the same, `dpp` does not create the new generic file and instead remembers the fact that it was not necessary.

If it wasn't necessary to generate the generic declaration file and the `-t` option is given, `dpp` will touch the specified files in order to prevent the make utility from unnecessarily building and compiling the corresponding `generics.c` file. This can lead to a significant savings in time.

The `-t` option may, optionally, be followed by a space delimited list of file names to touch. If the file list is given, `dpp` will touch the given files in the order specified. Otherwise, the default of `generics.h generics.c generics.obj` will be used. On Unix systems `generics.o` is used instead. If any of the files don't already exist, `dpp` will just skip it.

3.4.5.2 Removing Classes or Generics (-r)

It is sometimes necessary to cause `dpp` to remove classes or generics from an existing `generics.h` file. This is necessary when an old class or generic is no longer used, or when the return type or arguments to a generic have been changed. If the return type or arguments to a generic have been changed and you don't want to rebuild the `generics.h` file from scratch, it's often easier to remove the old generic from the `generics.h` file and then just re-scan the source file with the new generic definition to re-add the changed generic. Once the `generics.h` file has been corrected, a new `generics.c` file can be generated from the correct `generics.h` file (with the `-g -c` options).

To use this option when generating generics files simply follow the `-r` option with a list of classes and/or generics to remove (each separated by a space).

3.4.5.3 Splitting the Generics File (-M)

As a system becomes increasingly complex, the number of generic functions increases. `Dpp` typically generates a single `generics.c` file which contains all the generic functions defined within the system. The problem with this is that in some limited environments (DOS or 16 bit Windows) there is a limit to the size a single source (`.c`) file can be, so `generics.c` gets too big to be compiled. The solution is to split `generics.c` into multiple smaller files. Note that there isn't any problem associated with the size of the `generics.h` file in these environments.

The `-M` option causes `dpp` to split the `generics.c` file at the time it is created (using the `-c` option). As part of the `-M` option, you must include the number of generic functions to split at. This option is not separated with a space. For example, using `-M900` will cause the first 900 generic functions to be defined in `generics.c` and the remaining generic functions will be placed in a file named `gens1.c`. `Gens1.c` will have to be compiled and linked into the application just as `generics.c`.

3.4.5.4 Including Additional Header Files (-Isc, -Iac, -Ish & -Iah)

It is sometimes necessary to cause additional header files to be included in the `dpp` generated `generics.c` or `generics.h` files. This occurs when generic return or argument types are defined in other include files. Therefore, these files must be included prior to declaring these generics. `Dpp` has the `-I` series of options in order to support this feature.

Any particular `-I` type option should be followed by a space and a space-delimited list of file names to include. `Dpp` will cause the specified files to be included by the generated `generics` file in the specified order.

Use `-Isc` for system include files to be added to the `generics.c` file. Use `-Iac` for application include files to be added to the `generics.c` file. Use `-Ish` for system include files to be added to the `generics.h` file. And use `-Iah` for application include files to be added to the `generics.h` file.

3.4.5.5 Ignoring Errors (-i)

Normally, while processing the input files, if `dpp` encounters some kind of error, it displays the error message and continues processing the input files. However, if an error occurs,

`dpp` does not normally produce any output files. The `-i` option causes `dpp` to produce the output files requested anyway.

3.4.5.6 Quiet Operation (-q)

Normally, `dpp` displays continuous status messages indicating what files it is reading and writing in order to give some idea of where it is in its process. The `-q` option may be used to cause all unnecessary messages not to be displayed. This option, however, will not inhibit error messages.

3.4.5.7 Return Code (-z)

Under normal operation, `dpp` returns 0 if no error occurs and non-zero otherwise. This is typically used to signal the make facility that an error occurred. The `-z` option is used to cause `dpp` to return 0 regardless of whether or not an error occurs.

3.4.5.8 Compile Time Argument Checking (-N)

The `-N` option is used to cause `dpp` to treat all generics as if they were defined with compile time argument checking turned off and generate code accordingly. In this case all generics will take arbitrary arguments and depend entirely on runtime argument validation.

This was the way Dynace versions 1 and 2 worked. The capability is kept for two reasons: first for backward compatibility; and second, to support environments that require complete flexibility with respect to generic arguments.

3.4.5.9 Disabling #line directives (-nld)

Normally `dpp` generates line directives when preprocessing a class definition file into a C source file. This is done in order to allow debugging on the original class definition file instead of the generated C file.

If this feature is not wanted use the `-nld` option.

3.4.5.10 Extra #line directives (-eld)

Normally `dpp` generates line directives only when necessary (when the files get out of line sync). This option causes `dpp` to (in addition to its normal #line directive output) output a #line directive prior to each method.

3.4.5.11 Preprocessing Strategies (-S)

`Dpp` is capable of supporting four different code generation strategies. The various strategies refer to different linkage conventions between generics and methods. The various alternatives offer choices that determine the generated code's portability, efficiency, and use of C++ specific features.

Class definition files will work the same regardless of which strategy is selected. However, it is absolutely critical that the entire system be compiled for one and the same strategy. This includes generic files, class definition files and the Dynace system and class libraries as well.

To use this option you would use `-Sn` where `n` is a number between 1 and 4. There is no space between the `-S` and the number. The various strategies are as follows:

Strategy 1 incorporates a small, system and compiler specific, piece of assembler code. This strategy is the most efficient. This assembler piece is supplied for many of the popular platforms. Writing the assembler piece for a new platform is very easy for an assembler expert and is documented in the next section of this manual. The only downside of this strategy is the lack of portability (if the assembler piece is unavailable). **Note:** Due to the ever-changing CPU landscape, the difficulty in maintaining this code, the lack of portability, and the decreased value due to compiler optimizations, Strategy 1 has been deprecated. Strategy 2 (the default strategy) is recommended.

Strategy 2 uses a total C-based approach. This is the default and recommended method. This strategy maximizes portability and for that reason it is the default strategy and the one used to generate the Dynace system as shipped.

Strategy 3 attempts to reduce the runtime overhead associated with Strategy 2 by using C++'s inline function facility. The resulting code will run faster but will incur a heavy penalty in code size due to the inline expansions. This strategy is not typically recommended.

Strategy 4 produces code similar to Strategy 3, except that generics which have variable arguments are treated like Strategy 2 instead. This was done in order to accommodate compilers such as GCC which cannot inline variable argument functions.

Remember, Dynace is shipped using Strategy 2. If you wish to use a different strategy, you must rebuild the entire system with the selected strategy.

3.4.5.12 Generic Overloading (-X)

Dynace allows many methods to share the same generic function. This option has no effect on that facility of Dynace.

Dynace also has the ability to simultaneously associate a single method with a fixed argument generic, which has compile time argument checking and a variable argument generic, which has no compile time argument checking, using the '`<>`' method declaration syntax. Under normal operation, `dpp` ignores the '`<>`' syntax and does not perform this association. The `-X` option is used to enable `dpp`'s support for this feature.

This facility is used to enable Dynace to support users who like maximum compile time argument checking and those who prefer maximum flexibility with regard to generic overloading with the same code base.

3.4.5.13 Force Generics.h File Generation (-f)

Under normal operation, when generating a `generics.h` file `dpp` will first read in any existing `generics.h` file and not generate a new file if the existing one is correct. The `-f` option is used to force a new `generics.h` file regardless of its status.

This feature is useful when changing strategies. If none of the generics or classes have changed, Dynace would not normally generate a new generics file. This option will force a new one, with the correct strategy, to be created.

3.4.5.14 Generating External Structures (-iv & -cv)

These options cause Dynace to generate external declarations for a class's instance and class variables, respectively, when preprocessing a class definition file. The file names used will be the same as the source file, except that the file will end in `.iv` or `.cv`, respectively.

3.4.5.15 Class Initialization (-ni)

Under normal operation, `dpp` generates the necessary class initialization file when preprocessing a class definition file. This option is used to cause `dpp` not to generate this function. The programmer will then be responsible for this function.

3.4.5.16 Auto Include Generation (-nai)

Under normal operation `dpp` automatically adds code to include the `generics.h` file in generated `generics.c` and C source files. This option causes `dpp` not to perform this auto inclusion.

This option, along with the `-mg` option, is used to support users who prefer a single header per class.

3.4.5.17 Macro Guard Typedefs (-mg)

This option causes `dpp` to macro guard all typedefs and inline generics in the `generics.h` file. This is used in conjunction with the `-nai` option to support one header per class. This is an unrecommended configuration.

3.5 Dynace Customization & Special Techniques

The Dynace kernel includes a class called `Dynace`. This class is used to query and set various parameters that affect the global operation of Dynace. The following subsections detail the various parameters that may be affected in Dynace, as well as other special issues.

3.5.1 Makefile / Generics.h File Dependencies

All source files are technically dependent upon the application-specific `generics.h` file. However, it is important that the make utility not be made aware of this fact. This is important because one of the main goals and advantages of Dynace is the fact that rarely do changes to a class require other modules to be recompiled - even when the changes cause changes to the `generics.h` file. If the makefile were made aware of this dependency, there would be many, many totally unnecessary compiles performed.

Most changes to a class definition file, including changes in the instances or class variables associated with the class, changes in the class hierarchy or algorithmic changes, have no effect on the `generics.h` file. The only things that could potentially cause a change to the `generics.h` file are the addition or removal of an entire class, the addition or removal of a generic, or the changing of the argument list or return type associated with a generic.

With a little programmer cooperation and the disassociation (for make utility purposes) of the `generics.h` file from other source modules the need for re-compiles can be very drastically reduced. The make utility and the example programs (the ones that create classes) were designed with these concepts in mind. Emulation of these approaches is highly advised.

The following subsections detail all the things that could change a `generics.h` file and what effect it has on other modules. Advice on how to handle each situation will be discussed.

3.5.1.1 Adding New Classes

The addition of a new class never requires automatic recompilation of any other modules. Since the class was just added, the only modules that could even use the new class must have also been edited in order to reference the new class object. In that case, those modules will already be automatically recompiled anyway. Other modules don't directly reference the new class object and don't require recompilation.

Remember that because of Dynace's design, any previously compiled code that doesn't explicitly access a class by name may still process and use arbitrary objects without having knowledge of those objects prior to being compiled.

3.5.1.2 Removal Of Classes

Again, the removal of a class should never cause the automatic recompilation of all modules. For starters, any module which references that class object will have to be edited, causing it to be recompiled. You can't possibly forget to edit a file because the system won't link until all references to the removed class are eliminated.

The best approach is to use `grep` or some other file search utility to find all references to the removed class and make the appropriate changes.

In addition to the above, it should be noted that `dpp` never removes a class declaration from a `generics.h` file. Of course, it really doesn't matter if an unused declaration exists; however, periodic rebuilding of the `generics.h` file from scratch will remove the accumulated junk.

3.5.1.3 Adding New Generics

Adding new generics never causes the need for mass recompilation for the exact same reasons as the addition of new classes. The only modules which could possibly use it have to be edited to put the new generic in, and that will cause only the appropriate modules to be recompiled. No other module is affected in any way.

3.5.1.4 Generic Removal

The removal of generics is a little more complex. First of all, if you remove a generic from a class definition file, Dynace doesn't automatically remove it from the `generics.h` file. In fact, the removal of a generic from a class definition file has no effect at all on the `generics.h` file. This is because without scanning all the class definition files and building the `generics.h` file from scratch, Dynace has no way of knowing whether or not any other module makes use of that generic. Therefore, Dynace leaves it in.

Because of the above, there is no advantage to associating the `generics.h` file with other code. The bottom line is that if a generic is removed, you'll have to search the code for use of that generic and make appropriate changes. This will cause the appropriate modules to be recompiled.

Another interesting thing about these facts is that rebuilding `generics.h` from scratch on a periodic basis is a good idea in order to get rid of any unnecessary generics.

3.5.1.5 Generic Argument Or Return Type Changes

Of all the possible changes to a `generics.h` file this is the most troublesome. It's also the change that is performed the least. Once an application or class library reaches any level of maturity, the existing protocols tend to change little. New features are implemented with new generics.

It's also difficult to change generic arguments because the person making the change must take into account the effect it will have on any other classes that use the same generic. Generics with argument checking turned on must agree with which all methods which they are associated. Generics with argument checking turned off are not affected by this issue.

Having said that, however, there are only two ways of dealing with this situation. First of all, if working on a team project, it should be agreed that no generics will be changed without mutual agreement. It's more common to just create a new generic.

The best way to deal with the problem is to search through all the code for references to the generic and make the appropriate changes. This will cause only the affected files to be recompiled.

Another, and poor, answer is to recompile everything and fix the modules that get a compiler error.

This issue is not bad if you think about it. What would you normally have to do if you change the arguments to a regular C function? Answer - the same thing.

3.5.2 Compile Time Argument Checking

Dynace supports the simultaneous use of generics that have compile time argument checking enabled and those that have the option turned off. By convention only, generics that have compile time argument checking turned on begin with "`g`" and those that have compile time argument checking turned off begin with "`v`".

The programmer determines which generics have compile time argument checking turned on or off at the time the generics are defined. Those introduced with `imeth` or `cmeth` have argument checking turned on and those introduced with `ivmeth` or `cvmeth` have argument checking turned off.

For those generics that have compile time argument checking enabled, the Dynace preprocessor (`dpp`) will validate the argument lists between generics and their associated methods during preprocessing time. All discrepancies between generics and their associated methods will be reported at this time. Code that uses generics with compile time argument checking enabled will be validated by your normal C compiler by use of the `dpp` generated declarations that exist in the `generics.h` file.

For generics with compile time argument checking turned off none of these checks will be performed at compile time. The runtime system will be used to perform the necessary checks.

3.5.2.1 Compile Time vs. Runtime Argument Checking

Before jumping to the conclusion that it's always best to have compile time argument checking turned on, a few things should be mentioned. First of all, Dynace has ample facilities to perform argument validation at runtime. Second, runtime-only argument checking is much more accurate and safe than compile-only time argument checking. Third, enabling compile time argument checking on a generic causes inconvenient restrictions to be placed on that generic as follows.

All methods associated with a generic which has compile time argument checking turned on must have the exact same argument signature. On the other hand, a generic with compile time argument checking turned off may be associated with methods with arbitrary argument signatures. This adds a great deal of flexibility and convenience to the system at no loss of safety except that you'll not discover a problem until runtime instead of compile time.

The problem with compile time argument checking shows itself most with the `gNew` generic. Instead of being able to use `gNew` for the creation of instances of all classes, we have to use things like `gNewWithInt` and `gNewWithStr` to differentiate the various argument signatures. Not only is this ugly, but it also causes the programmer to constantly ask, "now, which `gNew` do I use for this class?"

The next sub-section details an alternative approach.

3.5.2.2 Sharing Methods

In order to accommodate programmers with different demands Dynace has the ability to simultaneously support programmers who prefer compile time argument checking turned on all the time and those who prefer more flexibility and better aesthetics at the cost of not detecting argument errors until runtime. Dynace has the ability to associate a single method with both compile time argument checking generics and those without compile time argument checking within a single code base. The programmer may enable this feature when building the system via the `-X dpp` option (see the appropriate section).

As shipped, Dynace includes several generics which are used to provide non-compile time argument checking versions of conveniently overloaded generic functions, including `vNew`, `vFind`, `vRemove`, `vFormat`, and `vAdd`. The entire system must be compiled with the `dpp -X` option in order to enable them, however.

3.5.3 Runtime Argument Validation

Whenever a generic function is evoked in Dynace, the system checks the validity of the first argument. It makes sure that the first argument is, in fact, a valid Dynace object. If it is not, Dynace will report that fact and abort the program. Otherwise, the continuation of the program could reek havoc on the entire system.

This *object checking* comes at a runtime cost, however. This cost is incurred at every evocation of any generic function. While the cost is relatively minimal, programs that are well debugged may turn the checking off to maximize performance. The following code fragment illustrates your control over object checking:

```
gObjectChecking(Dynace, 0);  /* This line turns off object checking */
gObjectChecking(Dynace, 1);  /* This line turns on  object checking */
```

The default is for object checking to be turned on.

3.5.3.1 Additional Argument Validation

If runtime argument checking is turned on (the default) Dynace automatically checks the validity and type of the first argument to all generics. It then dispatches to the appropriate method. Since different methods associated with a particular generic may contain a different number and types of arguments, it is the responsibility of the individual methods to validate their remaining arguments.

Dynace provides several mechanisms to accomplish and simplify this task. The most convenient mechanism is the use of several macros called `ChkArg`, `ChkArgTyp`, `ChkArgNul`, and `ChkArgTypNul`. These macros provide a very simple way of validating an argument as being a valid Dynace object and, optionally, of being of a particular type. If the argument is invalid, an appropriate and meaningful error message will appear. These macros are also used internally by Dynace where appropriate and may be controlled by the object checking mode described in the previous section.

Other mechanisms that may be used are the `IsObj` function, the `gIsKindOf` generic, the `ClassOf` macro, the `gInvalidObject` generic, the `gInvalidType` generic, and the `gAbort` generic, as well as others.

The following code fragment illustrates the use of the `ChkArgTyp` macro:

```
/*  Validate that arg3 (the 3rd argument) is an instance of String
    (or one of its sub-types)  */
ChkArgTyp(arg3, 3, String);

/*  Validate that arg4 (the 4th argument) is any Dynace object  */
ChkArg(arg4, 4);
```

Note that the `ChkArg` set of macros may only be used as the first executable statements in a method.

3.5.4 Methods With A Variable Number Of Arguments

Due to the way Dynace deals with the various method / generic interface strategies (described under `dpp` subsection "Preprocessing Strategies"), it was necessary to construct a standard way of dealing with methods that implement variable arguments. The use of these conventions or standards assures the portability of method definitions across all support strategies without loss of functionality or convenience.

The conventions used to deal with variable argument lists are fully described in the section on the `MAKE_REST` and `GetArg` Dynace macros.

3.5.5 Tracing Facility

Dynace includes a powerful and flexible tracing facility to assist in the debugging of applications. The facility traces generic calls of arbitrary combinations of classes, generics, and methods. Each traced generic prints out the generic name, the type of the first argument to the generic, the name of the method found and the class of the method found.

The tracing facility has three global modes controlled by `Trace::Dynace`. It can be turned off (the default) to disable all generic tracing. It can be turned on in order to trace generic calls associated with classes, generics, or methods that have their tracing options activated (via `Trace::Behavior`, `Trace::GenericFunction`, and `Trace::Method`) and none deactivated. The third mode is to turn tracing on all. This traces all generic calls that do not have their tracing specifically deactivated or are not associated with classes or methods that are specifically turned off.

Note that when referring to classes in the context of the tracing facility, it refers to both the class of the first argument to the generic and the class associated with the method found.

The tracing facility can be turned on and modified at any number of arbitrary points in the application.

Trace output is sent to the stream `traceStream`, which defaults to `stdout`.

See `Trace::Dynace`, `Trace::Behavior`, `Trace::GenericFunction`, `Trace::Method`, `TracePrint::Dynace`, and `traceStream` for more information.

3.5.6 Memory Management and Object Disposal

As stated earlier, all objects in Dynace are allocated from the heap. In fact the `object` declaration actually declares a variable to be a pointer to this allocated memory. Therefore, when an object is no longer needed, it must be returned to the heap. If this is not done,

you would eventually run out of heap space. Dynace has two methods of accomplishing this task: disposal generics and the garbage collector.

Any class whose instances must do something more when being disposed of than just freeing themselves must define an instance disposal method. The types of additional things this disposal method would normally do are free any additionally allocated storage that is local to the object being freed or perform any processing that needs to be performed at object disposal time.

The disposal methods are associated with the `gDispose` and `gDeepDispose` generic functions.

The `gDispose` generic is typically used to dispose of a single object and any associated allocated space it may reference. The system default `gDispose` will handle getting rid of the single object, so the only time a specific method is needed is when the object allocates its own memory from the heap or other processing needs to occur at disposal time. A programmer-defined `gDispose` method would typically free any allocated space associated with the object and then call its super `gDispose` method.

The `gDeepDispose` generic, in addition to performing the same functions as the `gDispose` generic, is typically used to dispose of a whole group of related objects at the same time. This method will typically recursively call the `gDeepDispose` generic on all objects being referenced within the object being disposed of. The system default `gDeepDispose` method performs the same function as the system default `gDispose` method.

3.5.7 Speeding Up IsObj

`IsObj` is a function that validates a Dynace object pointer at runtime. This function is small and fast but gets called quite a lot and often accounts for much of the overhead associated with Dynace. This overhead is significantly exacerbated in the Windows environment where the heap is non-contiguous.

Dynace's allocation scheme is designed to be memory efficient. However, this scheme is at odds, under Windows, with the speed of the system. Dynace has facilities to eliminate this overhead under Windows. If you build and run a Dynace-based application, you can call `gMAxMemUsed(Dynace)` after running through a representative run of the application and before exiting to determine the amount of Dynace memory the application used. You can then use the `Dynace_GetInitialPageSize()` function to cause Dynace to allocate that amount as a single chunk of memory, thus significantly speeding up the application. `Dynace_GetPageSize()` can also be used to set reasonable values. See the relevant documentation for those functions in this manual.

Note that this issue and its corresponding solution are only relevant in a Windows environment. Unix / Linux don't suffer from these problems.

3.5.8 Garbage Collection

The way the garbage collector works is that it keeps track of your maximum real memory requirements after a garbage collection. Whenever your current memory usage exceeds a user-defined number of bytes over the maximum real memory requirements, a garbage collection is evoked. Thus, the automatic aspects of the garbage collection process may

be controlled by specifying this user-defined buffer area. If the buffer area is negative, no garbage collection process will take place. You will just have to explicitly dispose of any unneeded objects. The buffer area may be defined as follows:

```
gSetMemoryBufferArea(Dynace, 40000L);
```

Notice that the argument must be a long type. If passed a negative long value, garbage collection will be turned off. The default setting is `-1L`.

The smaller the buffer area is, the more often the garbage collector will have to run, and the slower the system will operate. However, the memory requirements of the system will be as small as possible. The larger the buffer area, the less frequently the garbage collector will have to run, and therefore, the faster the system will run. However, if the buffer area is too large, the system will experience delays when the very infrequently called garbage collector has to collect a large volume of garbage. The best setting should be determined experimentally. Just remember that the memory requirements of the system will be equal to the real memory requirements of the system plus the buffer area. A starting value of about `40000L` is recommended.

The garbage collector may be manually evoked with the following line:

```
gGC(Dynace);
```

Any class whose instances allocate memory from the heap must define an instance method that is associated with the `gGCDispose` generic. The garbage collector automatically calls this method when it frees an object. It must free the allocated space and call the `gGCDispose` super method just like a `gDispose` method. The only difference between the `gDispose` and `gGCDispose` methods is that the `gGCDispose` method should free allocated memory but must *never* dispose of any other Dynace object, although it may perform any other necessary disposal-type processing. Typically, the `gDispose` method is adequate as a `gGCDispose` method and is used as such.

The garbage collector in Dynace uses an algorithm that does not require a lot of stack space even while collecting highly recursive data structures. The stack requirements are constant regardless of the objects being collected.

Dynace offers an alternative garbage collection scheme that has important trade-offs. See the sub-section on the Boehm Garbage Collector.

3.5.9 Global or Static Variables

The Dynace garbage collector only marks objects that are accessible, either directly or indirectly, via automatic variables (those on the stack). Therefore, if you create an object and assign it to a global or static variable (a variable not on the stack) only, the garbage collector will not mark those objects. If those objects are not marked, they will be freed. This is probably not what you want.

Note that under Win32, Dynace has special code and can automatically detect and mark unregistered globals or statics. Therefore manual registration of these memory locations is not needed under Win32.

Dynace has the ability to register a memory range with the garbage collector so that any valid objects referenced in this range will be marked. This will prevent the freeing of these needed objects. The following example illustrates the registration of a single global variable, `GV`:

```
RegisterVariable(GV);
```

This will cause the garbage collector to mark all objects that `GV` refers to, either directly or indirectly. It is OK for `GV` to not refer to a valid object.

Dynace also includes a method of marking a range of addresses. The following example illustrates its usage:

```
gRegisterMemory(Dynace, begRange, size);
```

The `RegisterVariable` macro actually calls `gRegisterMemory`.

3.5.10 Boehm Garbage Collector

In addition to the garbage collector (GC) that is part of Dynace, Dynace supports another GC that is freely available and widely used throughout the industry. This GC is not owned or supported by Algorithms Corporation or Blake McBride. It is called the Boehm GC and is copyrighted by Hans-J. Boehm, Alan J. Demers and Xerox Corporation.

The Dynace GC is only capable of collecting Dynace objects and requires a bit of programmer effort when dealing with non-Dynace objects such as application malloc'ed memory (which the Dynace GC will not collect) or global variables. The Boehm collector has been integrated with Dynace such that it will collect Dynace objects as well as application malloc'ed memory. In addition, the Boehm GC requires somewhat less programmer effort in order to work correctly. The main advantage of the Dynace GC is that it is more portable and comes with vanilla Dynace. The Boehm GC will not work in 16-bit environments such as DOS or 16-bit Windows.

In order to use the Boehm GC, you must re-compile the entire Dynace system with the macro `BOEHM_GC` defined. In addition, the application must also be compiled with that macro defined and linked with the Boehm GC library. Both collectors cannot be used at the same time; toy must use one or the other. When using the Boehm GC, there is no need to enable it, set the buffer area, or register memory regions. It's on all the time and functions automatically. See the Boehm GC documentation.

If your Dynace package didn't come with the Boehm GC source code you may obtain it via one of the following internet or e-mail sources:

```
http://www.hpl.hp.com/personal/Hans_Boehm/gc/
http://reality.sgi.com/employees/boehm_mti
ftp://parcftp.xerox.com/pub/gc
Hans_Boehm@hp.com
boehm@acm.org
```

3.5.11 Native Thread Support

Dynace has been designed to support a truly multi-threading environment. This is not the simulated threads supplied with Dynace but the true threads provided by the underlying

system. Dynace utilizes critical sections or semaphores to ensure safe operation in a true multi-threading environment.

To enable this facility, Dynace must be built from scratch with `NATIVE_THREADS=1` set on the makefile line. The same is true for all applications that intend to use native threads. `Dpp` generates class files with the structures defined and used by Dynace already protected.

Additionally, the interface to the underlying system's critical section or semaphore facility will have to be provided. With knowledge of this area, it is simple to provide. Dynace comes shipped with native thread support for Win32 only at this time. See `include/dynl.h` and the `*.c` files generated by `dpp`.

3.5.12 Method Cache

As stated earlier, the Dynace system caches method dispatching. This caching speeds up the system tremendously. The larger the cache the better. The default cache size is rather small and may be user controlled. The following line may be used to set the cache size:

```
gResizeMethodCache(Dynace, 51, 101);
```

This example sets the cache size to 51 classes and 101 generics. These numbers do not have to reflect the actual number of classes and generics you have. They are just used to set up the cache. Regardless of the numbers used, the system can handle any number of classes and generic functions.

If this function is used, it should be used soon after the initialization of the Dynace system.

3.5.13 Memory Compaction

Dynace includes a memory allocator that is capable of performing memory compaction. This is used by the `String` class (as well as possibly other classes) to reduce or eliminate memory fragmentation. Since the compactor (`MA_compact()`) may relocate memory, it is never called automatically by Dynace. After a call to `MA_compact` all pointers previously returned by Dynace generics such as `gStringValue` may be invalid. All object pointers, however, will always be valid.

The compactor would typically be called manually after a number of operations that would typically fragment memory.

3.5.14 Avoiding Runtime Method Lookup Costs

There is a small runtime overhead associated with the runtime binding mechanism of Dynace (or any system that has runtime binding). This overhead is kept to a minimum by the Dynace method cache and other techniques employed by the Dynace system. It is possible, however, to locally cache a method to reduce the future runtime overhead to zero. This technique is normally only used in tight loops where there is no chance of using the wrong method. Overuse of this technique is strongly discouraged.

See the macros `cmcPointer`, `cmiPointer`, `imcPointer` and `imiPointer` for further details and examples. There is also an example program that demonstrates the technique.

3.5.15 Generic Functions As First Class C Objects

In C, variables are treated as first class objects. That is they may be defined, initialized, assigned to (value change) and localized in any arbitrary block. C functions, on the other hand, are sort of second class (or static) objects. They may be defined but they can't be changed (at runtime) or localized in an arbitrary block.

In Dynace generic functions are implemented in such a way that they are first class objects, just like variables. That is, they may be assigned to, passed, and localized just like variables.

See the relevant example for more details.

3.5.16 Static Binding

It is possible to perform static binding with the Dynace system. Doing so, however, is highly discouraged because it eliminates much of the flexibility and encapsulation properties of Dynace. The only reason someone might want static binding is for efficiency reasons. These needs, however, may be met by local method caching described above. (See the sub-section on Avoiding Runtime Method Lookup Costs.)

In order to perform static binding, you declare your method as a normal C function instead of using the method declaration syntax. You must also declare its first argument to be `object self` and explicitly use the `accessIVs` macro.

From that point on, it will act as other methods and may then be called directly, without going through the generic.

When calling a method directly, it is the programmer's responsibility to validate the validity and type of the first argument.

3.5.17 Allocating Objects From The Stack

All objects in Dynace are normally allocated from the heap. This allows for the maximum flexibility in terms of their life time and scope. However, Dynace does have a limited capability to allocate objects from the stack.

The two advantages of allocating objects from the stack are that the allocation process, and hence the creation of the object, is very fast. The second advantage is that no explicit object disposal is necessary. Stack-based objects simply disappear when the code goes out of scope in a fashion similar to automatic variables. Thus, there is zero overhead associated with the object's disposal.

The two disadvantages associated with stack-based objects are that stack based objects are limited to the stack size and there is no provision for auto execution of an object destructor. In addition, object creation is a bit clumsy.

Dynace provides two ways to dispose of a heap-based object. First, it can be explicitly disposed of via the `gDispose` or similar generic methods. Second, it can be automatically disposed of via the garbage collector. Both mechanisms allow the programmer to specify additional processing that must be performed when the object is disposed of (such as freeing memory or closing files). Unfortunately, stack-based objects do not have this ability unless

they are explicitly disposed of, and explicitly disposing of objects mostly negates the whole value of having stack-based objects in the first place.

Explicit disposal of stack-based objects does not free up any stack space until the scope is exited. Therefore, if five objects are created and freed in sequence in the same stack frame, the stack space required will be that of all five objects.

Due to the above facts, stack-based objects are generally discouraged. However, in cases where they are needed and no special disposal is required, they can fit the bill.

See documentation on the `StackAlloc` for further details.

3.6 Notes On Compilers, Rebuilding And Porting Dynace

Dynace has been designed and tested to be as portable as possible. At the time of the writing of this manual Dynace has been compiled and tested under MS–DOS, Windows 3.1, Windows 95, Windows NT, Interactive UNIX, UHC UNIX, Linux and Sun SPARC and the Alpha processor. Compilers used were Microsoft Visual C++ 1.0 and above (16 and 32 bit), Borland C++ 3.1 and above (16 and 32 bit), WATCOM C/C++^32 9.0 and above, Symantec C++ version 7.21 (32 bit), and GNU C 2.4 and above. In addition, several prior versions of each of the above compilers were also tested.

When `dpp` strategy 1 is selected there is one small piece of assembler (the jumpTo code) used by the Dynace system. This code is specific to each compiler/model/OS combination. All jumpTo assembler routines that have been developed have been included with the system. There is no assembler required for the default strategy of 2.

The Dynace threader uses the C library functions setjmp/longjmp. When using these functions it is important not to over-optimize a program (as most compiler documentation attests). The included makefiles were designed to maximize the reliability of the Dynace threader. If the threader is not going to be used, more aggressive optimization may be used.

The generics file must always be compiled with the optimization options (usually maximum) defined in the makefiles.

3.6.0.1 Microsoft Visual C++

The makefiles used for this compiler are all called `makefile.msc`. These builds work in both 32 and 64 bits.

Since the Dynace threader makes use of setjmp/longjmp and the Microsoft Visual C++ documentation recommends not to use most of the optimization features when setjmp is used, it is recommended to use only the optimization options used in the included makefiles. It has been demonstrated that the `-Oe` option causes problems with the threader. If the threader is not going to be used, much heavier optimization may be used.

Maximum optimization with no stack checking must always be used when compiling the generics file.

3.6.1 UNIX / Linux

The makefiles associated with the UNIX/Linux build are all named *makefile*.

Only the GNU C compiler and GNU Make utility have been used.

Because of the linear memory associated with UNIX/Linux, Dynace runs quite well under it.

In addition note that the top-level makefile is set up to correct all the files in the system automatically.

3.6.2 Using A Debugger

Because of the way Dynace dispatches methods it is often difficult to trace through a generic call at the C source level. It has been found that if a breakpoint is placed at the beginning of the actual method, the debugger will stop there when going through the generic. So, for

example, if you want to trace into a call to the `gNew` generic and you know you are dealing with the `ShortInteger` class, you may put a breakpoint on the `ShortInteger_im_gNew` method in the `shortint.c` file and continue. The system will stop at the `ShortInteger_im_gNew` method, and you should be able to continue as normal. If you are not sure what class the object is, just put a breakpoint on all the possible methods the generic will dispatch to.

There is not a problem when debugging a program at the machine instruction level. Therefore, it is possible to step at the C level up to a generic call, switch to the machine instruction level, step through the generic call until the method is reached, and then switch back to the C source level.

See the section on the Dynace tracing facility for help debugging a program.

3.6.3 Porting To Other Compilers, Memory Models or OSs

There are several issues associated with porting Dynace to a new environment. These issues are covered in the following subsections.

Dynace includes a few short programs to assist in the porting process. They are located under `\DYNACE\KERNEL\PORT`.

3.6.3.1 JumpTo Assembler Code

The Dynace preprocessor (`dpp`) is capable of supporting various method / generic interface strategies which are described in the section documenting `dpp`. While the default strategy (2) does not require any assembler code, the more efficient strategy (1) does. Although Dynace comes with the necessary assembler code for many platforms, this section outlines the steps necessary to create the assembler code for a new environment.

Basically, the assembler must take a pointer to a function as an argument, pop two stack frames and jump (not jsr) to the function passed.

The best way to create this code is to compile jumpto.c (provided with Dynace) into assembler, modify it, and then assemble it. Compiling generics.c into assembler for reference is also helpful. Running the code through a machine instruction debugger is a must.

Basically, what you need to do is write down the state of the stack pointer and registers upon entry of the generic function. Next, allow the generic function to calculate the method to execute and call the jumpTo code. Finally, (and this is the part that needs to be modified) the jumpTo code should call the method such that upon entry of the method the stack pointer and registers look as though they did when the generic function was entered.

For someone who knows assembler this process should take about an hour.

3.6.3.2 GC & CPU Registers

If objects can exist in CPU registers, then the Dynace garbage collector must have access to them in order to mark them so as not to incorrectly collect them. Dynace has two ways of dealing with this.

The first way is to declare objects to be volatile. This causes the compiler not to keep any objects in registers. While this is a safe and portable solution, it also slows the system

down a bit. It is, however, required by 16-bit compilers because they tend to split far pointers and store them on the stack in non-contiguous locations.

The second possibility is to explicitly mark objects which are referenced via registers. This is the best solution but requires a small bit of inline assembler. This code would be located in `kernel.c` under the `GC` function.

3.6.3.3 Memory Alignment

Memory alignment determines where in memory objects can be stored. By selecting efficient memory alignments, you can maximize the code and garbage collector efficiency. Typically, alignment is on 4-byte boundaries for 32-bit systems and 2-byte on 16-bit systems.

Dynace includes support for 2 and 4-byte alignments via the `ALIGN2` and `ALIGN4` macros located in `kernel.c`.

3.6.3.4 Linear vs. Segmented Memory

As might be expected, Dynace runs much more efficiently in 32-bit, linear memory mode. Under linear mode Dynace can more quickly determine object validity, a function that is often performed.

Dynace supports both modes via the `SEGMENTED_MEMORY` macro contained within `kernel.c`.

3.6.3.5 Thread Timer

The Dynace multi-threader requires some mechanism whereby the OS is able to execute an application-defined function on a regular basis. The frequency would normally be between 50 and 100 times or more a second.

3.6.3.6 Setjmp/longjmp Functionality

Since the multi-threader makes heavy use of the setjmp/longjmp system functions, it is important that those functions save and restore the registers in addition to the stack frame. This will enable maximum reliability of local variables associated with threads.

4 Kernel Reference

The Dynace kernel consists of the bare minimum of classes, methods, and generic functions needed to implement the object oriented concepts discussed herein. This chapter will discuss the structure and various facilities available in the Dynace kernel.

Note that the first argument to all class methods is always the associated class object and the first argument to all instance methods is always an instance of the associated class. Therefore, documentation for the first argument of generics is not always given – it's redundant.

A convention used throughout the Dynace system is for method names to start with a capital letter and for the associated generic to have the same name with a lower case "g" placed at the beginning. Therefore, even though this manual documents the methods, the programmer would access the methods through their associated generics (add the "g").

4.1 Kernel Class Hierarchy

This Dynace kernel contains the following class hierarchy:

```
Object
     Behavior
          Class
          MetaClass
     Method
     GenericFunction
     Dynace
```

4.2 Object Class

The `Object` Class is the root of all objects in Dynace. It is the last place searched when looking for a given method. As such, the `Object` Class defines the default behavior for all objects in Dynace.

4.2.1 Object Class Methods

There are no class methods defined for the `Object` class.

4.2.2 Object Instance Methods

`Object`'s instance methods define the default behavior for all objects in Dynace.

`BasicSize::Object` [BasicSize]

```
sz = gBasicSize(obj);

object  obj;
int     sz;
```

This method is used to obtain the entire size of an object including all header information and instance variables. The number returned does not include space allocated and pointed to by the object.

Example:

```
object  x;
int     sz;

x = gNewWithInt(ShortInteger, 77);
sz = gBasicSize(x);    /*  sz = 6  */
```

See also: `Size::Object`, `InstanceSize::Behavior`

`Compare::Object` [Compare]

```
r = gCompare(i, obj);

object  i;
object  obj;
int     r;
```

This is the default method used by the generic container classes to determine the equality of the values represented by `i` and `obj`. `r` is -1 if the value represented by `i` is less than the value represented by `obj`, 1 if the value of `i` is greater than `obj`, and 0 if they are equal.

This method only compares the pointer values of the two objects and is, therefore, usually overridden in other classes.

See also: `Hash::Object`

`Copy::Object` [Copy]

```
new = gCopy(obj);

object  obj;
object  new;
```

This method is used to make a copy of an object. A new instance of the same class as `obj` is created. Then the information contained in `obj` is copied to the new object created (`new`). The new object is returned.

Any classes whose instance's allocate memory may need to override this method.

This default method is the same as the `DeepCopy::Object` method and would need to be overridden if they need to be different.

Example:

```
object  x, y;

x = gNewWithInt(ShortInteger, 77);
y = gCopy(x);
```

See also: `DeepCopy::Object`, `EQ`, `Equal::Object`

`DeepCopy::Object` [DeepCopy]

```
new = gDeepCopy(obj);

object  obj;
object  new;
```

This method is used to make a copy of an object. A new instance of the same class as `obj` is created. Then the information contained in `obj` is copied to the new object created (`new`). The new object is returned.

Any classes whose instance's allocate memory may need to override this method.

This default method is the same as the `Copy::Object` method and would need to be overridden if they need to be different.

Example:

```
object  x, y;

x = gNewWithInt(ShortInteger, 77);
y = gDeepCopy(x);
```

See also: `Copy::Object`, `EQ`, `Equal::Object`

`DeepDispose::Object` [DeepDispose]

```
r = gDeepDispose(obj);

object  obj;
object  r;     /*  NULL  */
```

This method is used to free the storage allocated when a new object is created. Any class whose instances don't contain pointers to other allocated space or references to other objects may normally just default to this `DeepDispose`.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  x;

x = gNewWithInt(ShortInteger, 77);
x = gDeepDispose(x);
```

See also: `Dispose::Object`, `GCDispose::Object`

`Dispose::Object` [Dispose]

```
r = gDispose(obj);

object  obj;
object  r;     /*  NULL  */
```

This method is used to free the storage allocated when a new object is created. Any class whose instances don't contain pointers to allocated space may normally just default to this `Dispose` for their disposal when no longer needed. Note that when the garbage collector disposes of objects it calls the `GCDispose` method.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

Example:

```
object  x;

x = gNewWithInt(ShortInteger, 77);
x = gDispose(x);
```

See also: `DeepDispose::Object`, `GCDispose::Object`

`Equal::Object` [Equal]

```
r = gEqual(obj1, obj2);

object  obj1, obj2;
int     r;
```

This method is the default method used to check two objects for equality by comparing their contents. 1 is returned if they are of the same class and contain the same contents and 0 otherwise.

This method should be overridden if the comparison should take into account storage other than the local instance storage. The `String` class is a good example of this.

Example:

```
object  a, c, c, d, e;
int     r;

a = gNewWithInt(ShortInteger, 77);
b = gNewWithInt(ShortInteger, 77);
c = gNewWithInt(ShortInteger, 8);
d = gNewWithInt(Dictionary, 99);
e = gCopy(b);
r = gEqual(a, a);   /*  r == 1  */
r = gEqual(a, b);   /*  r == 1  */
r = gEqual(a, c);   /*  r == 0  */
r = gEqual(a, d);   /*  r == 0  */
r = gEqual(a, e);   /*  r == 1  */
```

See also: `EQ`

`Error::Object` [Error]

```
gError(obj, msg)

object  obj;
char    *msg;  /*  Error message  */
```

This method is used to display an error message (`msg` using `Puts::Stream` and `stderrStream`) and abort the program. `msg` may also be an instance of `String`.

Example:

```
gError(Dynace, "Out of memory.\n");
```

See also: `Puts::Stream`, `stderrStream`

GCDispose::Object [GCDispose]

```
r = gGCDispose(obj);

object  obj;
object  r;     /*  NULL  */
```

This method is used to free the storage allocated when a new object is created. This is a special type of dispose method which should only be called by the garbage collector in Dynace. The `GCDispose` method must be defined for any class whose instances contain allocated space. It should unregister any registered memory associated with the object, free the allocated space and then call the super `GCDispose`. It must not free any objects which are referenced, either directly or indirectly, by the object being disposed of.

This default method simply disposes of the individual object.

See also: `Dispose::Object, DeepDispose::Object`

Hash::Object [Hash]

```
val = gHash(i);

object  i;
int     val;
```

This is the default method used by the generic container classes to obtain hash values for the object. `val` is a hash value between 0 and a large integer value.

This default method only hashes on the pointer value of the object, therefore, it is normally overridden in other classes.

See also: `Compare::Object`

Init::Object [Init]

```
r = gInit(obj);

object  obj;
object  r;
```

This method is used as the default object initialization method for new objects when used in conjunction with `Alloc::Behavior`. The default behavior is to do nothing, therefore, this method does nothing but return its argument. Normally, classes would redefine this method to perform some class specific useful initialization.

Example:

```
object  x;

x = gInit(gAlloc(MyClass));
```

See also: `Alloc::Behavior`, `New::Behavior`

`IsKindOf::Object` [IsKindOf]

```
r = gIsKindOf(obj, cls);

object  obj;
object  cls;
int     r;
```

This method is used to determine if class `cls` is the class of or a superclass of `obj`. If so a 1 is returned and 0 otherwise.

Example:

```
object  a, b;
int     c;

a = gNewWithInt(ShortInteger, 7);
b = gNewWithInt(Dictionary, 101);
c = gIsKindOf(a, ShortInteger);  /* c == 1  */
c = gIsKindOf(a, LongInteger);   /* c == 0  */
c = gIsKindOf(b, Dictionary);    /* c == 1  */
c = gIsKindOf(b, Set);           /* c == 1  */
c = gIsKindOf(b, LinkList);      /* c == 0  */
```

See also: `IsKindOf::Behavior`, `IsInstanceOf`, `ClassOf`

`Print::Object` [Print]

```
ret = gPrint(obj, stm);

object  obj;
object  stm;
object  ret;
```

This method is used to print out a representation of an object and its value to stream `stm`. It evokes `StringRep` on `obj` in order to obtain a printable representation of the object.

The returned value is just `obj`.

Example:

```
object  x;

x = gNewWithInt(ShortInteger, 77);
gPrint(x, stdoutStream);
/* this would print something like:
      ShortInteger:<0x0034:4721> [ 77 ] */
```

See also: `PrintValue::Object`, `StringRepValue::Object`, `stdoutStream`

`PrintValue::Object` [PrintValue]

```
ret = gPrintValue(obj, stm);

object  obj;
object  stm;
object  ret;
```

This method is used to print out a representation of an object's value to stream `stm`. It evokes `StringRepValue` on `obj` in order to obtain a printable representation of the object.

The returned value is just `obj`.

Example:

```
object  x;

x = gNewWithInt(ShortInteger, 77);
gPrintValue(x, stdoutStream);
/* this would print something like:  77  */
```

See also: `Print::Object`, `StringRepValue::Object`, `stdoutStream`

`ShouldNotImplement::Object` [ShouldNotImplement]

```
gShouldNotImplement(obj, meth)

object  obj;
char    *meth;
```

This method is used to display an error message indicating that the class of `obj` (or any of its subclasses) should not implement method `meth`. `meth` is the name of the method.

This method may be useful to help catch class misuse by users of a class. It issues a helpful message when attempting to use a class inappropriately.

The message is output via `Error::Object` which also aborts the program.

See also: `Error::Object`

`Size::Object` [Size]

```
sz = gSize(obj);

object  obj;
int     sz;
```

This is the default method used to obtain the size of an object. It returns the total number of bytes use by the objects instance variables. This method is often overridden.

Example:

```
object  x;
int     sz;

x = gNewWithInt(ShortInteger, 77);
sz = gSize(x);    /*  sz = 2  */
```

See also: `BasicSize::Object`

`StringRep::Object` [StringRep]

```
s = gStringRep(i);

object  i;
object  s;
```

This method is used to provide a default format for the display of instance objects. It creates an instance of the `String` class which represents the class of the object, its address and value. The value is obtained by calling the object's `StringRepValue` method. This is often used to print or display a representation of an object. It is also used by by `Print::Object` (a method useful during the debugging phase of a project) in order to directly print an object to a stream.

See also: `Print::Object`, `PrintValue::Object`, `StringRepValue::Object`

StringRepValue::Object [StringRepValue]

```
s = gStringRepValue(i);

object  i;
object  s;
```

This method is used to provide a default mechanism for creating an instance of the `String` class for the display of instance object's values. It displays the class of the object, and its address. This method is called by the `StringRep` method to obtain the value of the object and is normally overridden by particular classes.

It is also used by `PrintValue::Object` and indirectly by `Print::Object` (two methods useful during the debugging phase of a project) in order to directly print an object's value.

See also: `PrintValue::Object`, `Print::Object`

SubclassResponsibility::Object [SubclassResponsibility]

```
gSubclassResponsibility(obj, meth)

object  obj;
char    *meth;
```

This method is used to display an error message indicating that method `meth` should be implemented by a subclass of the class of `obj`. `meth` is the name of the method.

This method may be useful when grouping common interfaces in an abstract class to insure that its subclasses implement the correct methods.

The message is output via `Error::Object` which also aborts the program.

See also: `Error::Object`

4.3 Behavior Class

The `Behavior` Class is used to define the default behavior common to `Class` and `MetaClass` type objects.

4.3.1 Behavior Class Methods

There are no class methods defined for the `Behavior` class.

4.3.2 Behavior Instance Methods

`Behavior`'s instance methods define the default behavior common to all `Class` and `MetaClass` type objects.

`Alloc::Behavior` [Alloc]

```
obj = gAlloc(cls);

object  cls;
object  obj;
```

This method creates instances of class type objects. The returned value is an instance of class `cls`. All instance variables in the `obj` object are initialized to NULL or 0.

This method works exactly like `New::Behavior` except that it is not normally redefined by subclasses. `Alloc` is strictly used to allocate new instances without also initializing them. `Init::Object` would be defined and called to initialize the object. Conceptually, `New::Behavior` should be equivalent to calling `Alloc` and `Init`.

Example:

```
object  x;

x = gInit(gAlloc(MyClass));
```

See also: `Init::Object`, `New::Behavior`, `StackAlloc` `macro`

`DoesNotImplement::Behavior` [DoesNotImplement]

```
gDoesNotImplement(cls, gen);

object  cls;
object  gen;
```

This method is used to issue an error message when there is an attempt to execute a generic function (`gen`) for class `cls`. The message prints to standard error (`stderr`) and exits the program.

Example:

```
gDoesNotImplement(MyClass, Generic(MyGeneric));
```

See also: `RespondesTo, InvalidObject::Method`

`DontCollect::Behavior` [DontCollect]

```
obj = gDontCollect(cls);

object  cls;
object  obj;
```

This method is used to turn garbage collection off for a single class of objects. It doesn't effect any of its superclasses or subclasses and has no effect of garbage collection is not being used. It returns `cls`.

Example:

```
gDontCollect(ShortInteger);
```

`FindMethod::Behavior` [FindMethod]

```
meth = gFindMethod(cls, gen, lev)

object  cls;    /* class                */
object  gen;    /* the generic object   */
int     lev;    /* search level         */
object  (*meth)();  /*  method          */
```

This method may be used to find the correct method associated with a particular generic–class combination. `gen` is the generic function object (not a pointer to the generic function). `cls` is the class where the search should start and `lev` tells whether this is a regular or superclass type search. 1 is regular and 2 indicates a superclass search.

A pointer to the C function which implements the method is returned. If no method is found `NULL` is returned.

There also exists a function named `_FindMethod` which works similar to `gFindMethod` except it issues an error message if no method is found. `_FindMethod` is the one which does all the work to find the correct method associated with a particular generic–class combination when executing generics.

`FindMethod` should not be called by an application. The relevant macros should be used instead.

There is a similar method called `FindMethodObject::Behavior` which returns the method object instead of the C function pointer which implements the method.

See also: `imcPointer, cmcPointer, imiPointer, cmiPointer, iSuper, RespondsTo, cSuper, DoesNotImplement::Behavior, FindMethodObject::Behavior`

`FindMethodObject::Behavior` [FindMethodObject]

```
meth = gFindMethodObject(cls, gen, lev)

object  cls;    /* class                */
object  gen;    /* the generic object   */
int     lev;    /* search level         */
object  meth;   /*  method          */
```

This method may be used to find the correct method associated with a particular generic–class combination. `gen` is the generic function object (not a pointer to the generic function). `cls` is the class where the search should start and `lev` tells whether this is a regular or superclass type search. 1 is regular and 2 indicates a superclass search.

This method return the method object found. If no method is found `NULL` is returned.

There is a similar method called `FindMethod::Behavior` which returns the C function which implements the method instead of the method object.

See also: `FindMethod::Object, Generic`

`InstanceSize::Behavior` [InstanceSize]

```
is = gInstanceSize(cls)

object  cls;    /* class                */
int     is;     /* instance size        */
```

This method is used to obtain the total size a direct instances of class `cls` would be. This includes space used by the instance variables defined in class `cls` as well as all the superclasses of `cls`.

See also: `BasicSize::Object, StackAlloc::Behavior`

`IsKindOf::Behavior` [IsKindOf]

```
r = gIsKindOf(cls1, cls2);

object  cls1, cls2;
int     r;
```

This method is used to determine if class `cls2` is a superclass of `cls1`. If so a 1 is returned and 0 otherwise. A 1 will also be returned if `cls1` is the same class as `cls2`.

Example:

```
int     c;

c = gIsKindOf(ShortInteger, Object);      /* c == 1  */
c = gIsKindOf(ShortInteger, LongInteger); /* c == 0  */
c = gIsKindOf(Dictionary, Set);           /* c == 1  */
c = gIsKindOf(Dictionary, LinkList);      /* c == 0  */
```

See also: `IsKindOf::Object`, `IsInstanceOf`, `ClassOf`

`MarkingMethod::Behavior` [MarkingMethod]

```
obj = gMarkingMethod(cls, fun);

object  cls;
object  (*fun)(cls);
object  obj;
```

This method is used to specify an additional marking method to be used by the garbage collector. It should only be needed in extremely special cases. After the garbage collector marks all used objects it checks for any classes which have a special marking method. The special marking method would normally mark additional objects which are accessed in a very unusual way.

See also: `MarkRange::Dynace`, `MarkObject::Dynace`, `RegisterMemory::Dynace`

`Name::Behavior` [Name]

```
name = gName(cls);

object  cls;
char    *name;
```

This method is used to obtain the name of a given class type object.

Example:

```
char    *name;

name = gName(ShortInteger);
/*  name now points to:   "ShortInteger"  */
```

`New::Behavior` [New]

```
obj = gNew(cls);
     or
obj = vNew(cls);

object  cls;
object  obj;
```

This is the method which creates instances of class type objects. The returned value is an instance of class `cls`. All `New` methods call this method. All instance variables in the `obj` object are initialized to NULL or 0. This method is often redefined in particular classes so that instance variables may be set to appropriate initial values.

Both `gNew` and `vNew` are associated with this method.

Example:

```
object  x;

x = gNew(LinkList);
```

See also: `Alloc::Behavior`, `Init::Object`, `StackAlloc` `macro`

`StackAlloc::Behavior` [StackAlloc]

```
obj = gStackAlloc(cls, ptr);

object  cls;
void    *ptr;
object  obj;
```

This method is used to turn arbitrary storage pointed to by `ptr` into an instance of `cls`. The storage is assumed to be large enough (the value returned by `InstanceSize::Behavior`). The object returned will function as a normal object except that when it is disposed (via `gDispose`, etc.) no actual freeing of the object will occur. However, user defined disposal code will be utilized.

This facility is currently only used to allocate objects from the stack. The cover macro, `StackAlloc`, should be used instead of this method.

See also: `StackAlloc` `macro`, `InstanceSize::Behavior`

`StringRep::Behavior` [StringRep]

```
s = gStringRep(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the object `i` and its value. The `String` contains the name of the class, and its address. It then calls the `StringRepValue` method to obtain its contents. This is often used to print or display a representation of a class object. It is also used by by `Print::Object` (a method useful during the debugging phase of a project) in order to directly print an object to a stream.

See also: `Print::Object`, `PrintValue::Object`, `StringRepValue::Link`

`StringRepValue::Behavior` [StringRepValue]

```
s = gStringRepValue(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the value associated with `i`. This is often used to print or display the value. It is also used by `PrintValue::Object` and indirectly by `Print::Object` (two methods useful during the debugging phase of a project) in order to directly print an object's value.

See also: `PrintValue::Object`, `Print::Object`

`SubClasses::Behavior` [SubClasses]

```
lst = gSubClasses(cls);

object  cls;
object  lst;
```

This method is used to obtain a list of all the direct subclasses of class `cls`. The list returned is an instance of `LinkObject`.

Use `gDispose1()` on the list when no longer needed.

Example:

```
object  lst;

lst = gSubClasses(ShortInteger);
```

See also: `SuperClasses::Behavior`

`SuperClasses::Behavior` [SuperClasses]

```
lst = gSuperClasses(cls);

object  cls;
object  lst;
```

This method is used to obtain a list of all the direct superclasses of class `cls`. The list returned is an instance of `LinkObject`.

Use `gDispose1()` on the list when no longer needed.

Example:

```
object  lst;

lst = gSuperClasses(ShortInteger);
```

See also: `SubClasses::Behavior`

`Trace::Behavior` [Trace]

```
pm = gTrace(cls, mode)

object  cls;    /*  class               */
int     mode;   /*  trace mode          */
int     pm;     /*  previous trace mode  */
```

This method is used to set the trace mode associated with a particular class (`cls`). This mode only has an effect when the Dynace tracing facility is activated (via `Trace::Dynace`). The possible settings (defined in `dynl.h`) are as follows:

```
DYNACE_TRACE_DONT_CARE   OK to trace this class if a related
                         method or generic has tracing turned
                         on (default)

DYNACE_TRACE_OFF         Turn all tracing related to this
                         class off

DYNACE_TRACE_ON          Activate tracing for this class
```

The value returned is the previous trace mode in effect. See the section on tracing (under Dynace Customization & Special Techniques) for more information.

Example:

```
gTrace(Set, DYNACE_TRACE_ON); /* turn tracing on for Set */
```

See also: `Trace::Dynace`, `TracePrint::Dynace`, `Trace::GenericFunction`, `Trace::Method`

4.4 Class Class

The `Class` Class is mainly used for the creation of new class objects.

4.4.1 Class Class Methods

The only class methods in the class `Class` are used to create new class objects and find class objects.

`FindClass::Class` [FindClass]

```
cls = gFindClass(Class, cname)

char    *cname; /*  the name of the class  */
object  cls;    /*  The class found        */
```

This method is used to find a class object from its name. The requested class is returned or `NULL` if it doesn't exist.

Example:

```
object  cls;

cls = gFindClass(Class, "Set");
```

`GetAll::Class` [GetAll]

```
cl = gGetAll(Class);

object  cl;     /*  LinkObject of all class objects  */
```

This method is used to get a linked list (of type `LinkObject`) of all the class objects (classes). It is useful for enumerating through all the class objects. Once a class object is obtained, it can be used to obtain information about the class.

Once `cl` is no longer needed it is important that it is disposed with `gDispose` and not `gDeepDispose`. Use of `gDeepDispose` on `cl` will dispose of all the class objects in the system and crash it.

See also: `GetAll::Class`

`NewClass::Class` [NewClass]

```
cls = gNewClass(Class, cname, ivsize, cvsize, superClasses...)

char    *cname; /*  the name of the new class    */
int     ivsize; /*  the size of the instance variables in
                    the new class  */
int     cvsize; /*  the size of the class variables  */
object  superClasses;  /*  an END terminated list of
                          superclasses  */
object  cls;    /*  The new class being created  */
```

This method is used to create new classes. It automatically creates the class object, the metaclass object associated with the new class object and sets up all the necessary internal pointers and data structures.

The `name` parameter is the name associated with the new class. The metaclass created will have the same name with "meta" prepended to the name.

The `ivsize` parameter defines the size of the locally declared instance variables associated with the new class. If there are no local instance variables `0` should be used.

The `cvsize` parameter is used to declare the size of the locally defined class variables associated with the new class. If there are no locally defined class variables `0` should be used, otherwise, the size of the class variable structure should be used.

The remainder of this method's arguments define the superclasses the new class will have. The last argument must be `END`. If no superclasses are defined the single class `Object` will be the default superclass. The order of the superclasses will determine the method lookup order.

Example:

```
object  NewClass;

NewClass = gNewClass(Class, ShortInteger, END);
```

See also: `NewStdClass::Class`

`NewStdClass::Class` [NewStdClass]

```
cls = gNewStdClass(Class, cname, ivsize, mclass, nipib,
                   sCls...)

char    *cname; /*  the name of the new class    */
int     ivsize; /*  the size of the instance variables in
                    the new class  */
object  mclass; /*  the metaclass of the new class  */
int     nipib;  /*  number of instances per instance block  */
object  sCls;   /*  an END terminated list of
                           superclasses  */
object  cls;    /*  The new class being created  */
```

This method is used to create new classes. Unlike `NewClass::Class`, which automatically creates a corresponding metaclass (like Smalltalk – more convenient), `NewStdClass::Class` creates single independent classes with the ability to set the metaclass (like CLOS – more powerful but more cumbersome to use).

The `name` parameter is the name associated with the new class.

The `ivsize` parameter defines the size of the locally declared instance variables associated with the new class. If there are no local instance variables `0` should be used.

The `mclass` parameter is used to set the class of the newly created class. This is the class which defines the behavior and characteristics of the newly created class. Note that any number of classes may share the same metaclass. This adds a great deal of power to the meta-language aspects of the system.

For efficiency reasons Dynace does not allocate memory for objects one at a time. They are allocated in blocks which are measured by the number of instances per block of memory. This factor may be specified by the use of the `nipib` parameter. When creating a class which will have many small instances used it is best to have `nipib` high (~50). However, if the instances are large and there will only be a few of them 1 would be a good choice. The use of 0 will cause the system to estimate an appropriate value based on the instance size.

The remainder of this method's arguments define the superclasses the new class will have. The last argument must be `END`. If no superclasses are defined the single class `Object` will be the default superclass. The order of the superclasses will determine the method lookup order.

See also: `NewClass::Class`

4.4.2 Class Instance Methods

The class Class has no instance methods.

4.5 MetaClass Class

All metaclasses in the system are instances of the class `MetaClass`. Since all metaclasses are instances of the `MetaClass` class, and all class objects are instances of their associated metaclasses, the `MetaClass` class is the heart of the Dynace class system. The `MetaClass` class has no class or instance methods.

4.6 Method Class

The `Method` class gives Dynace the ability to represent methods (static C language functions) as true objects. The principal use of this class is to create new methods.

4.6.1 Method Class Methods

There is only one class method associated with the `Method` class and it is used for the creation of new methods.

`NewMethod::Method` [NewMethod]

```
method = gNewMethod(Method, name, cls, gen, cf, ff)

char    *name;  /*  the name of the new method   */
object  cls;    /*  the class associated with the method  */
object  gen;    /*  the generic associated with the method  */
object  (*cf)();/*  the C function which implements
                                the method  */
object  (*ff)();/*  the C function which implements
                                the method - fixed args  */
object  method; /*  the new method object  */
```

This method is used to create a new method object. The new method object associates a class object, a generic object, and a C function. The macros `iMethodFor, ivMethodFor, cMethodFor, cvMethodFor` provide the normal access to this method. See those macros for a full description.

See also: `iMethodFor, ivMethodFor, cvMethodFor, cMethodFor`

4.6.2 Method Instance Methods

The instances methods associated with this class are used to obtain information about a method object.

`ChangeFunction::Method` [ChangeFunction]

```
ofun = gChangeFunction(m, nfun);

object  m;
object  (*nfun)();
object  (*ofun)();
```

This method is used to change the C function associated with a method object to `nfun`. The old C function pointer is returned.

See also: `FindMethodObject::Behavior, Function::Method`

`Function::Method` [Function]

```
fun = gFunction(m);

object  m;
object  (*fun)();
```

This method is used to obtain the C function associated with a method object.

See also: `FindMethodObject::Behavior`,
`ChangeFunction::Method`

`Name::Method` [Name]

```
name = gName(m);

object  m;
char    *name;
```

This method is used to obtain the name of a given method type object.

See also: `FindMethodObject::Behavior`

`Trace::Method` [Trace]

```
pm = gTrace(mth, mode)

object  mth;    /*  method object        */
int     mode;   /*  trace mode           */
int     pm;     /*  previous trace mode  */
```

This method is used to set the trace mode associated with a particular method (`mth`). This mode only has an effect when the Dynace tracing facility is activated (via `Trace::Dynace`). The possible settings (defined in `dynl.h`) are as follows:

`DYNACE_TRACE_DONT_CARE`	`OK to trace this method if a related generic or class has tracing turned on (default)`
`DYNACE_TRACE_OFF`	`Turn all tracing related to this method off`
`DYNACE_TRACE_ON`	`Activate tracing for this method`

The value returned is the previous trace mode in effect. See the section on tracing (under Dynace Customization & Special Techniques) for more information.

Example:

```
object  mth;

mth = gFindMethodObject(Set, Generic(gRemove), 1);
  /* turn tracing on for Remove::Set */
gTrace(mth, DYNACE_TRACE_ON);
```

See also: `Trace::Dynace`, `TracePrint::Dynace`,
`FindMethodObject::Behavior`, `Generic`,
`Trace::Behavior`, `Trace::GenericFunction`

4.7 GenericFunction Class

The `GenericFunction` class gives Dynace the ability to represent generic functions (C language functions) as true objects. The principal use of this class is to create new generic function objects.

4.7.1 GenericFunction Class Methods

The class methods in this class are used to create new generic objects and obtain a list of all generics.

`FindGeneric::GenericFunction` [FindGeneric]

```
gf = gFindGeneric(GenericFunction, name);

char  *name;    /*  name of generic function to find  */
object  gf;     /*  GenericFunction object associated with name  */
```

This method is used to obtain the generic function object named `name`. The generic function object is returned unless it is not found, in that case a `NULL` is returned.

See also: `GetAll::GenericFunction`

`GetAll::GenericFunction` [GetAll]

```
gf = gGetAll(GenericFunction);

object  gf;     /*  LinkObject of all generic function objects  */
```

This method is used to get a linked list (of type `LinkObject`) of all the generic objects. It is useful for enumerating through all the generic objects. Once a generic object is obtained, it can be used to obtain information about the generic.

Once `gf` is no longer needed it is important that it is disposed with `gDispose` and not `gDeepDispose`. Use of `gDeepDispose` on `gf` will dispose of all the generic objects in the system and crash it.

See also: `GetAll::Class`, `FindGeneric::GenericFunction`

`NewGeneric::GenericFunction` [NewGeneric]

```
gf = gNewGeneric(GenericFunction, name, fp);

char    *name;  /*  the name of the new generic function */
object  gf;     /*  the new generic function object  */
void    *fp;    /*  pointer to generic function  */
```

This method is used to create a new generic function object. The generic function object is used to store the relationship between classes, generics and methods. Normally, all generics are created in a single file called "generics.c". All external declarations for the generics are kept in a file called "generics.h". These files are created automatically by the `dpp` program from the class documentation.

The macros `InitGeneric, defGeneric, externGeneric` provide the normal access to this method and the generic function object in general.

See also: `dpp, defGeneric, externGeneric, InitGeneric, GetGenericPtr::GenericFunction`

4.7.2 GenericFunction Instance Methods

`GenericFunction` instance methods define the behavior of all generic functions.

`GetGenericPtr::GenericFunction` [GetGenericPtr]

```
gfp = gGetGenericPtr(gf);

object  gf;     /*  a GenericFunction object  */
void    *gfp;   /*  pointer to the generic function
                    associated with gf  */
```

This method is used to get a pointer to the generic function associated with a generic object. `gfp` must be correctly typecast prior to use.

See also: `Generic, GetAll::GenericFunction`

`InvalidObject::GenericFunction` [InvalidObject]

```
gInvalidObject(gf, argn, arg1);

object  gf;     /*  the generic function object  */
int     argn;   /*  argument number  */
object  arg1;   /*  the first argument to generic  */
```

This method is used to report the case when a generic is passed an argument which is not a valid object. After generation of the error message it calls `Error::Object` to report the error and abort the program. It is normally called automatically by a generic or by the use of one of the `ChkArg` macros.

See also: `ChkArg, InvalidType::GenericFunction, Error::Object`

`InvalidType::GenericFunction` [InvalidType]

```
gInvalidType(gf, argn, arg1, cls, arg);

object  gf;     /*  the generic function object    */
int     argn;   /*  argument number            */
object  arg1;   /*  the first argument to generic  */
object  cls;    /*  type of object expected    */
object  arg;    /*  the argument which is bad  */
```

This method is used to report the case when a generic is passed an argument which is not the expected type. It passes the error message to `Error::Object` which prints the message and aborts the program. It is normally called via one of the `ChkArg` type macros.

See also: `ChkArg`, `InvalidObject::GenericFunction`, `Error::Object`

`Name::GenericFunction` [Name]

```
name = gName(g);

object  g;
char    *name;
```

This method is used to obtain the name of a given generic function type object.

Example:

```
char    *name;

name = gName(Generic(vNew));
/*  name now points to:   "vNew"  */
```

See also: `Generic`

`Trace::GenericFunction` [Trace]

```
pm = gTrace(gf, mode)

object  gf;     /*  generic function object */
int     mode;   /*  trace mode              */
int     pm;     /*  previous trace mode     */
```

This method is used to set the trace mode associated with a particular generic function (`gf`). This mode only has an effect when the Dynace tracing facility is activated (via `Trace::Dynace`). The possible settings (defined in `dynl.h`) are as follows:

`DYNACE_TRACE_DONT_CARE`	`OK to trace this generic if a related method or class has tracing turned on (default)`
`DYNACE_TRACE_OFF`	`Turn all tracing related to this generic off`
`DYNACE_TRACE_ON`	`Activate tracing for this generic`

The value returned is the previous trace mode in effect. See the section on tracing (under Dynace Customization & Special Techniques) for more information.

Example:

```
/* turn tracing on for vNew */
gTrace(Generic(vNew), DYNACE_TRACE_ON);
```

See also: `Generic`, `Trace::Dynace`, `TracePrint::Dynace`, `Trace::Behavior`, `Trace::Method`

4.8 Dynace Class

The `Dynace` Class is used to affect various parameters which control the internal operation of the Dynace kernel. The user may control whether the garbage collector is activated or not and what parameters it is running under. You may also control such things as the size of the method cache. All operations performed through this class are performed by sending messages directly to the Dynace class. There are no instance methods.

4.8.1 Dynace Class Methods

All functionality of the Dynace class is accessed through it's class methods.

`ChangeRegisteredMemory::Dynace` [ChangeRegisteredMemory]

```
rmp = gChangeRegisteredMemory(Dynace, rmp, beg, size)

void    *rmp;   /*  registered memory pointer  */
void    *beg;   /*  pointer to beginning of region  */
long    size;   /*  size of region  */
```

This method is used to change the memory region associated with a memory region pointer. `rmp` must have been a pointer returned by `gRegisterMemory`.

Example:

```
static  object  a, b;
void    *rmp;

rmp = gRegisterMemory(Dynace, &a, (long) sizeof(a));
   .
   .
   .
gChangeRegisteredMemory(Dynace, rmp, &b, (long) sizeof(b));
```

See also: `RegisterMemory::Dynace`, `GC::Dynace`

`CurMemUsed::Dynace` [CurMemUsed]

```
cmu = gCurMemUsed(Dynace)

long    cmu;    /*  current memory used  */
```

This method is used to obtain the current amount of storage being used by Dynace objects. This number increases every time a new object is created, and decreases every time an object is disposed.

Example:

```
long    cmu;

cmu = gCurMemUsed(Dynace);
```

See also: `MaxMemUsed::Dynace`, `GC::Dynace`, `DumpObjects::Dynace`

`DumpMemoryDiff::Dynace` [DumpMemoryDiff]

```
c = gDumpMemoryDiff(Dynace, start, fileName)

object  c;       /*  the Dynace class  */
object start;    /*  value returned from gMarkMemoryBeginning::Dynace  */
char *fileName;  /*  the name of the file to dump the list to   */
```

This method is used in conjunction with gMarkMemoryBeginning to find memory leaks. A list of all remaining objects between the call to gMarkMemoryBeginning and gDumpMemoryDiff.

Typically, one would call gMarkMemoryBeginning, run your normal code including calls to dispose of objects no longer needed, and then call gDumpMemoryDiff to see what objects remain after all of the dispose calls. Those are your leaks.

Note that `start` should be DeepDisposed when it is no longer needed. Also, these two methods should normally be used rather than DumpObjects.

Example:

```
object start = gMarkMemoryBeginning(Dynace);
...
gDumpMemoryDiff(Dynace, start, "MyMemDump.txt");
gDeepDispose(start);
```

See also: `MarkMemoryBeginning::Dynace`

`DumpObjects::Dynace` [DumpObjects]

```
c = gDumpObjects(Dynace, file, mc)

object  c;      /*  the Dynace class  */
char    *file;  /*  file to dump object list on  */
int     mc;     /*  0=no metaclasses, 1=metaclasses too  */
```

This method is used to dump a list of all Dynace objects in a human readable format into the file named by `file`. The `mc` flag is used to determine whether or not metaclasses are included in the dump.

This method is mainly used to find leaks in an application. Typically, a body of code would be run, then an object dump performed, then run again, and finally a second object dump to another file is performed. The two dump files would then be sorted and diff'ed. A list of object leaks would result.

Example:

```
gDumpObjects(Dynace, "objects.lst", 0);
```

See also: `MarkMemoryBeginning::Dynace`, `DumpObjectsString::Dynace`, `MaxMemUsed::Dynace`, `GC::Dynace`

`DumpObjectsDiff::Dynace` [DumpObjectsDiff]

```
sd = gDumpObjectsDiff(Dynace, d1, d2)

object  d1, d2; /*  values from gDumpObjectsString  */
object  sd;     /*  StringDictionary instance  */
```

This method is used to create a string dictionary representing the difference between two calls to `gDumpObjectsString`. The string is the name of the class and the value is an instance of `LongInteger` which represents the number of instances different.

This method is mainly used to find leaks in an application. Typically, a body of code would be run, then an object dump performed, then run again, and finally a second object dump is performed. The two dump string dictionaries would then be compared using `gDumpObjectsDiff`. A list of object leaks would result.

Example:

```
object   d1, d2, diff;

d1 = gDumpObjectsString(Dynace, 0);
       .
       .
       .
d2 = gDumpObjectsString(Dynace, 0);
diff = gDumpObjectsDiff(Dynace, d1, d2);
```

See also: `MarkMemoryBeginning::Dynace`, `DumpObjectsString::Dynace`, `DumpObjects::Dynace`

`DumpObjectsString::Dynace` [DumpObjectsString]

```
sd = gDumpObjectsString(Dynace, mc)

int     mc;     /*  0=no metaclasses, 1=metaclasses too  */
object  sd;     /*  StringDictionary instance  */
```

This method is used to create a string dictionary representing all Dynace objects in the system. The string is the name of the class and the value is an instance of `LongInteger` which represents the number of instances. The `mc` flag is used to determine whether or not metaclasses are included in the list.

This method is mainly used to find leaks in an application. Typically, a body of code would be run, then an object dump performed, then run again, and finally a second object dump is performed. The two dump string dictionaries would then be compared using `gDumpObjectsDiff`. A list of object leaks would result.

See also: `MarkMemoryBeginning::Dynace`, `DumpObjectsDiff::Dynace`, `DumpObjects::Dynace`

`GC::Dynace` [GC]

```
c = gGC(Dynace)

object  c;    /*  the Dynace class  */
```

This method is used to force the Dynace system to perform a garbage collection. All inaccessible objects will be collected and made available for reuse. This method is not normally used because the Dynace system will automatically start the garbage collector when necessary (if turned on via the `SetMemoryBufferArea` method).

Example:

```
gGC(Dynace);
```

See also: `SetMemoryBufferArea::Dynace`, `RegisterMemory::Dynace`, `DumpObjects::Dynace`, `NumbGC::Dynace`

`MarkMemoryBeginning::Dynace` [MarkMemoryBeginning]

```
start = gMarkMemoryBeginning(Dynace)

object start;
```

This method is used in conjunction with gDumpMemoryDiff to find memory leaks. A list of all remaining objects between the call to gMarkMemoryBeginning and gDump-MemoryDiff.

Typically, one would call gMarkMemoryBeginning, run your normal code including calls to dispose of objects no longer needed, and then call gDumpMemoryDiff to see what objects remain after all of the dispose calls. Those are your leaks.

Note that `start` should be DeepDisposed when it is no longer needed. Also, these two methods should normally be used rather than DumpObjects.

Example:

```
object start = gMarkMemoryBeginning(Dynace);
...
gDumpMemoryDiff(Dynace, start, "MyMemDump.txt");
gDeepDispose(start);
```

See also: `DumpMemoryDiff::Dynace`

`MaxAfterGC::Dynace` [MaxAfterGC]

```
mem = gMaxAfterGC(Dynace)

long    mem;    /*  maximum memory used  */
```

This method is used to obtain the maximum memory that was in use immediately after the last garbage collection. If no garbage collection has taken place the returned value will represent the current memory usage. The value of this number is that the amount of storage between the current memory usage and this number is considered potential garbage and used (internally) by Dynace to help determine when an automatic garbage collection should occur.

Example:

```
long    mem;

mem = gMaxAfterGC(Dynace);
```

See also: `MaxMemUsed::Dynace, CurMemUsed::Dynace, GC::Dynace`

`MaxMemUsed::Dynace` [MaxMemUsed]

```
mem = gMaxMemUsed(Dynace)

long    mem;    /*  maximum memory used  */
```

This method is used to obtain the maximum memory that was used by the system during the entire run of the application.

Example:

```
long    mem;

mem = gMaxMemUsed(Dynace);
```

See also: `MaxAfterGC::Dynace, CurMemUsed::Dynace, GC::Dynace, DumpObjects::Dynace`

`NumbGC::Dynace` [NumbGC]

```
ngc = gNumbGC(Dynace)

long    ngc;    /*  maximum memory used  */
```

This method is used to obtain the total number of times the garbage collector ran during the entire run of the application.

Example:

```
long    ngc;

ngc = gNumbGC(Dynace);
```

See also: `MaxAfterGC::Dynace, CurMemUsed::Dynace, GC::Dynace`

`ObjectChecking::Dynace` [ObjectChecking]

```
c = gObjectChecking(Dynace, flag)

int     flag;   /*  1=on,  0=off   */
object  c;      /*  the Dynace class  */
```

Since the first argument to all generics is the Dynace object which is being sent the message, the first argument to all generics must always be a valid Dynace object. Whenever a generic is called Dynace verifies that the first argument is in fact a valid Dynace object. If it is not Dynace will issue an error message and abort the program. This is a fatal error.

There is a runtime overhead associated with this check. It's very similar to the stack checking which some C compilers perform at runtime. Once a program is sufficiently debugged it is possible to turn this runtime checking off in order to achieve maximum performance. Turning the checking off is, however, not recommended. If an untested portion of code which contains a bug is executed and the checking is off very unpredictable results will occur. It's very similar to dereferencing an uninitialized pointer in C.

There are techniques to totally eliminate the runtime overhead associated with object validation and method lookup within Dynace. See the text.

By default Dynace has object checking turned on. The `ObjectChecking` method may be called at arbitrary points in a program to turn the checking on or off. A `flag` value of 1 turns checking on and a value of 0 turns it off.

Example:

```
gObjectChecking(Dynace, 0);
```

See also: `ChkArg`

`RegisterMemory::Dynace` [RegisterMemory]

```
rmp = gRegisterMemory(Dynace, beg, size)

void    *beg;   /*  beginning of registered memory  */
long    size;   /*  size of region      */
void    *rmp;   /*  registered memory pointer  */
```

The Dynace garbage collector uses the C stack in order to determine which objects are still accessible, either directly or indirectly, by the program and need to be saved. All remaining objects will be collected and returned to the free store. This scheme allows Dynace to automatically handle objects which are being referenced, directly or indirectly, by automatic variables and function parameters.

Dynace has no way of knowing about objects which are referenced by global and static variables or objects which are being referenced by memory allocated from the heap. If an objects is only being referenced by one of these methods Dynace will dispose of it the next time the garbage collector is called. If the program later tries to use the object an error will occur. The solution to this problem is to register these memory regions with Dynace.

The `beg` parameter specifies the address of the beginning of the region which needs to be protected. The `size` parameter specifies the size in bytes of the region. `RegisterMemory` returns a pointer which may be used to un-register the region at a later time.

The `RegisterVariable` macro provides a convenient interface to protect global and static variables.

If the Dynace garbage collector is not being used there is no need to register any memory regions.

Example:

```
static  object  a, b;

gRegisterMemory(Dynace, &a, (long) sizeof(a));
RegisterVariable(b);
```

See also: `GC::Dynace`, `RemoveRegisteredMemory::Dynace`, `ChangeRegisteredMemory::Dynace`

`RemoveRegisteredMemory::Dynace` [RemoveRegisteredMemory]

```
gRemoveRegisteredMemory(Dynace, rmp)

void    *rmp;   /*  registered memory pointer  */
```

This method is used to remove a memory region from those registered with the Dynace system. The `rmp` parameter must be a pointer which was obtained from a call to `gRegisterMemory`. Once this method is called the specified region of memory will no longer be protected from the garbage collector.

Example:

```
static  object  a, b;
void    *rmpa, *rmpb;

rmpa = gRegisterMemory(Dynace, &a, (long) sizeof(a));
rmpb = RegisterVariable(b);
   .
   .
   .
gRemoveRegisteredMemory(Dynace, rmpa);
gRemoveRegisteredMemory(Dynace, rmpb);
```

See also: `RegisterMemory::Dynace`, `ChangeRegisteredMemory::Dynace`, `GC::Dynace`

`ResizeMethodCache::Dynace` [ResizeMethodCache]

```
c = gResizeMethodCache(Dynace, nClasses, nGenerics)

int     nClasses;       /*  size of class cache table */
int     nGenerics;      /*  size of generic cache table */
object  c;              /*  the Dynace class  */
```

The first time a particular method is evoked it must be searched for as described elsewhere in this manual. This is a time consuming process. It is, however, necessary for this process to occur at runtime in order to obtain the true benefit of Object Oriented Programming. C++, Objective–C, and Smalltalk all perform this search when using their dynamic binding capabilities.

Once the appropriate method is found it is placed in a fast cache automatically maintained by Dynace. This cache is designed such that it never overflows. It just grows on an as-needed basis. Therefore, once a correct method is found it will never be searched for again. All future messages to that particular method will be found in the cache.

When Dynace is initialized a default cache is established. Under normal circumstances you may never need to be concerned with the cache at all. However, under conditions where there are a very large number of classes or generics it may be beneficial to enlarge the cache table sizes.

If the cache size is to be changed the best place to do it is immediately after the initialization of Dynace. You could, however, change the cache size at any point at the one time cost of some additional CPU time.

The `nClasses` parameter should be set to some number larger than the total number of classes in your program. The `nGenerics` parameter should also be set to some number larger than the number of generics in the program. These numbers may be set lower than the number of classes and/or generics in order to save memory at the relatively low cost of runtime speed. Since the cache is implemented as a hash table choosing numbers that are prime is best. Choosing odd numbers is good too.

There are techniques to totally eliminate the runtime overhead associated with object validation and method lookup within Dynace. See the text.

Example:

```
gResizeMethodCache(Dynace, 101, 203);
```

`SetMemoryBufferArea::Dynace` [SetMemoryBufferArea]

```
c = gSetMemoryBufferArea(Dynace, size)

long    size;   /*  size of buffer area  */
object  c;      /*  the Dynace class  */
```

This method is used to activate and control the automatic operation of the Dynace garbage collector. When Dynace is first started the default memory buffer area is set to -1. This value indicates that Dynace should not automatically start the garbage collector.

If the value of `size` is positive then Dynace will automatically start the garbage collector every time the program creates enough *new* objects to exceed the storage specified by the `size` parameter (in bytes). Dynace will automatically adjust its heap

storage requirements based on the `size` parameter and the storage history of the application program.

The actual value of `size` must be determined experimentally. A typical starting place would be about 40000. The higher the value of `size` the less frequently the garbage collector will be started but the longer it will take to run. The lower the value of `size` the more often the garbage collector will run but also the faster. The garbage collector is very fast and you will probably not notice much of a difference for a wide range of `size`.

The typical storage Dynace requires to support this facility is roughly equal to the maximum storage of all the active objects in the application at any given time plus the value of the `size` parameter. Therefore, making `size` unnecessarily large would be wasteful.

Note that the `size` parameter must be of type `long`.

Example:

```
gSetMemoryBufferArea(Dynace, 40000L);
```

See also: `GC::Dynace, RegisterMemory::Dynace`

`Trace::Dynace` [Trace]

```
pm = gTrace(Dynace, mode)

int     mode;   /*  trace mode           */
int     pm;     /*  previous trace mode  */
```

This class method is used to set the global trace mode of the Dynace system. The possible settings (defined in `dynl.h`) are as follows:

```
DYNACE_TRACE_OFF        Turn all tracing off (default setting)

DYNACE_TRACE_ON         Trace objects which are activated for
                        tracing

DYNACE_TRACE_ALL        Trace all except those explicitly
                        deactivated
```

The value returned is the previous trace mode in effect. See the section on tracing (under Dynace Customization & Special Techniques) for more information.

Example:

```
gTrace(Dynace, DYNACE_TRACE_ON);   /*  turn tracing on  */
```

See also: `TracePrint::Dynace`, `Trace::Behavior`,
`Trace::GenericFunction`, `Trace::Method`

`TracePrint::Dynace` [TracePrint]

```
c = gTracePrint(Dynace, msg)

char    *msg;   /*  trace output message  */
object  c;      /*  the Dynace class      */
```

This class method is used to output trace messages and is called by the Dynace internal tracing functions. This method simply outputs `msg` to `traceStream`. It is documented here in case there is a need to override it.

See the section on tracing (under Dynace Customization & Special Techniques) for more information.

See also: `traceStream::Stream`, `Trace::Dynace`, `Trace::Behavior`,
`Trace::GenericFunction`, `Trace::Method`

4.8.2 Dynace Instance Methods

There are no instance methods associated with the Dynace class.

4.9 Kernel Macros

Dynace has a number of macros to simplify the creation and usage of objects. These macros are all defined in the file `dynl.h`. The following text describe each macro.

`accessIVs` [accessIVs]

```
accessIVs;
```

This macro is used to access the instance variables from within an instance method. It may be used anywhere within the variable declaration section of a normal C function. `accessIVs` actually declares a local variable named `iv` and initializes it to a pointer to the instance variable structure. Once `accessIVs` is called the local variable `iv` may be used to access all the locally defined instance variables in the class being defined in the current file.

`accessIVs` uses the variable named `self` to determine which object to obtain instance variables for. See `ivPtr` to access instance variables for other variables.

If an instance method makes only a single access to its instance variables the use of the `ivsPtr` macro would be better suited.

Note that it is rarely ever necessary to explicitly use the `accessIVs` macro due to the fact the the Dynace preprocessor (`dpp`) automatically adds this code when preprocessing the class definition file.

Example:

```
defclass  {
      char    name[30];
      int     age;
};

imeth void    PrintInstances(object self, FILE *fp)
{
      accessIVs;

      fprintf(fp, "%s is %d years old\n", iv->name, iv->age);
}
```

See also: `ivsPtr`, `ivPtr`, `ivType`, `GetIVs`

`ChkArg` [ChkArg]

```
ChkArg(arg, argn);

object  arg;    /*  the argument to be checked  */
int     argn;   /*  the argument number         */
```

This macro provides a convenient way of validating method arguments. If `arg` is not a valid Dynace object a meaningful error message will be displayed and the program will be aborted. `argn` is used in the error message to tell which argument caused the problem.

The argument checking properties of this macro may be enabled / disabled via the `gObjectChecking` generic. It defaults to ON.

Since Dynace automatically checks the first argument to all generics (if argument checking is on) there is never a need to manually check the first argument. `ChkArg` would only be needed for arguments beyond the first.

`ChkArg` may only be used as the first executable statements in a method (prior to any other generic calls).

Example:

```
imeth object  MyMethod(object self, object arg2, object arg3)
{
        accessIVs;

        ChkArg(arg2, 2);
        ChkArg(arg3, 3);
        .
        .
        .
```

See also: `ObjectChecking::Dynace`, `IsObj`, `IsKindOf::Object`, `ChkArgTyp`, `ChkArgNul`, `ChkArgTypNul`

`ChkArgNul` [ChkArgNul]

```
ChkArgNul(arg, argn);

object  arg;    /*  the argument to be checked  */
int     argn;   /*  the argument number         */
```

This macro provides a convenient way of validating method arguments. Unlike `ChkArg` this will also permit an argument of `NULL`. If `arg` is not a valid Dynace object (or `NULL`) a meaningful error message will be displayed and the program will be aborted. `argn` is used in the error message to tell which argument caused the problem.

The argument checking properties of this macro may be enabled / disabled via the `gObjectChecking` generic. It defaults to ON.

Since Dynace automatically checks the first argument to all generics (if argument checking is on) there is never a need to manually check the first argument. `ChkArgNul` would only be needed for arguments beyond the first.

`ChkArgNul` may only be used as the first executable statements in a method (prior to any other generic calls).

Example:

```
imeth object  MyMethod(object self, object arg2, object arg3)
{
        accessIVs;

        ChkArgNul(arg2, 2);
        ChkArgNul(arg3, 3);
        .
        .
        .
```

See also: `ObjectChecking::Dynace`, `IsObj`, `IsKindOf::Object`, `ChkArgTyp`, `ChkArg`, `ChkArgTypNul`

ChkArgTyp [ChkArgTyp]

```
ChkArgTyp(arg, argn, type);

object  arg;    /*  the argument to be checked  */
int     argn;   /*  the argument number         */
object  type;   /*  the expected class of arg   */
```

This macro provides a convenient way of validating method arguments. If `arg` is not a valid Dynace object a meaningful error message will be displayed and the program will be aborted. `argn` is used in the error message to tell which argument caused the problem.

`type` is used to specify the expected type of the argument. If `arg` is not an instance of class `type` (or one of its subclasses) an error message will be displayed and the program will be aborted. If no argument type checking is required `type` should be passed a `NULL` and no type checking will be performed.

The argument checking properties of this macro may be enabled / disabled via the `gObjectChecking` generic. It defaults to ON.

Since Dynace automatically checks the first argument to all generics (if argument checking is on) there is never a need to manually check the first argument. `ChkArgTyp` would only be needed for arguments beyond the first.

`ChkArgTyp` may only be used as the first executable statements in a method (prior to any other generic calls).

Example:

```
imeth object  MyMethod(object self, object arg2, object arg3)
{
        accessIVs;

        ChkArgTyp(arg2, 2, String);
        ChkArgTyp(arg3, 3, Set);
        .
        .
        .
```

See also: `ObjectChecking::Dynace`, `IsObj`, `IsKindOf::Object`, `ChkArg`, `ChkArgNul`, `ChkArgTypNul`

`ChkArgTypNul` [ChkArgTypNul]

```
ChkArgTypNul(arg, argn, type);

object  arg;    /*  the argument to be checked  */
int     argn;   /*  the argument number         */
object  type;   /*  the expected class of arg   */
```

This macro provides a convenient way of validating method arguments. Unlike `ChkArgTyp` this will also permit an argument of `NULL`. If `arg` is not a valid Dynace object (or `NULL`) a meaningful error message will be displayed and the program will be aborted. `argn` is used in the error message to tell which argument caused the problem.

`type` is used to specify the expected type of the argument. If `arg` is not an instance of class `type` (or one of its subclasses) an error message will be displayed and the program will be aborted. If no argument type checking is required `type` should be passed a `NULL` and no type checking will be performed.

The argument checking properties of this macro may be enabled / disabled via the `gObjectChecking` generic. It defaults to ON.

Since Dynace automatically checks the first argument to all generics (if argument checking is on) there is never a need to manually check the first argument. `ChkArgTypNul` would only be needed for arguments beyond the first.

`ChkArgTypNul` may only be used as the first executable statements in a method (prior to any other generic calls).

Example:

```
imeth object  MyMethod(object self, object arg2, object arg3)
{
        accessIVs;

        ChkArgTypNul(arg2, 2, String);
        ChkArgTypNul(arg3, 3, Set);
        .
        .
        .
```

See also: `ObjectChecking::Dynace`, `IsObj`, `IsKindOf::Object`, `ChkArg`, `ChkArgNul`, `ChkArgTyp`

`ClassOf` [ClassOf]

```
c = ClassOf(obj);

object  obj;    /*  an object                   */
object  c;      /*  the class of the object     */
```

This macro provides a very fast way of obtaining the class of any object. Since all objects in Dynace (including classes) are also regular objects and since all objects are instances of some class the `ClassOf` macro may be called repeatedly and indefinitely for any Dynace object and you would always receive a valid object. This macro may also be used to map any class hierarchy.

Example:

```
object  a, b;

a = gNewWithInt(ShortInteger, 7);
b = ClassOf(a);   /*  b now contains ShortInteger  */
```

See also: `IsKindOf::Object`, `IsInstanceOf`

`cmcPointer` [cmcPointer]

```
meth = cmcPointer(cls, gen);

object  cls;            /*  the class           */
object  (*gen)();       /*  generic function    */
RTYPE   (*meth)();      /*  method              */
```

This macro returns the class method associated with class `cls` and generic function `gen`. The normal runtime search is used to obtain the method. The process performed by this macro accounts for all the additional runtime overhead associated with the runtime binding of Dynace generic function usage. The method pointer returned may be used as a normal C function pointer, thus nullifying all future runtime overhead associated with dynamic binding. This macro may therefore be used to locally cache a method and eliminate any runtime costs while retaining the full dynamic capabilities of Dynace.

This macro correctly typecasts its returned pointer to agree with the associated generic's return type and argument list.

The use of this macro should be used sparingly. It is a little cumbersome to use and, if not used with care, may defeat the dynamic nature and benefits of dynamic binding. However, if used appropriately (in tight loops where efficiency is important), this macro may be a valuable tool.

This macro returns a pointer to the C function which implements the method. `NULL` is returned if no method is found.

Example:

```
vNew_t    meth;
object  vec[100];
int     i;

meth = cmcPointer(ShortInteger, vNew);
for (i=0 ; i != 100 ; )
       vec[i++] = meth(ShortInteger, 4);
/* or: vec[i++] = (*meth)(ShortInteger, 4); */
       /*  on old C compilers */
/* same as vec[i++] = gNewWithInt(ShortInteger, 4);  */
/* except 0 runtime overhead in loop          */
```

See also: `cmiPointer, imiPointer, imcPointer, FindMethod::Behavior`

cmeth [cmeth]

```
cmeth
```

This macro is used to label and introduce a C function which is to serve as a class method. All it really does is declare the C function to be static, and therefore not directly accessible externally to other source modules. It is important that all methods be declared static to enforce the encapsulation and enable name space overloading.

Example:

```
cmeth object  New(self)
object self;
{
        ...
}
```

See also: `imeth`, `method`

`cMethodFor` [cMethodFor]

```
meth = cMethodFor(cls, gen, cf);

object  cls;            /*  class               */
object  (*gen)();       /*  generic function    */
object  (*cf)();        /*  C function          */
object  meth;           /*  method object       */
```

This macro is used to associate a C function, which implements a class method, with a generic function for class `cls`. It is normally only used within the class initialization function associated with each class file.

This macro returns a Dynace object (an instance of the Method class) which represents the C function which implements the method. Each Class and Generic object contains a list of Method objects which it relates to so there is not normally a need to save the returned value.

Example:

```
cMethodFor(MyClass, gNew, my_new);
```

See also: `cvMethodFor`, `iMethodFor`

`cmiPointer` [cmiPointer]

```
meth = cmiPointer(i, gen);

object  i;              /*  the instance object */
object  (*gen)();       /*  generic function    */
RTYPE   (*meth)();      /*  method              */
```

This macro returns the class method associated with the class of instance `i` and generic function `gen`. The normal runtime search is used to obtain the method. The process performed by this macro accounts for all the additional runtime overhead associated with the runtime binding of Dynace generic function usage. The method pointer returned may be used as a normal C function pointer, thus nullifying all

future runtime overhead associated with dynamic binding. This macro may therefore be used to locally cache a method and eliminate any runtime costs while retaining the full dynamic capabilities of Dynace.

This macro correctly typecasts its returned pointer to agree with the associated generic's return type and argument list.

The use of this macro should be used sparingly. It is a little cumbersome to use and, if not used with care, may defeat the dynamic nature and benefits of dynamic binding. However, if used appropriately (in tight loops where efficiency is important), this macro may be a valuable tool.

This macro returns a pointer to the C function which implements the method. `NULL` is returned if no method is found.

Example:

```
vNew_t    meth;
object  vec[100], x;
int     i;

x = gNewWithInt(ShortInteger, 5);
/* normally x would have come from somewhere else  */
meth = cmiPointer(x, vNew);
for (i=0 ; i != 100 ; )
        vec[i++] = meth(ShortInteger, 4);
/* or:  vec[i++] = (*meth)(ShortInteger, 4); */
         /*  on old C compilers */
/* same as vec[i++] = gNewWithInt(ShortInteger, 4);  */
/* except 0 runtime overhead in loop          */
```

See also: `cmcPointer`, `imiPointer`, `imcPointer`, `FindMethod::Behavior`

`cSuper` [cSuper]

```
meth = cSuper(cls, gen);

object  cls;            /*  class               */
RTYPE   (*gen)();       /*  generic function    */
RTYPE   (*meth)();      /*  method              */
```

This macro returns the superclass class method associated with the generic function `gen` for class `cls`. If the method is not found the normal method search procedure will be used.

`RTYPE` is the return type associated with the generic and macro in question. The method pointer returned will be correctly typecast.

This macro has been superseded by `oSuper`.

Example:

```
cSuper(MyClass, vNew)(self);  /* execute the superclass class
                                 method associated with vNew
                                 and pass it the single
                                 argument self  */
```

See also: `oSuper, iSuper`

cvMethodFor [cvMethodFor]

```
meth = cvMethodFor(cls, gen, cf, ff);

object  cls;            /*  class               */
object  (*gen)();       /*  generic function    */
object  (*cf)();        /*  C function          */
object  (*ff)();        /*  fixed C function    */
object  meth;           /*  method object       */
```

This macro is used to associate a C function, which implements a class method, with a generic function for class `cls`. It is normally only used within the class initialization function associated with each class file.

This macro returns a Dynace object (an instance of the Method class) which represents the C function which implements the method. Each Class and Generic object contains a list of Method objects which it relates to so there is not normally a need to save the returned value.

The difference between this macro and `cMethodFor` is the extra `ff` parameter. When using `dpp` strategies other then 1 (see the `dpp` section) under certain circumstance `dpp` will alter the argument signature of the defined method in order to accommodate the non-assembler generic / method interface. It will then create a new function which has the correct signature in case the programmer requests a pointer to the method. This way the returned function pointer will match the argument signature defined in the class definition file. `ff` is a pointer to this cover function.

Example:

```
cvMethodFor(MyClass, gNew, my_new, f_my_new);
```

See also: `iMethodFor, cMethodFor, ivMethodFor`

defGeneric [defGeneric]

```
defGeneric(typ, gen)

type    typ;       /*  any valid type  */
object  (*gen)();  /*  generic name    */
```

The `defGeneric` macro is used to create a new generic function. All generic functions are normally created in a single source file called `generics.c` which is automatically created by the `dpp` program. `typ` defines the normal return type of the generic and `gen` is the name of the new generic function.

There would not normally be a reason to use this macro.

Example:

```
defGeneric(object, vNew)
```

See also: `dpp`, `externGeneric`, `InitGeneric`, `NewGeneric::GenericFunction`

END [END]

```
END
```

The `END` macro is simply syntactic sugar and is defined as `(object) 0`. It is used when creating a new class to indicate the end of the list of superclasses.

Example:

```
CLASS = gNewClass(Class, Dictionary, END);
```

See also: `NewClass::Class`

EQ [EQ]

```
r = EQ(obj1, obj2);

object  obj1, obj2;
int     r;
```

This macro is used to determine if `obj1` and `obj2` refer to the same exact object (not just the same value within the object). If so a 1 is returned and 0 otherwise.

Example:

```
object  a, b, e;
int     c;

a = gNewWithInt(ShortInteger, 7);
b = gNewWithInt(ShortInteger, 7);
e = a;
c = EQ(a, b);   /* c == 0  */
c = EQ(b, e);   /* c == 0  */
c = EQ(a, e);   /* c == 1  */
```

See also: `Equal::Object`, `NEQ`

externGeneric [externGeneric]

```
externGeneric(typ, gen)

type    typ;       /*  any valid type  */
object  (*gen)();  /*  generic name    */
```

The `externGeneric` macro is used to declare a generic function to external modules. All generic functions are normally declared in a single source file called `generics.h` which is automatically created by the `dpp` program. `typ` defines the normal return type of the generic and `gen` is the name of the generic function.

There would not normally be a reason to call this macro.

Example:

```
externGeneric(object, vNew)
```

See also: `dpp`, `defGeneric`, `InitGeneric`

Generic [Generic]

```
gen = Generic(name)

object  (*name)();
object  gen;
```

The `Generic` macro is used to obtain the generic object from its associated C function which implements the generic.

Example:

```
object  obj;

obj = Generic(vNew)
```

See also: `FindMethodObject::Behavior`

GetArg [GetArg]

```
GetArg(type)
```

This macro is used in conjunction with the `MAKE_REST` macro in order to create a consistent mechanism for methods which take a variable number of arguments to obtain its optional arguments. This mechanism, as opposed to the normal `va_*` mechanism, is necessary in order to allow the same source code to work across the various `dpp` code strategies (see the `dpp` subsection "Preprocessing Strategies").

`GetArg` may be used anytime subsequent to the `MAKE_REST` macro call. Its single argument is the type of the argument being taken off the variable argument list. Each successive call gets successive arguments after the last named argument.

See `MAKE_REST` for a complete example.

See also: `MAKE_REST, RESET_REST`

GetCVs [GetCVs]

```
p = GetCVs(cls);

object       cls;
CLASS_cv_t  *p;
```

This macro is used to obtain a pointer to the class variable structure locally defined in class `cls`. See the section "Internal Naming Conventions" for a description of `CLASS_cv_t`.

Example:

```
MyClass_cv_t  *cv = GetCVs(MyClass);
```

See also: `GetIVs`

`GetIVs` [GetIVs]

```
p = GetIVs(cls, obj);

object       cls;
object       obj;
CLASS_iv_t  *p;
```

This macro is used to obtain a pointer to the instance variable structure locally defined in class `cls` and associated with object `obj`. `obj` must, at some level, be an instance of `cls` or one of its subclasses. See the section "Internal Naming Conventions" for a description of `CLASS_iv_t`.

Example:

```
MyClass_iv_t  *iv = GetIVs(MyClass, myObj);
```

See also: `GetCVs`, `accessIVs`, `ivPtr`

`imcPointer` [imcPointer]

```
meth = imcPointer(cls, gen);

object  cls;            /*  the class           */
object  (*gen)();       /*  generic function    */
RTYPE   (*meth)();      /*  method              */
```

This macro returns the instance method associated with an instance of class `cls` and generic function `gen`. The normal runtime search is used to obtain the method. The process performed by this macro accounts for all the additional runtime overhead associated with the runtime binding of Dynace generic function usage. The method pointer returned may be used as a normal C function pointer, thus nullifying all future runtime overhead associated with dynamic binding. This macro may therefore be used to locally cache a method and eliminate any runtime costs while retaining the full dynamic capabilities of Dynace.

This macro correctly typecasts its returned pointer to agree with the associated generic's return type and argument list.

The use of this macro should be used sparingly. It is a little cumbersome to use and, if not used with care, may defeat the dynamic nature and benefits of dynamic binding. However, if used appropriately (in tight loops where efficiency is important), this macro may be a valuable tool.

This macro returns a pointer to the C function which implements the method. `NULL` is returned if no method is found.

Example:

```
gChangeValue_t  meth;
object  x;
int     i;

x = gNewWithInt(ShortInteger, 5);
meth = imcPointer(ShortInteger, gChangeValue);
for (i=0 ; i != 100 ; )
        meth(x, i*2);
/* or use:  (*meth)(x, i*2);  on old C compilers */
/* same as gChangeValue(x, i*2);  */
/* except 0 runtime overhead in loop          */
```

See also: `imiPointer`, `cmiPointer`, `cmcPointer`, `FindMethod::Behavior`

`imeth` [imeth]

```
imeth
```

This macro is used to label and introduce a C function which is to serve as an instance method. All it really does is declare the C function to be static, and therefore not directly accessible externally to other source modules. It is important that all methods be declared static to enforce the encapsulation and enable name space overloading.

Example:

```
imeth object  ChangeValue(self)
object self;
{
        ...
}
```

See also: `cmeth`, `method`

`iMethodFor` [iMethodFor]

```
meth = iMethodFor(cls, gen, cf);

object  cls;            /*  class               */
object  (*gen)();       /*  generic function    */
object  (*cf)();        /*  C function          */
object  meth;           /*  method object       */
```

This macro is used to associate a C function, which implements an instance method, with a generic function for class `cls`. It is normally only used within the class initialization function associated with each class file.

This macro returns a Dynace object (an instance of the Method class) which represents the C function which implements the method. Each Class and Generic object contains a list of Method objects which it relates to so there is not normally a need to save the returned value.

Example:

```
iMethodFor(MyClass, gChangeValue, my_change);
```

See also: `ivMethodFor, cMethodFor`

`imiPointer` [imiPointer]

```
meth = imiPointer(i, gen);

object  i;              /*  the instance        */
object  (*gen)();       /*  generic function    */
RTYPE   (*meth)();      /*  method              */
```

This macro returns the instance method associated with an instance `i` and generic function `gen`. The normal runtime search is used to obtain the method. The process performed by this macro accounts for all the additional runtime overhead associated with the runtime binding of Dynace generic function usage. The method pointer returned may be used as a normal C function pointer, thus nullifying all future runtime overhead associated with dynamic binding. This macro may therefore be used to locally cache a method and eliminate any runtime costs while retaining the full dynamic capabilities of Dynace.

This macro correctly typecasts its returned pointer to agree with the associated generic's return type and argument list.

The use of this macro should be used sparingly. It is a little cumbersome to use and, if not used with care, may defeat the dynamic nature and benefits of dynamic binding. However, if used appropriately (in tight loops where efficiency is important), this macro may be a valuable tool.

This macro returns a pointer to the C function which implements the method. `NULL` is returned if no method is found.

Example:

```
gChangeValue_t    meth;
object  x;
int     i;

x = gNewWithInt(ShortInteger, 5);
meth = imiPointer(x, gChangeValue);
for (i=0 ; i != 100 ; )
        meth(x, i*2);
/* or use:  (*meth)(x, i*2);  on old C compilers */
/* same as gChangeValue(x, i*2);  */
/* except 0 runtime overhead in loop          */
```

See also: `imcPointer`, `cmiPointer`, `cmcPointer`, `FindMethod::Behavior`

`InitGeneric` [InitGeneric]

```
InitGeneric(gen);

object  (*gen)();
```

The `InitGeneric` macro is used to create Dynace generic function objects which know the address of the C function which implements it.

All generic functions are normally initialized from a single source file called `generics.c` which is automatically created by the `dpp` program. `gen` is the name of the generic function.

There would not normally be a reason to call this macro.

Example:

```
InitGeneric(vNew);
```

See also: `dpp`, `NewGeneric::GenericFunction`, `defGeneric`, `externGeneric`

`IsaClass` [IsaClass]

```
r = IsaClass(obj);

object  obj;
int     r;
```

This macro is used to determine if `obj` refers to a class type object. It returns a 1 for class and metaclass type objects and 0 for instance type objects.

Example:

```
object  a;
int     c;

a = gNewWithInt(ShortInteger, 7);
c = IsaClass(a);            /* c == 0  */
c = IsaClass(ShortInteger)  /* c == 1  */
c = IsaClass(ClassOf(a));   /* c == 1  */
c = IsaClass(ClassOf(ShortInteger))  /* c == 1  */
```

See also: `IsaMetaClass`, `ClassOf`, `EQ`

IsaMetaClass [IsaMetaClass]

```
r = IsaMetaClass(obj);

object  obj;
int     r;
```

This macro is used to determine if `obj` refers to a metaclass type object. It returns a 1 for metaclass type objects and 0 for instance and class type objects.

Example:

```
object  a;
int     c;

a = gNewWithInt(ShortInteger, 7);
c = IsaMetaClass(a);            /* c == 0  */
c = IsaMetaClass(ShortInteger)  /* c == 0  */
c = IsaMetaClass(ClassOf(a));   /* c == 0  */
c = IsaMetaClass(ClassOf(ShortInteger))  /* c == 1  */
```

See also: `IsaClass`, `ClassOf`, `EQ`

IsInstanceOf [IsInstanceOf]

```
r = IsInstanceOf(i, cls);

object  i;
object  cls;
int     r;
```

This macro is used to determine if `i` is a direct instance of `cls`. If so a 1 is returned and 0 otherwise.

Example:

```
object  a, b;
int     c;

a = gNewWithInt(ShortInteger, 7);
b = gNewWithInt(Dictionary, 101);
c = IsInstanceOf(a, ShortInteger);  /* c == 1  */
c = IsInstanceOf(a, LongInteger);   /* c == 0  */
c = IsInstanceOf(b, Dictionary);    /* c == 1  */
c = IsInstanceOf(b, Set);           /* c == 0  */
c = IsInstanceOf(b, LinkList);      /* c == 0  */
```

See also: `IsKindOf::Object`, `ClassOf`

`iSuper` [iSuper]

```
meth = iSuper(cls, gen);

object  cls;            /*  class               */
RTYPE   (*gen)();       /*  generic function    */
RTYPE   (*meth)();      /*  method              */
```

This macro returns the superclass instance method associated with the generic function `gen` for class `cls`. If the method is not found the normal method search procedure will be used.

`RTYPE` is the return type associated with the generic and macro in question. The method pointer returned will be correctly typecast.

This macro has been superseded by `oSuper`.

Example:

```
 /* execute the superclass instance
    method associated with gDispose
    and pass it the single argument self  */
iSuper(MyClass, gDispose)(self);
```

See also: `oSuper`, `cSuper`

`ivMethodFor` [ivMethodFor]

```
meth = ivMethodFor(cls, gen, cf, ff);

object  cls;            /*  class               */
object  (*gen)();       /*  generic function    */
object  (*cf)();        /*  C function          */
object  (*ff)();        /*  fixed C function    */
object  meth;           /*  method object       */
```

This macro is used to associate a C function, which implements an instance method, with a generic function for class `cls`. It is normally only used within the class initialization function associated with each class file.

This macro returns a Dynace object (an instance of the Method class) which represents the C function which implements the method. Each Class and Generic object contains a list of Method objects which it relates to so there is not normally a need to save the returned value.

The difference between this macro and `iMethodFor` is the extra `ff` parameter. When using `dpp` strategies other then 1 (see the `dpp` section) under certain circumstance `dpp` will alter the argument signature of the defined method in order to accommodate the non-assembler generic / method interface. It will then create a new function which has the correct signature in case the programmer requests a pointer to the method. This way the returned function pointer will match the argument signature defined in the class definition file. `ff` is a pointer to this cover function.

Example:

```
ivMethodFor(MyClass, gChangeValue, my_change, f_my_change);
```

See also: `iMethodFor, cMethodFor`

`ivPtr` [ivPtr]

```
iv = ivPtr(i);

object  i;
ivType  *iv;
```

The `ivPtr` macro returns a pointer to the locally defined instance variable structure associated with instance `i`. It is used to gain access to the instance variables associated with the instance object pointed to by the `i` variable. It may only be used in a file which defines a class and `i` must be an instance of the class being defined or one of its subclasses.

If the method is going to make more than one access to its instances variables the pointer returned by `ivPtr` should normally be saved and reused.

If the variable containing the instance variables you wish to access is named `self` than the `ivsPtr` or `accessIVs` macro should be used.

Although the example below makes its illustration with the `self` variable (because it's always the name of the first argument to a method) `ivPtr` is most often used to access the instance variables of a second argument (which wouldn't have the name `self`) which is an instance of the same (or sub) class.

Example:

```
defclass  {
    char    name[30];
    int     age;
};

imeth object  PrintAge(object self)
{
    printf("Age = %d\n", ivPtr(self)->age);
    return self;
}

imeth object  PrintInstance(object self)
{
    ivType  *iv = ivPtr(self);      /*  save the pointer */
/* the above line is the same as accessIVs  */
    printf("Name = %s\n", iv->name); /* use saved pointer   */
    printf("Age = %d\n", iv->age);
    return self;
}
```

See also: `accessIVs`, `ivsPtr`, `ivType`, `GetIVs`

`ivsPtr` [ivsPtr]

```
iv = ivsPtr;

ivType  *iv;
```

The `ivsPtr` macro returns a pointer to the locally defined instance variable structure for the variable named `self`. It is used to gain access to the instance variables associated with the instance object pointed to by the `self` variable. It may only be used in instance methods, in a file which defines a class.

This macro is typically used when an instance method makes a single access to its instance variables. If the method is going to make more than one access to its instances variables the macro `accessIVs` would normally be used or the pointer returned by `ivsPtr` would normally be saved and reused.

If the variable containing the instance variables you wish to access is not named `self` than the `ivPtr` macro should be used.

Example:

```
defclass  {
    char    name[30];
    int     age;
};

imeth object  PrintAge(object self)
{
    printf("Age = %d\n", ivsPtr->age);  /*  single use      */
    return self;
}

imeth object  PrintInstance(object self)
{
    ivType  *iv = ivsPtr;   /*  save the pointer */
/* the above line is the same as accessIVs  */
    printf("Name = %s\n", iv->name); /* use saved pointer   */
    printf("Age = %d\n", iv->age);
    return self;
}
```

See also: `accessIVs`, `ivPtr`, `ivType`, `GetIVs`

`ivType` [ivType]

```
ivType
```

This macro is used to declare a variable to be of a structure which is defined by the local instance variable declaration. It can only be used in a source file which is used to define a class and only of the structure of the locally defined instance variable structure. It's main use is while passing a pointer to a local variable structure to other, usually statically defined, C functions within a class definition file.

Example:

```
ivType  *ivp;

ivp = ivsPtr;
```

See also: `accessIVs`, `ivPtr`, `ivsPtr`

MAKE_REST [MAKE_REST]

```
MAKE_REST(lst);
```

This macro is used in conjunction with the GetArg macro in order to create a consistent mechanism for methods which take a variable number of arguments to obtain its optional arguments. This mechanism, as opposed to the normal va_* mechanism, is necessary in order to allow the same source code to work across the various dpp code strategies (see the dpp subsection "Preprocessing Strategies").

Methods which take a variable number of arguments and wish to access those arguments must use the MAKE_REST macro. This macro must exist immediately following the declaration section of the method and prior to the normal executable code.

MAKE_REST takes a single argument which must be the last named argument in the method's argument list. The result of calling MAKE_REST is the creation of a variable named _rest_ which acts as a pointer to the variable argument list, a va_list.

Example:

```
imeth  gChangeValue(int a, char *b, ...)
{
        double  var1;
        int     dd;
        MAKE_REST(b);

        var1 = GetArg(double);
        dd = GetArg(int);
        ...
}
```

See also: GetArg, RESET_REST

NEQ [NEQ]

```
r = NEQ(obj1, obj2);

object  obj1, obj2;
int     r;
```

This macro is used to determine if obj1 and obj2 refer to the same exact object (not just the same value within the object). If so a 0 is returned and 1 otherwise.

Example:

```
object  a, b, e;
int     c;

a = gNewWithInt(ShortInteger, 7);
b = gNewWithInt(ShortInteger, 7);
e = a;
c = NEQ(a, b);   /* c == 1  */
c = NEQ(b, e);   /* c == 1  */
c = NEQ(a, e);   /* c == 0  */
```

See also: `Equal::Object`, `EQ`

oSuper [oSuper]

```
meth = oSuper(cls, gen, obj);

object  cls;
RTYPE   (*gen)();
object  obj;
RTYPE   (*meth)();
```

This macro returns the superclass instance or class method associated with the generic function `gen` for class `cls`. `obj` is used to tell whether the object being acted upon is a class or instance object. It should be the argument which will be the first argument to the methods being returned. If the method is not found the normal method search procedure will be used.

`RTYPE` is the return type associated with the generic and macro in question. The method pointer returned will be correctly typecast.

This macro effectively eliminates the need for the `iSuper` or `cSuper` macros.

Example:

```
 /* execute the superclass
    method associated with gDispose
    and pass it the single argument self  */
oSuper(MyClass, gDispose, self)(self);
```

See also: `iSuper`, `cSuper`

`RegisterVariable` [RegisterVariable]

```
p = RegisterVariable(v);

object  v;
void    *p;
```

The `RegisterVariable` macro is used as a short hand way of registering global object variables to protect them from the automatic garbage collector. `v` is the global variable to be protected and `p` is a registered memory pointer which may be used to change or delete the registration at some later point.

See the text for a detailed description of the garbage collector.

See also: `RegisterMemory::Dynace`, `RemoveRegisteredMemory::Dynace`, `ChangeRegisteredMemory::Dynace`

`RESET_REST` [RESET_REST]

```
RESET_REST;
```

This macro is used in conjunction with the `GetArg` and `MAKE_REST` macros in order to create a consistent mechanism for methods which take a variable number of arguments to obtain its optional arguments. This macro allows the `GetArg` to go back over the argument list from the beginning, thus allowing multiple passes over the entire argument list. `RESET_REST` may be called anytime, and any number of times, after `MAKE_REST` has been called. Each time `RESET_REST` is called, `GetArg` will start back at the beginning of the argument list.

Example:

```
imeth  gChangeValue(int a, char *b, ...)
{
        double  var1;
        int     dd;
        MAKE_REST(b);

        var1 = GetArg(double);
        dd = GetArg(int);
        /*  get the same arguments again  */
        RESET_REST;
        var1 = GetArg(double);
        dd = GetArg(int);
        ...
}
```

See also: `GetArg`, `MAKE_REST`

RespondsTo [RespondsTo]

```
r = RespondsTo(i, gen);

object  i;
object  (*gen)();
int     r;
```

This macro is used to determine if the class of `i` implements a method which is associated with generic `gen`. If so a 1 is returned and 0 otherwise.

Example:

```
object  a;
int     c;

a = gNewWithInt(ShortInteger, 7);
c = RespondsTo(a, gShortValue);  /* c == 1  */
c = RespondsTo(a, gNext);        /* c == 0  */
```

See also: `FindMethod::Behavior`

StackAlloc [StackAlloc]

```
i = StackAlloc(cls);

object  cls;
object  i;
```

This macro is used to allocate a new instance of class `cls` from the local stack space. `i` will be an instance of class `cls` with all its instance variables initialized to `NULL`.

Disposal of the returned object will occur automatically when the local stack is exited, however, no automatic execution of additional disposal code will occur. Manual disposal may also be used (via `gDispose`, etc.) but the object will still take up stack space until the stack frame is exited.

Since the returned object is allocated from the current stack frame it cannot be returned from the function which allocated it.

This facility depends on a robust implementation of the `alloca` system call.

Example:

```
object  a;

a = StackAlloc(ShortInteger);
```

See also: `StackAlloc::Behavior`

4.10 Kernel Functions

The number of regular C language functions available externally has been kept to an absolute minimum. The reason for this is the static nature of C functions and an effort to minimize the number of routines the programmer needs to use the system. The following text describes each function.

`Dynace_GetInitialPageSize` [Dynace_GetInitialPageSize]

```
sz = Dynace_GetInitialPageSize();

long    sz;
```

This is an application defined function optionally supplied to determine the amount of memory Dynace should allocate upon startup. If used, it must be defined and supplied as part of the application's code. The value returned, given in number of bytes, is what Dynace uses to allocate its initial block of memory for various objects to be allocated from. This is used to speed up Dynace under Windows. See [Speeding-Up-IsObj], page 76.

Example:

```
The following function would be defined by the application:

long    Dynace_GetInitialPageSize()
{
        return 1000000;
}
```

See also: `Dynace_GetPageSize`

`Dynace_GetPageSize` [Dynace_GetPageSize]

```
sz = Dynace_GetPageSize();

long    sz;
```

This is an application defined function optionally supplied to determine the amount of memory Dynace should allocate after its startup allocation. If used, it must be defined and supplied as part of the application's code. The value returned, given in number of bytes, is what Dynace uses to allocate its blocks of memory, when needed, after its initial memory block is used up. This is used to speed up Dynace under Windows. See [Speeding-Up-IsObj], page 76.

Example:

```
The following function would be defined by the application:

long    Dynace_GetPageSize()
{
        return 100000;
}
```

See also: `Dynace_GetInitialPageSize`

`GetIVptr` [GetIVptr]

```
iv = GetIVptr(obj, cls);

object  obj;
object  cls;
void    *iv;
```

This function is the one which does all the work to obtain a pointer to the instances variables from an object. `obj` is the object whose instance variables are needed. `cls` is the class which describes the structure of the instance variables needed. `obj` must be an instance of class `cls` at some point in its hierarchy. The pointer returned will only allow access to the instance variables defined by class `cls` regardless of any other instance variables `obj` may contain.

This function should not be called by an application. The relevant macros should be used instead.

See also: `ivPtr`, `ivsPtr`, `accessIVs`

`InitDynace` [InitDynace]

```
InitDynace(sb);

void    *sb;  /*  stack beginning  */
```

This function is used to initialize the Dynace kernel and all classes and generics. It must be called prior to the use of any other Dynace facility. It is typically called in `main()`.

This function calls `InitKernel` internally to initialize the kernel first. The `sb` parameter tells Dynace where the beginning of the stack resides. The garbage collector needs this information to automatically scan automatic variables. `sb` is typically passed the address of the `argc` argument to `main()`.

Example:

```
main(int argc, char *argv[])
{
        InitDynace(&argc);
        .....
}
```

See also: `InitKernel`, `GC::Dynace`

`InitKernel` [InitKernel]

```
InitKernel(sb);

void    *sb;  /*  stack beginning  */
```

This function is used to initialize the Dynace kernel. It must be called prior to the use of any other Dynace facility. Typically, however, `InitKernel` is called indirectly through another function (such as `InitDynace` which also initializes the class library and user added classes). The `sb` parameter tells Dynace where the beginning of the stack resides. The garbage collector needs this information to automatically scan automatic variables. `sb` is typically passed the address of the `argc` argument to `main()`.

Example:

```
main(int argc, char *argv[])
{
        InitKernel(&argc);
        .....
}
```

See also: `InitDynace`, `GC::Dynace`

`IsObj` [IsObj]

```
r = IsObj(obj);

object  obj;
int     r;
```

This function is used to determine the validity of an object. If `obj` is a valid Dynace object `IsObj` returns a 1 otherwise it returns a 0.

Example:

```
if (IsObj(obj))
        gGeneric(obj);
else
        abort("Not an object");
```

See [Speeding-Up-IsObj], page 76.

4.11 Kernel Data Types

There is really only one new data type associated with Dynace, the `object` type. However, Dynace also has some type definitions which are simply used to ease programming in the Dynace environment.

`object` [object]

```
object
```

This is the principal type of object in Dynace. All Dynace objects are of type `object`. `object` is actually a pointer to memory which is allocated (by Dynace) from the heap. Objects never reside on the stack or static memory, although pointers to them may.

The `object` declaration may be used any place the `int` declaration may be used.

Example:

```
object  a, b, c;
```

`ofun` [ofun]

```
ofun
```

This declaration is just a short hand way of declaring a variable to be a pointer to a function which returns an object.

The `ofun` declaration may be used any place the `int` declaration may be used.

Example:

```
ofun    a;
object  (*b)();

/*  a and b are of the same type  */
```

5 Class Library Reference

The Dynace class library is built on top of the Dynace kernel. It provides a library of facilities which make the Dynace system useful to an application programmer. It provides various data representational type classes which represent the normal C language data types such as short, long, double, pointer, etc. It also provides new data type classes such as date.

The class library also provides generic container type classes such as dictionaries and linked lists. It also provides classes which implement the ability for Dynace to run multiple threads and for those threads to communicate and control each other.

This chapter will discuss the structure and various facilities available in the Dynace class library.

Note that most of the Dynace kernel will run independently of the Dynace class library. The value of this is that should a programmer desire to create his own class library (to replace the included class library) he may build on top of the Dynace kernel just as the included library does.

Note that the first argument to all class methods is always the associated class object and the first argument to all instance methods is always an instance of the associated class. Therefore, documentation for the first argument of generics is not always given – it's redundant.

There is a naming convention used with Dynace generics. All generics start with either a lower case "g" or "v", and are always followed by an upper case letter. The ones which start with "g" are normal generics and may be treated like normal C functions. The ones which begin with "v" use the variable argument facilities of C and you should, therefore, take a bit extra care when using them since there is no compile time argument checking being done with these functions.

Since all generics start with either "g" or "v" and in order to avoid the difficulty associated with grouping all the generics under two letters, the first letter is dropped for indexing or heading purposes. Therefore if you are looking up a generic, it will always appear in the index or header with its first letter missing. The syntax description and example code, however, will show the entire name.

5.1 Class Library Hierarchy

This Dynace class library contains the following class hierarchy (which of course includes the hierarchy described in the Kernel Reference):

Dynace Class Hierarchy

- Object
  - Behavior
    - Class
    - MetaClass
  - Method
  - GenericFunction
  - Dynace
  - Number
    - Character
    - ShortInteger
    - UnsignedShortInteger
    - LongInteger
      - Date
        - DateTime *
    - DoubleFloat
  - Time
    - DateTime *
  - Pointer
  - Link
    - LinkList
      - LinkObject
    - LinkValue
  - Sequence
    - LinkSequence
    - LinkObjectSequence
    - SetSequence
  - Association
    - LookupKey
      - ObjectAssociation
    - StringAssociation
    - IntegerAssociation
  - Set
    - Dictionary
    - StringDictionary
    - IntegerDictionary
  - Stream
    - String
    - File
    - LowFile
    - Pipe
    - Socket
  - PropertyList
  - Thread
  - Semaphore
  - Constant
  - Array
    - NumberArray
      - CharacterArray
      - ShortArray
      - UnsignedShortArray
      - IntegerArray
      - LongArray
      - FloatArray
      - DoubleFloatArray
    - BitArray
    - ObjectArray
    - PointerArray
  - FindFile
  - BTree
  - BTreeNode

* DateTime inherits from Date and Time

5.2 Array Class

The `Array` class is used to represent heterogeneous and non-heterogeneous indexable collections of arbitrary dimensions. These arrays are designed to be flexible – they have the ability to be dynamically re-dimensioned or reshaped – and space and speed efficient. Although most of the actual functionality contained in subclasses of the `Array` class are implemented in the `Array` class, most users will find the subclasses more useful. The documentation associated with this class serves to document the functionality which is common to all its subclasses.

5.2.1 Array Class Methods

The majority of class methods associated with this class are mainly used by and documented in subclasses of the `Array` class.

`IndexOrigin::Array` [IndexOrigin]

```
s = gIndexOrigin(s, n)

object  s;
int     n;
```

This method is used to set the index origin associated with *all* array indexing. The default is 0. This means that a 4 element, 1 dimension array would be indexed by 0, 1, 2, 3. If the index origin is set to one that same array would be indexed by 1, 2, 3, 4.

The index origin may be changed any number of times and has no actual effect on any arrays (either created previously or subsequently). It only effects how indexing works.

5.2.2 Array Instance Methods

Instance methods associated with this class provide functionality which is common to all subclasses of the `Array` class.

`ArrayPointer::Array` [ArrayPointer]

```
p = gArrayPointer(s)

object  s;
void    *p;
```

This method returns a pointer to the first element of an array represented be `s`. The return type will be a pointer to the type of array it is. For example a `ShorArray` will return a pointer to a `short` and a `DoubleFloatArray` will return a pointer to a `double`. Consecutive elements may be accessed by incrementing the pointer (although this is not the preferred way of going through an array).

`Copy::Array` [Copy]

```
r = gCopy(i);

object  i;
object  r;
```

This method is used to create a new array object which is an exact copy, including type, rank, shape and values, of `i`. The value returned is the new array object.

See also: `DeepCopy::Array`

`DeepCopy::Array` [DeepCopy]

```
r = gDeepCopy(i);

object  i;
object  r;
```

This method is used to create a new array object which is an exact copy, including type, rank, shape and values, of `i`. The value returned is the new array object.

The difference between this method and `Copy::Array` is that if the array being copied is an `ObjectArray`, `DeepCopy` will also make copies of the elements. `Copy` will not.

See also: `Copy::Array`

`DeepDispose::Array` [DeepDispose]

```
r = gDeepDispose(s)

object  s;
object  r;     /*  NULL  */
```

This method disposes and frees all memory associated with an `Array` object. If it is an instance of the `ObjectArray` class each non-NULL element will also be `DeepDispose`'d.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

`Dispose::Array` [Dispose]

```
r = gDispose(s)

object  s;
object  r;     /*  NULL  */
```

This method disposes and frees all memory associated with an `Array` object.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

`Equal::Array` [Equal]

```
r = gEqual(i, a)

object  i, a;
int     r;
```

This method is used to determine the equality of two arrays. If they (`i` and `a`) are instances of the same class, same rank, same shape and same values than `Equal` returns a 1. `Equal` returns 0 otherwise.

See also: `EQ`

`Index::Array` [Index]

```
p = gIndex(i, idx)

object     i;
va_list  idx;
```

This method is used to obtain a pointer to a single element of an array. It is only intended to be used internally by the subclasses of `Array`. `i` is the array to be indexed and `idx` is a va_list where each element indicates successive indices into the array. The number of elements in `idx` *must* equal the rank of the array. The pointer returned may be typecast to the appropriate type.

See also: `ChangeValue::NumberArray, ShortValue::NumberArray`

`Rank::Array` [Rank]

```
r = gRank(i)

object    i;
unsigned  r;
```

This method is used to obtain the number of dimensions a particular array (`i`) has.

See also: `Shape::Array, Size::Array, Reshape::Array`

`Reshape::Array` [Reshape]

```
i = vReshape(i, r, ...)

object     i;
unsigned   r;
```

This method is used to change the rank and shape of an array (`i`). `r` is the new rank (number of dimensions) of the array and the remaining arguments (type `unsigned`) indicate the size of each consecutive dimension. Not that the number of arguments after `r` *must* be equal to the number `r`.

If the new shape has fewer elements than the original, the remaining elements are discarded. If, however, the new shape has more elements, the elements in the original array will be reused (from the beginning) over and over until the new array is filled.

The value returned is the modified array passed.

See also: `Shape::Array, Rank::Array`

`Shape::Array` [Shape]

```
s = gShape(i)

object  i;
object  s;
```

This method is used to obtain the shape (or dimensions) of a particular array (`i`). The value returned (`s`) is an instance of the `ShortArray` class, each element of which contains a consecutive dimension.

See also: `Reshape::Array, Rank::Array, Size::Array`

`Size::Array` [Size]

```
s = gSize(i)

object  i;
int     s;
```

This method is used to obtain the total number of elements in array `i`.

See also: `Rank::Array, Shape::Array`

`StringRep::Array` [StringRep]

```
s = gStringRep(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the type, rank, shape and values associated with `i`. This is often used to print or display the value. It is also used by `PrintValue::Object` and indirectly by `Print::Object` (two methods useful during the debugging phase of a project) in order to directly print an object's value.

See also: `PrintValue::Object`, `Print::Object`, `StringRepValue::Array`

`StringRepValue::Array` [StringRepValue]

```
s = gStringRepValue(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the values associated with `i`. This is often used to print or display the value. It is also used by `PrintValue::Object` and indirectly by `Print::Object` (two methods useful during the debugging phase of a project) in order to directly print an object's value.

See also: `PrintValue::Object`, `Print::Object`, `StringRep::Array`

5.3 Association Class

The `Association` class is an abstract class which serves the sole purpose of grouping several subclasses which have common functionality. There is no specific class or instance methods directly implemented by this class, however, some methods are documented here because of their common interface with all subclasses of the `Association` class.

The subclasses of this class (`LookupKey`, `ObjectAssociation`, `StringAssociation` and `IntegerAssociation`) are used to associate a key with a value and provide a means for manipulating these associations. Instances of these classes are principally used by the `Set` and `Dictionary` classes but may also be useful wherever a key/value pair may be useful.

5.3.1 Association Class Methods

There are no class methods implemented by this class.

5.3.2 Association Instance Methods

There are no class methods implemented by this class, however, some methods are documented here because of their common interface with all subclasses of the `Association` class. These methods are actually implemented by all subclasses of this class.

`Compare::Association` [Compare]

```
r = gCompare(i, obj);

object  i;
object  obj;
int     r;
```

This method is used by the generic container classes to determine the equality of the keys represented by `i` and `obj`. `r` is -1 if the value represented by `i` is less than the value represented by `obj`, 1 if the value of `i` is greater than `obj`, and 0 if they are equal.

See also: `Hash::Association`

`Hash::Association` [Hash]

```
val = gHash(i);

object  i;
int     val;
```

This method is used by the generic container classes to obtain hash values for the key. `val` is a hash value between 0 and a large integer value.

See also: `Compare::Association`

5.4 BitArray Class

This class, which is a subclass of `Array`, is used to represent arbitrary shaped arrays of on/off–yes/no information in an efficient manner. Much of the functionality of this class is implemented and documented in the `Array` class. Differences are documented in this section.

5.4.1 BitArray Class Methods

The only class method implemented by this class is one used to create new `BitArray` instances.

`New::BitArray` [New]

```
ary = vNew(BitArray, rnk, ...)

unsigned  rnk, ...
object    ary;
```

This class method is used to create a new bit array.

`rnk` is the number of dimensions the new array should have. The remaining arguments (each of type unsigned) indicates the size of each consecutive dimension. Note that the number of arguments following `rnk` *must* be the same as the value in `rnk`.

`ary` is the new array object created and will be initialized to all zeros.

Example:

```
object  ary;

ary = vNew(BitArray, 2, 5, 4);
/*  ary is a 5x4 matrix  */
```

5.4.2 BitArray Instance Methods

The `BitArray` instance methods are implemented in the `Array` class for efficiency reasons but those methods which are particular to the `BitArray` are documented here.

`BitValue::BitArray` [BitValue]

```
v = vBitValue(ary, ...);

object    ary;
unsigned  ...
int       v;
```

This method is used to obtain the bit (1 or 0) value associated with a particular element of an instance of the `BitArray` class.

The arguments after the `ary` argument (each an `unsigned`) are used to specify the exact index into each consecutive dimension of the array. The number of arguments after the `ary` argument *must* be equal to the number of dimensions (or rank) of array `ary`. See `IndexOrigin::Array` for more information.

Note that this is one of the few generics which doesn't return a Dynace object. It returns an `int`.

Example:

```
object  ary;
int     v;

ary = vNew(BitArray, 2, 5, 4);
v = vBitValue(ary, 1, 2);
/*  v has ary[1][2]  */
```

See also: `ChangeBitValue::BitArray`

`ChangeBitValue::BitArray` [ChangeBitValue]

```
ary = vChangeBitValue(ary, val, ...);

object    ary;
int       val;
unsigned  ...
```

This method is used to change the value of one element of `BitArray` `ary`.

`val` is the value which the element of the array should be changed to (a 1 or 0).

The arguments after the `val` argument (each an `unsigned`) are used to specify the exact index into each consecutive dimension of the array. The number of arguments after the `val` argument *must* be equal to the number of dimensions (or rank) of array `ary`. See `IndexOrigin::Array` for more information.

The value returned is the modified array passed.

Example:

```
object  ary;

ary = vNew(BitArray, 2, 5, 4);
vChangeBitValue(ary, 1, 1, 2);
/*  ary[1][2] <- 1  */
```

See also: `BitValue::BitArray`

5.5 BTree Class

The `BTree` class provides a means to hold a group of key / value pairs in a collection. Objects in a `BTree` collection can then be rapidly accessed arbitrarily (as in hash tables) as well as retrieved in a specific sequence (as in an ordered linked lists). Keys and values may be arbitrary Dynace objects.

It is important that the key objects correctly respond to `gCopy` and `gDeepCopy` because copies of the keys are used within the key structure.

5.5.1 BTree Class Methods

The only class method used in this class is one to create instances of itself.

`New::BTree` [New]

```
bt = gNew(BTree);

object  bt;
```

This class method creates instances of the `BTree` class.

5.5.2 BTree Instance Methods

The instance methods associated with this class are used to add, remove, and retrieve key / value pairs.

`AddValue::BTree` [AddValue]

```
old = gAddValue(bt, key, val);

object  bt;
object  key;
object  val;
object  old;
```

This method is used to add an arbitrary key / value pair to BTree `bt`. Both `key` and `val` may be arbitrary Dynace objects.

By default the built in generic `gCompare` is used for all key comparisons. This may be over written via `gSetFunction::BTree`.

Each key added to a particular `BTree` must be unique (i.e. duplicate keys are not allowed).

`AddValue` normally returns `NULL`. However, if `NULL` is not returned then it means that the key / value pair just added replaced a similar key / value pair. In that case the old value associated with the key will be returned and the old key object will still be used. This will normally mean you will have to dispose of the old value returned and your new key (since the old one was used).

Example:

```
object  key = gNewWithStr(String, "My Key");
object  val = gNewWithStr(String, "My Value");
object  old = gAddValue(bt, key, val);
if (old) {
        gDispose(key);
        gDispose(old);
}
```

`DeepDispose::BTree` [DeepDispose]

```
r = gDeepDispose(bt);

object  bt;
object  r;     /*  NULL  */
```

This method is used to `DeepDispose` all key / value pairs associated with the `BTree` and then dispose of `bt` itself.

See also: `Dispose::BTree`, `DeepDisposeAllNodes::BTree`

`DeepDisposeAllNodes::BTree` [DeepDisposeAllNodes]

```
r = gDeepDisposeAllNodes(bt);

object  bt;
object  r;
```

This method is used to `DeepDispose` all key / value pairs associated with the `BTree` without disposing of `bt` itself. The `BTree` object passed is returned.

See also: `Dispose::BTree`, `DeepDispose::BTree`

`DeepDisposeObj::BTree` [DeepDisposeObj]

```
r = gDeepDisposeObj(bt, key);

object  bt;
object  key;
object  r;
```

This method is used to `DeepDispose` a specific key / value pair associated with the `BTree`. The key / value pair is determined by `key`. If the key / value pair was found and removed, `gDeepDisposeObj` will return non-`NULL`. If the key was not found then `NULL` is returned.

See also: `DisposeObj::BTree`, `DeepDisposeAllNodes::BTree`

`Dispose::BTree` [Dispose]

```
r = gDispose(bt);

object  bt;
object  r;     /*  NULL  */
```

This method is used to `Dispose` of the `BTree` object without disposing of all of the key / value pairs.

See also: `DeepDispose::BTree`, `DisposeAllNodes::BTree`

`DisposeAllNodes::BTree` [DisposeAllNodes]

```
r = gDisposeAllNodes(bt);

object  bt;
object  r;
```

This method is used to disassociate all key / value pairs from the given `BTree`. Neither the keys, values, nor `bt` are disposed. The `BTree` object passed is returned.

See also: `DeepDispose::BTree`, `DisposeAllNodes::BTree`

`DisposeObj::BTree` [DisposeObj]

```
r = gDisposeObj(bt, key);

object  bt;
object  key;
object  r;
```

This method is used to disassociate (not dispose) a specific key / value pair from the `BTree`. The key / value pair is determined by `key`. If the key / value pair was found and removed, `gDisposeObj` will return non-`NULL`. If the key was not found then `NULL` is returned.

See also: `DeepDisposeObj::BTree`, `DisposeAllNodes::BTree`

`FindEQ::BTree` [FindEQ]

```
val = gFindEQ(bt, key, fkey);

object  bt;
object  key;
object  *fkey;
object  val;
```

This method is used to retrieve the value associated with a given key. The object associated with a key which compares equally with `key` is returned. If none is found `NULL` is returned.

`fkey` points to a location which will receive the actual key object found. `fkey` may be `NULL` in which case that argument will not be used.

Example:

```
object  fkey;
object  key = ...;
object  val = gFindEQ(bt, key, &fkey);
```

See also: `AddValue::BTree`, `FindNext::BTree`

`FindFirst::BTree` [FindFirst]

```
val = gFindFirst(bt, fkey);

object  bt;
object  *fkey;
object  val;
```

This method is used to retrieve the first key / value pair. If no key value pairs exist `NULL` is returned.

`fkey` will be updated to point to the key associated to the value returned. It may be used on subsequent calls to `gFindNext`.

See also: `AddValue::BTree`, `FindNext::BTree`

FindGE::BTree [FindGE]

```
val = gFindGE(bt, key, fkey);

object  bt;
object  key;
object  *fkey;
object  val;
```

This method is used to retrieve the value associated with a given key. The object associated with a key which compares greater than or equal to `key` is returned. If none is found `NULL` is returned.

`fkey` points to a location which will receive the actual key object found. `fkey` may be `NULL` in which case that argument will not be used.

Example:

```
object  fkey;
object  key = ...;
object  val = gFindGE(bt, key, &fkey);
```

See also: `AddValue::BTree`, `FindNext::BTree`

FindGT::BTree [FindGT]

```
val = gFindGT(bt, key, fkey);

object  bt;
object  key;
object  *fkey;
object  val;
```

This method is used to retrieve the value associated with a given key. The object associated with a key which compares strictly greater than `key` is returned. If none is found `NULL` is returned.

`fkey` points to a location which will receive the actual key object found. `fkey` may be `NULL` in which case that argument will not be used.

Example:

```
object  fkey;
object  key = ...;
object  val = gFindGT(bt, key, &fkey);
```

See also: `AddValue::BTree`, `FindNext::BTree`

FindLast::BTree [FindLast]

```
val = gFindLast(bt, fkey);

object  bt;
object  *fkey;
object  val;
```

This method is used to retrieve the last key / value pair. If no key value pairs exist NULL is returned.

fkey will be updated to point to the key associated to the value returned. It may be used on subsequent calls to gFindPrev.

See also: AddValue::BTree, FindPrev::BTree

FindLE::BTree [FindLE]

```
val = gFindLE(bt, key, fkey);

object  bt;
object  key;
object  *fkey;
object  val;
```

This method is used to retrieve the value associated with a given key. The object associated with a key which compares less than or equal to key is returned. If none is found NULL is returned.

fkey points to a location which will receive the actual key object found. fkey may be NULL in which case that argument will not be used.

Example:

```
object  fkey;
object  key = ...;
object  val = gFindLE(bt, key, &fkey);
```

See also: AddValue::BTree, FindPrev::BTree

`FindLT::BTree` [FindLT]

```
val = gFindLT(bt, key, fkey);

object  bt;
object  key;
object  *fkey;
object  val;
```

This method is used to retrieve the value associated with a given key. The object associated with a key which compares strictly less than `key` is returned. If none is found `NULL` is returned.

`fkey` points to a location which will receive the actual key object found. `fkey` may be `NULL` in which case that argument will not be used.

Example:

```
object  fkey;
object  key = ...;
object  val = gFindLT(bt, key, &fkey);
```

See also: `AddValue::BTree`, `FindPrev::BTree`

`FindNext::BTree` [FindNext]

```
val = gFindNext(bt, fkey);

object  bt;
object  *fkey;
object  val;
```

This method is used to retrieve a key / value pair which comes immediately after the key pointed to by `fkey`. If `fkey` points to `NULL` than the first key / value pair is retrieved. If none is found `NULL` is returned.

`fkey` will be updated to point to the key associated to the value returned. It may be used on subsequent calls to `gFindNext`.

Example:

```
object  fkey = NULL;
object  val;

while (val = gFindNext(bt, &fkey))
     do whatever;
```

See also: `AddValue::BTree`, `FindPrev::BTree`

`FindPrev::BTree` [FindPrev]

```
val = gFindPrev(bt, fkey);

object  bt;
object  *fkey;
object  val;
```

This method is used to retrieve a key / value pair which comes immediately before the key pointed to by `fkey`. If `fkey` points to `NULL` than the last key / value pair is retrieved. If none is found `NULL` is returned.

`fkey` will be updated to point to the key associated to the value returned. It may be used on subsequent calls to `gFindPrev`.

Example:

```
object  fkey = NULL;
object  val;

while (val = gFindPrev(bt, &fkey))
     do whatever;
```

See also: `AddValue::BTree`, `FindNext::BTree`

`SetFunction::BTree` [SetFunction]

```
f = gSetFunction(bt, fun);

object  bt;
int     (*fun)(object, object);
object  (*f)();
```

Normally, the `BTree` class uses the generic function `gCompare` for all its key comparisons. With `gSetFunction` you may override this behavior and specify an alternative generic (or regular C) function to call. See `Compare::String` for details on the semantics of that function.

`gSetFunction` returns the previous compare function set.

See also: `AddValue::BTree`, `Compare::String`

`Size::BTree` [Size]

```
n = gSize(bt);

object  bt;
int     n;
```

This method is used to obtain the number of key / value pairs associated with `bt`.

5.6 BTreeNode Class

This class is used by the `BTree` class to store objects in memory. There are no user methods associated with this class.

5.7 Character Class

The `Character` class is used to represent the C language `char` data type as a Dynace object. It is a subclass of the `Number` class. Even though the `Character` class implements most of its own functionality, it is documented as part of the `Number` class because most of the interface is the same for all subclasses of the `Number` class. Differences are documented in this section.

5.7.1 Character Class Methods

The `Character` class has only one class method and it is used to create new instances of itself.

`NewWithChar::Character` [NewWithChar]

```
i = gNewWithChar(Character, c);

int     c;
object  i;
```

This class method creates instances of the Character class. `c` is the initial value of the character being represented. Note that `c` is of type `int`. That is because if you pass a character to a generic function (which uses a variable argument declaration) the C language will automatically promote it to an `int`.

The value returned is a Dynace instance object which represents the character passed.

Note that the default disposal methods are used by this class since there are no special storage allocation requirements.

Example:

```
object  x;

x = gNewWithChar(Character, 'a');
```

See also: `CharValue::Number`, `ChangeCharValue::Number`, `Dispose::Object`

5.7.2 Character Instance Methods

The instance methods associated with the `Character` class provide a means of changing and obtaining the value associated with an instance of the `Character` class. Most of the `Character` class instance methods are documented in the `Number` class because of their common interface with all other subclasses of the `Number` class. The remainder are documented in this section.

`FormatChar::Character` [FormatChar]

```
s = gFormatChar(i);

object  i;
object  s;
```

This method creates an instance of the `String` class which contains the character represented by `i`.

Example:

```
object  x, y;

x = gNewWithChar(Character, 'a');
y = gFormatChar(x);
/*  y is a String object which represents "a"  */
```

See also: `FormatNumber::Number`

5.8 CharacterArray Class

This class, which is a subclass of `NumberArray`, is used to represent arbitrary shaped arrays of the C language `char` data type in an efficient manner. Although this class implements much of its own functionality it is documented within the `NumberArray` and `Array` classes because the interface is shared by all subclasses of the `NumberArray` class.

5.9 Constant Class

This class is used to represent constants. Constants are objects which can never be destroyed or copied. They always retain their value and when compared, they only equal themselves. An example of a constant would be an object which represents a true or false condition such as `T` and `NIL` in lisp.

5.9.1 Constant Class Methods

There are no specific class methods associated with this class. The default instance creation method (`New`) is used.

5.9.2 Constant Instance Methods

The instance methods associated with this class are used to override certain default methods in order to insure the "constant" status of instances of this class.

`Copy::Constant` [Copy]

```
s = gCopy(s)

object  s;
```

This method performs no function and returns the value passed. Its only use is to insure that there is only one copy of any particular instances of the `Constant` class.

`DeepCopy::Constant` [DeepCopy]

```
s = gDeepCopy(s)

object  s;
```

This method performs no function and returns the value passed. Its only use is to insure that there is only one copy of any particular instances of the `Constant` class.

`DeepDispose::Constant` [DeepDispose]

```
r = gDeepDispose(s)

object  s;
object  r;
```

This method performs no function and returns the value passed. Its only use is to insure the undelete-ability of instances of the `Constant` class.

`Dispose::Constant` [Dispose]

```
r = gDispose(s)

object  s;
object  r;
```

This method performs no function and returns the value passed. Its only use is to insure the undelete-ability of instances of the `Constant` class.

`Equal::Constant` [Equal]

```
r = gEqual(s, t)

object  s;
object  t;
int     r;
```

This method checks for the equality of two objects (`s` and `t`). If they are the same constant a 1 is returned and 0 otherwise.

5.10 Date Class

The `Date` class is a subclass of the `LongInteger` class and used to represent dates. The equivalent C language representation would be a long integer in the form `YYYYMMDD` so July 23, 1993 would be represented as 19930723L.

5.10.1 Date Class Methods

The `Date` class has only one class method. The `New` method is inherited from the `LongInteger` class.

`CalToJul::Date` [CalToJul]

```
i = gCalToJul(Date, dt);

long  i;
long  dt;
```

This class method returns a julian date calculated from the input calendar date.

Example:

```
long  x;

x = gCalToJul(Date, 20000605L);  /* x = julian date 730276L */
```

See also: `JulToCal::Date`, `Julian::Date`

`FixInvalidDate::Date` [FixInvalidDate]

```
i = gFixInvalidDate(Date, mode);

int  i;
int  mode;
```

This class method sets the method of handling invalid dates. The previous mode is returned.

The list of valid modes is as follows:

```
0   Does nothing.  Leaves the date invalid.
1   Changes the date to be the last day of the month.
2   Changes the date to be the corresponding day of the
    next month.
```

Example:

```
int  x;

x = gFixInvalidDate(Date, 1);   /*  x = the previous mode */
```

`JulToCal::Date` [JulToCal]

```
i = gJulToCal(Date, jdt);

long  i;
long  jdt;
```

This class method returns a calendar date calculated from the input julian date.

Example:

```
long  x;

x = gCalToJul(Date, 730276L);
    /*  x = calendar date 20000605L */
```

See also: `CalToJul::Date`, `Julian::Date`

`Today::Date` [Today]

```
i = gToday(Date);

object  i;
```

This class method creates instances of the `Date` class which represents the current date contained within the system.

Example:

```
object  x;

x = gToday(Date);   /*  x = the current date  */
```

See also: `FormatDate::Date`, `Dispose::Object`

5.10.2 Date Instance Methods

A portion of the `Date` class's functionality is obtained through its inheritance of the `LongInteger` class. The remaining functionality is defined by the following specific methods.

`AddDays::Date` [AddDays]

```
i = gAddDays(i, days);

object  i;
long    days;
```

This method is used to add an arbitrary number of days (`days`) to date `i`. Leap years and the correct number of days in each month are accounted for. The value returned is the object passed.

Example:

```
object  dt;

dt = gNewWithLong(Date, 19940130L);
gAddDays(dt, 2L);   /*  dt contains 19940201  */
```

See also: `AddMonths::Date, AddYears::Date`

`AddMonths::Date` [AddMonths]

```
i = gAddMonths(i, m);

object  i;
int     m;
```

This method is used to add an arbitrary number of months (`m`) to date `i`. The correct number of months in each year is accounted for. The value returned is the object passed.

Example:

```
object  dt;

dt = gNewWithLong(Date, 19940104L);
gAddMonths(dt, 2);   /*  dt contains 19940304  */
```

See also: `AddDays::Date, AddYears::Date`

`AddYears::Date` [AddYears]

```
i = gAddYears(i, y);

object  i;
int     y;
```

This method is used to add an arbitrary number of years (`y`) to date `i`. The value returned is the object passed.

Example:

```
object  dt;

dt = gNewWithLong(Date, 19940104L);
gAddYears(dt, 2);   /*  dt contains 19960104  */
```

See also: `AddDays::Date`, `AddMonths::Date`

`ChangeDateValue::Date` [ChangeDateValue]

```
i = gChangeDateValue(i, dt);

object  i;
long    dt;
```

This method is used to change the date associated with an instance of the `Date` class. Notice that this method returns the instance being passed. `dt` is the new date.

Example:

```
object  x;

x = gNewWithLong(Date, 20000101L);
gChangeLongValue(x, 20000605L);
```

See also: `ChangeLongValue::Number`

`DateValue::Date` [DateValue]

```
dt = gDateValue(i);

object  i;
long    dt;
```

This method is used to obtain the `long` value that represents the date associated with an instance the `Date` class. Note that this is one of the few generics which doesn't return a Dynace object. It returns a `long`.

Example:

```
object  x;
long    dt;

x = gNewWithLong(LongInteger, 20000605L);
dt = gDateValue(x);    /*  dt = 20000605L  */
```

See also: `LongValue::Number`

`DayName::Date` [DayName]

```
s = gDayName(i);

object  i;
object  s;
```

This method returns an instance of the `String` class which represents the day of the week associated with instance `i`.

Example:

```
object  s, dt;

dt = gToday(Date);
s = gDayName(dt);   /*  s contains "Monday" (or whatever)  */
```

See also: `FormatDate::Date`, `MonthName::Date`

`Difference::Date` [Difference]

```
r = gDifference(i, dt);

object  i;
object  dt;
long    r;
```

This method is used to obtain the difference, in days, between two date objects. Leap years and the correct number of days per month is accounted for.

Example:

```
object  dt1, dt2;
long    r;

dt1 = gNewWithLong(Date, 19940104L);
dt2 = gNewWithLong(Date, 19940204L);
r = gDifference(dt2, dt1);    /*  r = 31L  */
```

`FormatDate::Date` [FormatDate]

```
s = gFormatDate(i, fmt);

object  i;
char    *fmt;
object  s;
```

This method returns an instance of the `String` class which is a formatted representation of the date associated with instance `i`. `fmt` is the format specification used to determine the resulting `String` object. Each character in `fmt` is sequentially processed and except for the character '%', they are simply copied to the resulting `String` object. Whenever this method encounters the '%' character, the character following is used to determine what aspect and format of the date represented by `i` should be added to the resultant `String` object. It works much like the standard C `printf`. The following table indicates the available formatting option characters:

```
%       the % character
w       the day of the week (Friday)
M       the month name (June)
m       short month name (Feb)
d       day (8)
D       day to two places (08)
y       short year to two places (04)
Y       full year (1994)
s       day suffix (th)
n       month number (6)
N       month number to two places (06)
t       current system time (10:05 AM)
```

Example:

```
object  s, dt;

dt = gToday(Date);
s = gFormatDate(dt, "%w the %d%s of %M");
/*  s contains "Monday the 1st of March" (or whatever)  */
```

See also: `MonthName::Date`, `DayName::Date`

`Julian::Date` [Julian]

```
jdt = gJulian(i);

object  i;
object  jdt;
```

This method returns a Julian date representation of the date associated with instance `i`.

Example:

```
object  jdt, dt;

dt = gToday(Date);
jdt = gMonthName(dt);   /*  s contains a Julian Date  */
```

See also: `CalToJul::Date`, `JulToCal::Date`

`MonthName::Date` [MonthName]

```
s = gMonthName(i);

object  i;
object  s;
```

This method returns an instance of the `String` class which represents the month of the year associated with instance `i`.

Example:

```
object  s, dt;

dt = gToday(Date);
s = gMonthName(dt);   /*  s contains "June" (or whatever)  */
```

See also: `FormatDate::Date`, `DayName::Date`

`StringRepValue::Date` [StringRepValue]

```
s = gStringRepValue(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the value associated with `i`. This is often used to print or display the value. It is also used by `PrintValue::Object` and indirectly by `Print::Object` (two methods useful during the debugging phase of a project) in order to directly print an object's value.

Example:

```
object  x;
object  s;

x = gNewWithLong(Date, 20000605L);
s = gStringRepValue(x);      /*  s represents "6/05/00"   */
```

See also: `PrintValue::Object`, `Print::Object`, `FormatDate::Date`

`ValidDate::Date` [ValidDate]

```
r = gValidDate(i);

object  i;
int     r;
```

This method is used to determine the validity of a date. If `i` is a valid date 1 is returned and 0 otherwise. Leap years and the number of days in each month are all accounted for.

Example:

```
object  dt;
int     r;

dt = gNewWithLong(Date, 19940104L);
r = gValidDate(dt);   /*  r = 1  */
dt = gNewWithLong(Date, 19941404L);
r = gValidDate(dt);   /*  r = 0  */
dt = gNewWithLong(Date, 19940230L);
r = gValidDate(dt);   /*  r = 0  */
```

5.11 DateTime Class

The `DateTime` class uses multiple inheritance to inherit from both the `Date` and the `Time` classes and is used to represent precise moments in time.

5.11.1 DateTime Class Methods

The `DateTime` class has two class methods to create instances.

`NewDateTime::DateTime` [NewDateTime]

```
i = gNewDateTime(Time, dt, tm);

long    dt, tm;
object  i;
```

This class method creates instances of the `DateTime` class. `dt` represents the initial date value represented and `tm` represents the initial time value.

Example:

```
object  x;

x = gNewWithLong(DateTime, 19990402L, 132345678L);
/*  x = 4/02/1999 1:23:45.678 pm  */
```

See also: `Now::DateTime`, `Dispose::Object`

`Now::DateTime` [Now]

```
i = gNow(DateTime);

object  i;
```

This class method creates instances of the `DateTime` class which represents the current time contained within the system.

Example:

```
object  x;

x = gNow(DateTime);   /*  x = the current date and time  */
```

See also: `NewDateTime::DateTime`, `Dispose::Object`

5.11.2 DateTime Instance Methods

A portion of the `DateTime` class's functionality is obtained through its inheritance of the `Date` class and of the `Time`. The remaining functionality is defined by the following specific methods.

`AddHours::DateTime` [AddHours]

```
i = gAddHours(i, hours);

object  i;
long    hours;
```

This method is used to add an arbitrary number of hours (`hours`) to date and time `i`. Adding an hour that goes past midnight will cause the time to cycle around to morning and add a day. Leap years and the correct number of days in each month are accounted for. The value returned is the object passed.

Example:

```
object  d;

d = gNewDateTime(DateTime, 19990402L, 234500000L);
gAddHours(d, 2L);
/*  d contains 19990403L, 14500000L
    (4/03/1999 1:45:00.000 am) */
```

See also: `AddMilliseconds::DateTime`, `AddMinutes::DateTime`, `AddSeconds::DateTime`

`AddMilliseconds::DateTime` [AddMilliseconds]

```
i = gAddMilliseconds(i, m);

object  i;
long    m;
```

This method is used to add an arbitrary number of milliseconds (`m`) to date and time `i`. Adding a millisecond that goes past midnight will cause the time to cycle around to morning and add a day. Leap years and the correct number of days in each month are accounted for. The value returned is the object passed.

Example:

```
object  d;

d = gNewDateTime(DateTime, 19990402L, 234500000L);
gAddMilliseconds(d, 20L);
/*  d contains 19990402L, 234500000L
    (4/02/1999 11:45:00.020 pm) */
```

See also: `AddHours::DateTime`, `AddMinutes::DateTime`, `AddSeconds::DateTime`

`AddMinutes::DateTime` [AddMinutes]

```
i = gAddMinutes(i, m);

object  i;
long    m;
```

This method is used to add an arbitrary number of minutes (`m`) to date and time `i`. Adding a minute that goes past midnight will cause the time to cycle around to morning and add a day. Leap years and the correct number of days in each month are accounted for. The value returned is the object passed.

Example:

```
object  d;

d = gNewDateTime(DateTime, 19990402L, 234500000L);
gAddMinutes(d, 20L);
/*  d contains 19990403L, 500000L
    (4/03/1999 1:05:00.000 am) */
```

See also: `AddHours::DateTime`, `AddMilliseconds::DateTime`, `AddSeconds::DateTime`

`AddSeconds::DateTime` [AddSeconds]

```
i = gAddSeconds(i, s);

object  i;
long    s;
```

This method is used to add an arbitrary number of seconds (`s`) to date and time `i`. Adding a second that goes past midnight will cause the time to cycle around to

morning and add a day. Leap years and the correct number of days in each month are accounted for. The value returned is the object passed.

Example:

```
object  d;

d = gNewDateTime(DateTime, 19990402L, 234500000L);
gAddSeconds(d, 20L);
/*  d contains 19990402L, 234520000L
    (4/02/1999 11:45:20.000 pm) */
```

See also: `AddHours::DateTime`, `AddMilliseconds::DateTime`, `AddMinutes::DateTime`

`ChangeDateTimeValues::DateTime` [ChangeDateTimeValues]

```
i = gChangeDateTimeValues(i, dt, tm);

object  i;
long    dt, tm;
```

This method is used to change the date and time values associated with an instance of the `DateTime` class. Notice that this method returns the instance being passed. `dt` is the new date value and `tm` is the new time value and.

Example:

```
object  d;

d = gNewDateTime(DateTime, 19990402L, 234500000L);
gChangeDateTimeValues(d, 19981231L, 44500000L);
/*  d contains 12/31/1998 4:45:00.000 am  */
```

See also: `ChangeLongValue::DateTime`, `NewDateTime::DateTime`

`ChangeLongValue::DateTime` [ChangeLongValue]

```
i = gChangeLongValue(i, val);

object  i;
long    val;
```

This method is used to change the date value associated with an instance of the `DateTime` class. Notice that this method returns the instance being passed. `val` is the new date value. The time value associated with the instance of the `DateTime` class is reset to 0L (midnight).

Example:

```
object  d;

d = gNewDateTime(DateTime, 19990402L, 234500000L);
gChangeLongValue(tm, 19991231L);
/*  d contains 12/31/1999 12:00:00.000 am  */
```

See also: `ChangeDateTimeValues::DateTime`, `NewDateTime::DateTime`

`Compare::DateTime` [Compare]

```
r = gCompare(i, obj);

object  i;
object  obj;
int     r;
```

This method is used by the generic container classes to determine the relationship of the values represented by `i` and `obj`. `r` is -1 if the date / time represented by `i` is less than the date / time represented by `obj`, 1 if the date / time value of `i` is greater than `obj`, and 0 if they are equal.

See also: `Hash::DateTime`

`DateTimeDifference::DateTime` [DateTimeDifference]

```
i = gDateTimeDifference(i, d, dd, td);

object  i, d;
long    *dd, *td;
```

This method is used to obtain the difference, in days and milliseconds, between two time objects. dd is loaded with the difference in days and td is loaded with the difference in milliseconds. Notice that this method returns the instance being passed.

Example:

```
object  d1, d2;
long    dd, td;

d1 = gNewDateTime(DateTime, 19990402L, 234500000L);
d2 = gNewDateTime(DateTime, 19990403L, 234500111L);
gDateTimeDifference(d2, d1, &dd, &td);
/*  dd = 1L, td = 111L  */
```

DateTimeValues::DateTime [DateTimeValues]

```
i = gDateTimeValues(i, dt, tm);

object  i;
long    *dt, *tm;
```

This method is used to obtain the `long` values that represent the date and time associated with an instance the `DateTime` class. Notice that this method returns the instance being passed.

Example:

```
object  x;
long    dt, tm;

x = gNewDateTime(DateTime, 19990402L, 234500000L);
gDateTimeValues(x, &dt, &tm);
/*  dt = 19990402L and tm = 234500000L  */
```

See also: `NewDateTime::DateTime`, `ChangeDateTimeValues::Time`

FormatDateTime::DateTime [FormatDateTime]

```
s = gFormatDateTime(i, dtFmt, tmFmt);

object  i;
char    *dtFmt;
char    *tmFmt;
object  s;
```

This method returns an instance of the `String` class which is a formatted representation of the date and time associated with instance `i`. `dtFmt` is the format specification (see `FormatDate::Date`) used to determine the date portion of the resulting `String` object. `tmFmt` is the format specification (see `FormatTime::Time`) used to determine the time portion of the resulting `String` object.

Example:

```
object  s, dt;

dt = gNow(DateTime);
s = gFormatDateTime(dt, "%w the %d%s of %M at ", "%h:%M %p");
/*  s = "Monday the 1st of March at 9:45 am" (or whatever)  */
```

See also: `FormatDate::Date`, `FormatTime::Time`

`Hash::DateTime` [Hash]

```
val = gHash(i);

object  i;
int     val;
```

This method is used by the generic container classes to obtain hash values for the date / time object. `val` is a hash value between 0 and a large integer value.

See also: `Compare::DateTime`

`StringRepValue::DateTime` [StringRepValue]

```
s = gStringRepValue(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the time associated with `i`. This is often used to print or display the date and time. It is also used by `PrintValue::Object` and indirectly by `Print::Object` (two methods useful during the debugging phase of a project) in order to directly print an object's value.

Example:

```
object  x;
object  s;

x = gNewDateTime(DateTime, 19991231L, 234500000L);
s = gStringRepValue(x);
/*  s represents "1999-12-31 11:45:00.000 pm"   */
```

See also: `PrintValue::Object`, `Print::Object`

`ValidDateTime::DateTime` [ValidDateTime]

```
r = gValidDateTime(i);

object  i;
int     r;
```

This method is used to determine the validity of a date / time. If `i` is a valid date / time 1 is returned and 0 otherwise.

Example:

```
object  d;
int     r;

d = gNewDateTime(DateTime, 19991231L, 234500000L);
r = gValidDateTime(tm);   /*  r = 1  */
d = gNewDateTime(DateTime, 19941404L, 234500000L);
r = gValidDateTime(tm);   /*  r = 0  */
d = gNewDateTime(DateTime, 19991231L, 254500899L);
r = gValidDateTime(tm);   /*  r = 0  */
```

5.12 Dictionary Class

The `Dictionary` class combines the functionality of the `Set` and `ObjectAssociation` classes in order to store a collection of arbitrary key/value pairs. The key may be any unique Dynace object. The `gHash` and `gCompare` generics are used in order to access and compare the objects.

This class is a subclass of the `Set` class and therefore inherits all of its functionality.

See the examples included with the Dynace system for an illustration of the use of the `Set/Dictionary` related classes.

5.12.1 Dictionary Class Methods

There are no class methods for this class. It inherits the ability to create instances of itself through its superclass, `Set`.

5.12.2 Dictionary Instance Methods

The instance methods associated with this class are used to add, retrieve and remove key/value pairs from the `Dictionary`. Note that additional functionality may be obtained through its superclass, `Set`.

`AddValue::Dictionary` [AddValue]

```
r = gAddValue(i, key, value);

object  i;
object  key
object  value;    /*  or NULL   */
object  r;
```

This method is used to add a new key/value pair to the `Dictionary` instance (`i`). If an object with the same key already exists in the `Dictionary` it will be left as is and `AddValue` will return `NULL`. If the key/value objects are added `AddValue` will return the `ObjectAssociation` instance created to represent the key/value pair passed.

`key` may be any Dynace object. The `gHash` and `gCompare` generics are used to find and compare the various objects in the `Dictionary`.

See also: `Find::Dictionary`, `FindValue::Dictionary`, `RemoveObj::Dictionary`

ChangeValueWithObj::Dictionary [ChangeValueWithObj]

```
r = gChangeValueWithObj(i, key, value);

object  i;
object  key;
object  value;    /*  or NULL   */
object  r;
```

This method is used to change the value associated with an existing key/value pair to the `Dictionary` instance (`i`). `key` represents the identity of the pre-existing key/value pair and `value` represents the new value to be associated with the key.

Normally, this method changes the value part of the association and returns the previous value which is not disposed. If `key` doesn't identify a pre-existing association, this method simply returns `NULL`.

See also: `Find::Dictionary`, `FindValue::Dictionary`, `RemoveObj::Dictionary`

DeepDisposeObj::Dictionary [DeepDisposeObj]

```
r = gDeepDisposeObj(i, key);

object  i;
object  key;
object  r;
```

This method is used to remove and dispose of a key/value pair from a `Dictionary`. If found and removed `i` is returned. If `key` is not found `NULL` is returned.

The key, value and `ObjectAssociation` used to bind the two are all deep disposed.

`key` may be any Dynace object. The `gHash` and `gCompare` generics are used to find and compare the various objects in the `Dictionary`.

This method is the same as `DisposeObj::Dictionary`.

See also: `RemoveObj::Dictionary`

Dispose::Dictionary [Dispose]

```
r = gDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to dispose of an entire `Dictionary`. It does not dispose of any of the keys or values but does dispose of all the associations used to represent the pair.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `RemoveObj::Dictionary`, `DeepDispose::Set`, `DisposeAllNodes::Dictionary`

`DisposeAllNodes::Dictionary` [DisposeAllNodes]

```
i = gDisposeAllNodes(i);

object  i;
```

This method is used to remove all objects in a `Dictionary` instance without disposing of the instance itself. The objects in the `Dictionary` are simply disassociated from the `Dictionary` instance and are not disposed.

The value returned is always the instance passed.

See also: `DeepDisposeAllNodes::Set`, `Dispose::Dictionary`

`DisposeObj::Dictionary` [DisposeObj]

```
r = gDisposeObj(i, key);

object  i;
object  key;
object  r;
```

This method is used to remove and dispose of a key/value pair from a `Dictionary`. If found and removed `i` is returned. If `key` is not found `NULL` is returned.

The key, value and `ObjectAssociation` used to bind the two are all deep disposed.

`key` may be any Dynace object. The `gHash` and `gCompare` generics are used to find and compare the various objects in the `Dictionary`.

This method is the same as `DeepDisposeObj::Dictionary`.

See also: `RemoveObj::Dictionary`

`Find::Dictionary` [Find]

```
r = gFind(i, key);

object  i;
object  key;
object  r;
```

This method is used to find the instance of the `ObjectAssociation` class which is used to represent the key/value pair stored under `key` in `Dictionary` `i`. If `key` is not found `NULL` is returned.

`key` may be any Dynace object. The `gHash` and `gCompare` generics are used to find and compare the various objects in the `Dictionary`.

See also: `FindValue::Dictionary`

`FindAddValue::Dictionary` [FindAddValue]

```
r = gFindAddValue(i, key, value);

object  i;
object  key;
object  value;    /*  or NULL   */
object  r;
```

This method is used to find and return the instance of the `ObjectAssociation` class used to represent the key/value pair stored under the key `key`. If it is not found a new `ObjectAssociation` will be added and returned which represent the key/value pair representing `key` and `value`.

`key` may be any Dynace object. The `gHash` and `gCompare` generics are used to find and compare the various objects in the `Dictionary`.

See also: `Find::Dictionary`, `FindValue::Dictionary`

`FindValue::Dictionary` [FindValue]

```
r = gFindValue(i, key);

object  i;
object  key;
object  r;
```

This method is used to find the value stored under `key` in `Dictionary` `i`. If `key` is not found `NULL` is returned.

`key` may be any Dynace object. The `gHash` and `gCompare` generics are used to find and compare the various objects in the `Dictionary`.

See also: `Find::Dictionary`

`RemoveObj::Dictionary` [RemoveObj]

```
r = gRemoveObj(i, key);

object  i;
object  key;
object  r;
```

This method is used to remove a key/value pair from a `Dictionary`. If found and removed `i` is returned. If `key` is not found `NULL` is returned.

The key and value are not disposed, however, the `ObjectAssociation` used to bind the two is.

`key` may be any Dynace object. The `gHash` and `gCompare` generics are used to find and compare the various objects in the `Dictionary`.

See also: `DeepDisposeObj::Dictionary`

5.13 DoubleFloat Class

The `DoubleFloat` class is used to represent the C language `double` data type as a Dynace object. It is a subclass of the `Number` class. Even though the `DoubleFloat` class implements most of its own functionality, it is documented as part of the `Number` class because most of the interface is the same for all subclasses of the `Number` class. Differences are documented in this section.

5.13.1 DoubleFloat Class Methods

The `DoubleFloat` class has only one class method and it is used to create new instances of itself.

`NewWithDouble::DoubleFloat` [NewWithDouble]

```
i = gNewWithDouble(DoubleFloat, val);

double  val;
object  i;
```

This class method creates instances of the DoubleFloat class. `val` is the initial value of the number being represented.

The value returned is a Dynace instance object which represents the number passed.

Note that the default disposal methods are used by this class since there are no special storage allocation requirements.

Example:

```
object  pi;

pi = gNewWithDouble(DoubleFloat, 3.14159265358979);
```

See also: `DoubleValue::Number`, `Dispose::Object`

5.13.2 DoubleFloat Instance Methods

The instance methods associated with the `DoubleFloat` class provide a means of changing and obtaining the value associated with an instance of the `DoubleFloat` class. Most of the `DoubleFloat` class instance methods are documented in the `Number` class because of their common interface with all other subclasses of the `Number` class. The remainder are documented in this section.

`Round::DoubleFloat` [Round]

```
i = gRound(i, pl);

object  i;
int     pl;
```

This method is used to round the value represented by `i` to `pl` decimal places. The object returned is the rounded object passed.

Example:

```
object  pi;

pi = gNewWithDouble(DoubleFloat, 3.14159265358979);
gRound(pi, 4);     /*  pi = 3.1416  */
```

See also: `DoubleValue::Number`, `Truncate::DoubleFloat`

`Truncate::DoubleFloat` [Truncate]

```
i = gTruncate(i, pl);

object  i;
int     pl;
```

This method is used to truncate the value represented by `i` to `pl` decimal places. The object returned is the truncated object passed.

Example:

```
object  pi;

pi = gNewWithDouble(DoubleFloat, 3.14159265358979);
gTruncate(pi, 4);     /*  pi = 3.1415  */
```

See also: `DoubleValue::Number`, `Round::DoubleFloat`

5.14 DoubleFloatArray Class

This class, which is a subclass of `NumberArray`, is used to represent arbitrary shaped arrays of the C language `double` data type in an efficient manner. Although this class implements much of its own functionality it is documented within the `NumberArray` and `Array` classes because the interface is shared by all subclasses of the `NumberArray` class.

5.15 File Class

This class is used to encapsulate the standard C stream IO facility. It is a subclass of `Stream` and enables access to these routines through an interface which is common to all subclasses of `Stream`.

Although this class implements most of its own functionality, it is documented as part of the `Stream` class because most of the interface is the same for all subclasses of `Stream`. Differences are documented in this section.

5.15.1 File Class Methods

The class methods associated with this class are methods used to open or create files.

`Flush::File` [Flush]

```
i = gFlush(File);

int  i;
```

This class method is used to flush all files opened with the stream IO system to disk. It returns the number of opened streams.

Example:

```
gFlush(File);
```

See also: `instance Flush::File`

`OpenFile::File` [OpenFile]

```
i = gOpenFile(File, name, mode);

char    *name;  /*  file name  */
char    *mode;  /*  file mode  */
object  i;
```

This class method is used to open or create a normal file using the C stream IO facility. The `name` and `mode` parameters correspond to the local C library function `fopen`. The local library manuals can more completely describe those arguments. The value returned is an object which refers to the opened file. If the file cannot be opened a `NULL` will be returned.

Example:

```
object  f;

f = gOpenFile(File, "myfile", "r");
```

See also: `OpenTempFile::File, Dispose::File, Read::Stream`

`OpenTempFile::File` [OpenTempFile]

```
i = gOpenTempFile(File);

object  i;
```

This class method is used to create a tempoary file using the C stream IO facility. The location of the file will be dictated by the `TMP` or `TEMP` environment variables (in that order). If those variables are not defined then the current directory is used. The file is created with the `"w+"` binary mode.

The value returned is an object which refers to the opened file. Since this is a temporary file, disposing of this object will also cause the file to be removed. If the file cannot be opened a `NULL` will be returned.

Example:

```
object  f;

f = gOpenTempFile(File);
```

See also: `SetTempSubDir::File, OpenFile::File, Dispose::File, Read::Stream`

`SetTempSubDir::File` [SetTempSubDir]

```
i = gSetTempSubDir(File, dir);

char  *dir;
object  i;
```

This class method is used to specify a path for the system to use to create temporary files (using `gOpenTempFile`. The given directory is appended to the normal system temp path. For example, if the system path is '`/tmp`' and `dir` is passed `"App"` then the path used will be '`/tmp/App`'.

The value returned is `File`.

Example:

```
gSetTempSubDir(File, "Application");
```

See also: `OpenTempFile::File`

5.15.2 File Instance Methods

The instance methods associated with this class are used to read and write data to the file represented by the instance.

Although this class implements most of its own functionality, it is documented as part of the `Stream` class because most of the interface is the same for all subclasses of `Stream`. Differences are documented in this section.

`DeepDispose::File` [DeepDispose]

```
r = gDeepDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to close the file associated with the instance and dispose of the instance. It performs the same function as `Dispose::File`.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `Dispose::File, OpenFile::File`

`Dispose::File` [Dispose]

```
r = gDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to close the file associated with the instance and dispose of the instance. It performs the same function as `DeepDispose::File`.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `DeepDispose::File, OpenFile::File`

`Flush::File` [Flush]

```
i = gFlush(file);

object file;
int  i;
```

This method is used to flush the stream associated with file to disk. It returns zero on success.

Example:

```
gFlush(file);
```

See also: `class Flush::File`

`Name::File` [Name]

```
n = gName(i);

object  i;
char    *n;  /*  the file name  */
```

This method is used to obtain the name of the file represented by `i`.

`PointerValue::File` [PointerValue]

```
fp = gPointerValue(i);

object  i;
FILE    *fp;
```

This method is used to obtain the file pointer associated with `i`.

5.16 FindFile Class

The `FindFile` class is designed to provide a portable means of enumerating over the file names contained within a particular file system directory. Currently, this class has only been ported to the DOS and all Windows environments. There is no current support for Unix or Macintosh environments.

5.16.1 FindFile Class Methods

The only class method used in this class is one to create instances of itself.

`NewFindFile::FindFile` [NewFindFile]

```
i = gNewFindFile(FindFile, name, attr);

char    *name;
int     attr;
object  i;
```

This class method creates instances of the `FindFile` class. Each such instance is initialized to enable the instance to enumerate through all the file names which meet a particular set of parameters.

The `name` parameter is used to specify the pattern of the file name to be searched for. This pattern may or may not contain a relative or absolute directory path but *must* contain some wildcard characters (such as "?" and "*").

The `attr` parameter is used to narrow the search to file names which contain certain attributes. The list of available attribute type (which should be or'ed together) is as follows:

```
FF_FILE                 include regular files
FF_DIRECTORY            include directories

FF_READWRITE            include files or directories in which
                        you have read/write permission
FF_READONLY             include files or directories in which
                        you only have read permission

FF_ARCHIVE_ONLY         don't include files/directories which
                        don't have their archive bit set
FF_HIDDEN               include hidden files
FF_SYSTEM               include system files
```

The new instance is returned.

Example:

```
object  ff;
char    *file;

ff = gNewFindFile(FindFile, "*.c", FF_FILE | FF_READWRITE);
while (file = gNextFile(ff))  {
        do something with file
}
gDispose(ff);
```

See also: `NextFile::FindFile`

5.16.2 FindFile Instance Methods

The instance methods associated with this class provide a means for sequentially enumerating through the selected file names and obtaining some degree of information associated with each particular file.

`Attributes::FindFile` [Attributes]

```
attr = gAttributes(ff);

object       ff;
unsigned     attr;
```

This method is used to obtain attribute information relating to the last selected file returned from `NextFile`. The attribute flags are those defined under `NewFindFile`. If no valid file has been returned from `NextFile` this function simply returns 0.

Example:

```
object    ff;
char     *file;
unsigned  attr;

ff = gNewFindFile(FindFile, "*.*",
                  FF_FILE | FF_DIRECTORY | FF_READWRITE);
while (file = gNextFile(ff))  {
        attr = gAttributes(ff);
        if (attr & FF_DIRECTORY)
                handle a directory
        else if (attr & FF_FILE)
                handle a file
}
gDispose(ff);
```

See also: `NewFindFile::FindFile`, `NextFile::FindFile`, `WriteTime::FindFile`

`Length::FindFile` [Length]

```
len = gLength(ff);

object  ff;
long    len;
```

This method is used to obtain the length or size (in bytes) of the file relating to the last selected file returned from `NextFile`. If no valid file has been returned from `NextFile` this function simply returns -1.

Example:

```
object  ff;
char    *file;
long    len;

ff = gNewFindFile(FindFile, "*.c", FF_FILE | FF_READWRITE);
while (file = gNextFile(ff))  {
        len = gLength(ff);
}
gDispose(ff);
```

See also: `Attributes::FindFile`, `NextFile::FindFile`

`Name::FindFile` [Name]

```
nam = gName(ff);

object  ff;
char    *nam;
```

This method is used to obtain the name of the last file found via `NextFile`. It will be the same name as returned from `NextFile`. If no valid file has been returned from `NextFile` this function simply returns `NULL`.

Example:

```
object  ff;
char   *file;

ff = gNewFindFile(FindFile, "*.c", FF_FILE | FF_READWRITE);
while (gNextFile(ff))  {
        file = gName(ff);
}
gDispose(ff);
```

See also: `NextFile::FindFile`

NextFile::FindFile [NextFile]

```
nam = gNextFile(ff);

object  ff;
char   *nam;
```

This method is used to obtain the next file name in the sequential list specified when using `NewFindFile`. It also changes the context associated with the object `ff` such that all other methods operated on `ff` will refer to the file returned in the last call to this method. In spite of the name of this method, it is also used to obtain the first file name in the list, therefore, this method must be called prior to any other method on `ff` in order to obtain the first file context.

This method returns the name of the file associated with the file context this method establishes. If there is no first file or additional files this method returns `NULL`.

See also: `NewFindFirst::FindFile`

WriteTime::FindFile [WriteTime]

```
tim = gWriteTime(ff);

object  ff;
long    tim;
```

This method is used to obtain the date and time the file associated with the `ff` context was last written to. The value returned can be typecast to the common `time_t` type and represents the number of seconds past some date like 1970. If no valid file context was setup via `NextFile`, this method returns -1.

See also: `Attributes::FindFile, NextFile::FindFile`

5.17 FloatArray Class

This class, which is a subclass of `NumberArray`, is used to represent arbitrary shaped arrays of the C language `float` data type in an efficient manner. Although this class implements much of its own functionality it is documented within the `NumberArray` and `Array` classes because the interface is shared by all subclasses of the `NumberArray` class.

5.18 IntegerArray Class

This class, which is a subclass of `NumberArray`, is used to represent arbitrary shaped arrays of the C language `int` data type in an efficient manner. Although this class implements much of its own functionality it is documented within the `NumberArray` and `Array` classes because the interface is shared by all subclasses of the `NumberArray` class.

5.19 IntegerAssociation Class

This class is a subclass of `Association` and is used to represent an association between a C language `int` and an arbitrary Dynace object. It performs all the functions as the `LookupKey` and `ObjectAssociation` classes and is used in conjunction with the `Set` and `Dictionary` classes. The purpose of this class is to provide an efficient mechanism to form a common association.

See the examples included with the Dynace system for an illustration of the use of the `Set`/`Dictionary` related classes.

5.19.1 IntegerAssociation Class Methods

The only class method associated with this class is one needed to create new instances of itself.

`NewWithIntObj::IntegerAssociation` [NewWithIntObj]

```
i = gNewWithIntObj(IntegerAssociation, key, val);

int     key;
object  val;    /*  or NULL   */
object  i;
```

This class method creates instances of the `IntegerAssociation` class. `key` is used to initialize the key associated with the instance created and `val` is used to initialize its associated value.

The new instance is returned.

5.19.2 IntegerAssociation Instance Methods

The instance methods associated with this class are used to obtain, change and print the key and value associated with the instance. Additional functionality, although implemented by this class, is documented in `IntegerAssociation`'s superclass, `Association`. This is done because of the common interface these methods share with all subclasses of `Association`.

`ChangeIntKey::IntegerAssociation` [ChangeIntKey]

```
i = gChangeIntKey(i, key);

object  i;
int     key;
```

This method is used to change the key associated with instance `i`. `key` is the new key. The old key is simply replaced. This method returns `i`.

See also: `IntKey::IntegerAssociation`,
`ChangeValue::IntegerAssociation`

ChangeValue::IntegerAssociation [ChangeValue]

```
old = gChangeValue(i, val);

object  i;
object  val;    /*  or NULL   */
object  old;
```

This method is used to change the value associated with `i` to a value, `val`. The old value is simply no longer associated with the instance. It is not disposed. This method returns the old value being replaced.

See also: `Value::IntegerAssociation`, `ChangeIntKey::IntegerAssociation`

DeepCopy::IntegerAssociation [DeepCopy]

```
c = gDeepCopy(i);

object  i, c;
```

This method is used to create a new instance of the `IntegerAssociation` class which contains a *copy* of the value from the original `IntegerAssociation`. `DeepCopy` is used to create the copy of the value. The new `IntegerAssociation` instance is returned.

See also: `Copy::Object`

DeepDispose::IntegerAssociation [DeepDispose]

```
r = gDeepDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to dispose of a `IntegerAssociation` instance. It also disposes of the value associated with the instance by the use of its `DeepDispose` method.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `Dispose::Object`

`IntKey::IntegerAssociation` [IntKey]

```
key = gIntKey(i);

object  i;
int     key;
```

This method is used to obtain the integer key associated with instance `i`.

See also: `ChangeIntKey::IntegerAssociation`

`StringRepValue::IntegerAssociation` [StringRepValue]

```
s = gStringRepValue(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the value associated with `i`. This is often used to print or display the value. It is also used by `PrintValue::Object` and indirectly by `Print::Object` (two methods useful during the debugging phase of a project) in order to directly print an object's value.

See also: `PrintValue::Object`, `Print::Object`

`Value::IntegerAssociation` [Value]

```
val = gValue(i);

object  i;
object  val;
```

This method is used to obtain the value associated with `i`.

See also: `ChangeValue::IntegerAssociation`,
`IntKey::IntegerAssociation`

5.20 IntegerDictionary Class

This class combines the functionality of the `Set` and `IntegerAssociation` classes in order to store a collection of arbitrary key/value pairs.

The difference between this class and the `Dictionary` class is that this class only accepts C language `int` values as keys. This class provides an efficient and simple to use means of representing a commonly needed collection.

This class is a subclass of the `Set` class and therefore inherits all of its functionality.

See the examples included with the Dynace system for an illustration of the use of the `Set`/`Dictionary` related classes.

5.20.1 IntegerDictionary Class Methods

There are no class methods for this class. It inherits the ability to create instances of itself through its superclass, `Set`.

5.20.2 IntegerDictionary Instance Methods

The instance methods associated with this class are used to add, retrieve and remove key/value pairs from the `Dictionary`. Note that additional functionality may be obtained through its superclass, `Set`.

`AddInt::IntegerDictionary` [AddInt]

```
r = gAddInt(i, key, value);

object  i;
int     key
object  value;
object  r;
```

This method is used to add a new key/value pair to the `Dictionary` instance (`i`). If an object with the same key already exists in the `Dictionary` it will be left as is and `AddInt` will return `NULL`. If the key/value objects are added `AddInt` will return the `IntegerAssociation` instance created to represent the key/value pair passed.

See also: `FindInt::IntegerDictionary`,
`FindValueInt::IntegerDictionary`,
`ChangeValueWithInt::IntegerDictionary`,
`RemoveInt::IntegerDictionary`

`ChangeValueWithInt::IntegerDictionary` [ChangeValueWithInt]

```
r = gChangeValueWithInt(i, key, value);

object  i;
int     key
object  value;
object  r;
```

This method is used to change the value associated with an existing key/value pair to the `IntegerDictionary` instance (`i`). `key` represents the identity of the pre-existing key/value pair and `value` represents the new value to be associated with the key.

Normally, this method changes the value part of the association and returns the previous value which is not disposed. If `key` doesn't identify a pre-existing association, this method simply returns `NULL`.

See also: `FindInt::IntegerDictionary`,
`FindValueInt::IntegerDictionary`,
`RemoveInt::IntegerDictionary`

`DeepDisposeInt::IntegerDictionary` [DeepDisposeInt]

```
r = gDeepDisposeInt(i, key);

object  i;
int     key;
object  r;
```

This method is used to remove and dispose of a key/value pair from a `Dictionary`. If found and removed `i` is returned. If `key` is not found `NULL` is returned.

The value and `IntegerAssociation` used to bind the two are all deep disposed.

This method is the same as `DisposeInt::IntegerDictionary`.

See also: `RemoveInt::IntegerDictionary`

`Dispose::IntegerDictionary` [Dispose]

```
r = gDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to dispose of an entire `Dictionary`. It does not dispose of any of the values but does dispose of all the associations used to represent the pair.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `RemoveInt::IntegerDictionary`, `DeepDispose::Set`, `DisposeAllNodes::IntegerDictionary`

`DisposeAllNodes::IntegerDictionary` [DisposeAllNodes]

```
i = gDisposeAllNodes(i);

object  i;
```

This method is used to remove all objects in an `IntegerDictionary` instance without disposing of the instance itself. The objects in the `IntegerDictionary` are simply disassociated from the `IntegerDictionary` instance and are not disposed.

The value returned is always the instance passed.

See also: `DeepDisposeAllNodes::Set`, `Dispose::IntegerDictionary`

`DisposeInt::IntegerDictionary` [DisposeInt]

```
r = gDisposeInt(i, key);

object  i;
int     key;
object  r;
```

This method is used to remove and dispose of a key/value pair from a `Dictionary`. If found and removed `i` is returned. If `key` is not found `NULL` is returned.

The value and `IntegerAssociation` used to bind the two are all deep disposed.

This method is the same as `DeepDisposeInt::IntegerDictionary`.

See also: `RemoveInt::IntegerDictionary`

`FindInt::IntegerDictionary` [FindInt]

```
r = gFindInt(i, key);

object  i;
int     key;
object  r;
```

This method is used to find the instance of the `IntegerAssociation` class which is used to represent the key/value pair stored under `key` in `Dictionary i`. If `key` is not found `NULL` is returned.

See also: `FindValueInt::IntegerDictionary`

`FindAddInt::IntegerDictionary` [FindAddInt]

```
r = gFindAddInt(i, key, value);

object  i;
int     key;
object  value;
object  r;
```

This method is used to find and return the instance of the `IntegerAssociation` class used to represent the key/value pair stored under the key `key`. If it is not found a new `IntegerAssociation` will be added and returned which represent the key/value pair representing `key` and `value`.

See also: `FindInt::IntegerDictionary`,
`FindValueInt::IntegerDictionary`

`FindValueInt::IntegerDictionary` [FindValueInt]

```
r = gFindValueInt(i, key);

object  i;
int     key;
object  r;
```

This method is used to find the value stored under `key` in `Dictionary i`. If `key` is not found `NULL` is returned.

See also: `FindInt::IntegerDictionary`

`RemoveInt::IntegerDictionary` [RemoveInt]

```
r = gRemoveInt(i, key);

object  i;
int     key;
object  r;
```

This method is used to remove a key/value pair from a `Dictionary`. If found and removed `i` is returned. If `key` is not found `NULL` is returned.

The value is not disposed, however, the `IntegerAssociation` used to bind the two is.

See also: `DeepDisposeInt::IntegerDictionary`

5.21 Link Class

The `Link` class is used to represent a single link in a doubly linked list. It is normally used in conjunction with the `LinkList` class in order to manage arbitrary objects on a doubly linked list data structure.

Since instances of the `Link` class only contain enough information to describe the link, the `Link` class can be thought of as an abstract class. It is never used by itself (instances of it are never created). Instead a new class would be created which would have the `Link` class as a superclass. This new class would describe the data items which would be stored at each link. The `Link` class would keep track of its place in the linked list.

The `Link` class also has a subclass called `LinkValue` which allows a single arbitrary object to be stored at each link.

See the examples included with the Dynace system for an illustration of the use of the doubly linked list related classes.

5.21.1 Link Class Methods

The `Link` class has only a single class method. That method is used to create and initialize new instances of itself.

`New::Link` [New]

```
i = gNew(Link);

object  i;
```

This class method creates instances of the Link class. It is usually only used by subclasses of the `Link` class. The new link will not point to anything. It will just be an empty link.

This method is actually just inherited from `Object` in order to create a null link.

The value returned is the instance object created (the new link).

See also: `Dispose::Link`

5.21.2 Link Instance Methods

The instances methods associated with the `Link` class are used to link, unlink, move and dispose of particular instances of the `Link` class or (more often) one of its subclasses.

`AddAfter::Link` [AddAfter]

```
i = gAddAfter(i, lnk);

object  i;
object  lnk;
```

This method is used to insert a new link (`lnk`) after another link (`i`) on the linked list which link `i` is on. In order to accomplish this it modifies pointers in link `i`, link `lnk` and the link (if any) which is after link `i` in the associated linked list structure.

Link `lnk` must not already be a member of another list.

This method returns its first argument.

See also: `AddBefore::Link`

`AddBefore::Link` [AddBefore]

```
i = gAddBefore(i, lnk);

object  i;
object  lnk;
```

This method is used to insert a new link (`lnk`) before another link (`i`) on the linked list which link `i` is on. In order to accomplish this it modifies pointers in link `i`, link `lnk` and the link (if any) which is prior to link `i` in the associated linked list structure.

Link `lnk` must not already be a member of another list.

This method returns its first argument.

See also: `AddAfter::Link`, `ChangeNext::Link`, `ChangePrevious::Link`

`ChangeNext::Link` [ChangeNext]

```
i = gChangeNext(i, nxt);

object  i;
object  nxt;
```

This method is used to change the next pointer of a particular link represented by instance `i`. `nxt` is the link to be pointed to or `NULL`. This method does not effect the instance pointed to by `nxt`.

This method returns the link passed in its first argument.

See also: `ChangePrevious::Link`, `Next::Link`

`ChangePrevious::Link` [ChangePrevious]

```
i = gChangePrevious(i, prev);

object  i;
object  prev;
```

This method is used to change the previous link pointer of a particular link represented by instance `i`. `prev` is the link to be pointed to or `NULL`. This method does not effect the instance pointed to by `prev`.

This method returns the link passed in its first argument.

See also: `ChangeNext::Link, Previous::Link`

`Copy::Link` [Copy]

```
c = gCopy(i);

object  i, c;
```

This method is used to create a new object which is of the same type as its argument. However, the new link will not point to the same links. It will be uninitialized.

The new object is returned.

See also: `DeepCopy::Link, Copy::Object`

`DeepCopy::Link` [DeepCopy]

This method performs the same function as `Copy::Link`.

`DeepDispose::Link` [DeepDispose]

```
r = gDeepDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to unlink and dispose of a link object. It performs the same function as the `Dispose` method.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `Dispose::Link, Remove::Link`

Dispose::Link [Dispose]

```
r = gDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to unlink and dispose of a link object. It performs the same function as the `DeepDispose` method.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `DeepDispose::Link`, `Remove::Link`

InitLink::Link [InitLink]

```
lnk = gInitLink(lnk, lst, prev, nxt);

object  lnk;
object  lst;
object  prev;
object  nxt;
```

This method is used to initialize a link. `lst` is the list it will be associated with and `prev` and `nxt` are used to initialize the link pointers. Each argument must either be a link or `NULL`.

The link passed is returned.

See also: `Previous::Link`, `Next::Link`

List::Link [List]

```
lst = gList(i);

object  i;
object  lst;
```

This method is used to obtain the list object which link `i` is on. `NULL` will be returned if the link is not on a list.

See also: `Previous::Link`, `Next::Link`, `InitLink::Link`

`MoveAfter::Link` [MoveAfter]

```
i = gMoveAfter(i, lnk);

object  i;
object  lnk;
```

This method is used to unlink an arbitrary (located anywhere on the list) link (`i`) from any list and relink it immediately after link `lnk` which may be on the same or a different list. If link `i` wasn't originally on a list, it will simply be added after link `lnk`.

Link `i` is returned.

See also: `MoveBefore::Link`, `MoveBeginning::Link`

`MoveBefore::Link` [MoveBefore]

```
i = gMoveBefore(i, lnk);

object  i;
object  lnk;
```

This method is used to unlink an arbitrary (located anywhere on the list) link (`i`) from any list and relink it immediately before link `lnk` which may be on the same or a different list. If link `i` wasn't originally on a list, it will simply be added before link `lnk`.

Link `i` is returned.

See also: `MoveAfter::Link`, `MoveEnd::Link`

`MoveBeginning::Link` [MoveBeginning]

```
i = gMoveBeginning(i);

object  i;
```

This method is used to unlink an arbitrary (located anywhere on the list) link and relink it at the beginning of the linked list it's associated with.

The object passed is returned unless the link was not on a list. In that case `NULL` will be returned.

See also: `MoveEnd::Link`, `MoveAfter::Link`

`MoveEnd::Link` [MoveEnd]

```
i = gMoveEnd(i);

object  i;
```

This method is used to unlink an arbitrary (located anywhere on the list) link and relink it at the end of the linked list it's associated with.

The object passed is returned unless the link was not on a list. In that case `NULL` will be returned.

See also: `MoveBeginning::Link`, `MoveBefore::Link`

`Next::Link` [Next]

```
nxt = gNext(i);

object  i;
object  nxt;
```

This method is used to obtain the link following the link represented by `i`. If no next link exists, `NULL` will be returned.

See also: `Previous::Link`, `Nth::Link`, `List::Link`

`Nth::Link` [Nth]

```
nxt = gNth(lnk, idx);

object  lnk;
int     idx;
object  nxt;
```

This method is used to provide a mechanism to index into a linked list in a manner similar to an array. If `idx` is positive this method returns the link which is `idx` links after `lnk`. If `idx` is negative this method returns the link which is `idx` links prior to `lnk`. If `idx` is zero `lnk` is returned, and if `idx` represents a position beyond the end of the list `NULL` is returned.

See also: `Previous::Link`, `Next::Link`, `List::Link`

`Previous::Link` [Previous]

```
prev = gPrevious(i);

object  i;
object  prev;
```

This method is used to obtain the link prior to the link represented by `i`. If no such link exists, `NULL` will be returned.

See also: `Next::Link`, `Nth::Link`, `List::Link`

`Remove::Link` [Remove]

```
i = gRemove(i);

object  i;
```

This method is used to remove a link (`i`) from a linked list. In addition to effecting link `i`, it also effects the pointers associated with the links which link `i` points to in its next and previous pointers (which may be NULL) to make the removal complete.

This method should have no effect on a link which is not on a list.

This method returns the link passed.

See also: `ChangeNext::Link`, `ChangePrevious::Link`

`StringRep::Link` [StringRep]

```
s = gStringRep(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the object `i` and its value. This is often used to print or display a representation of an object. It is also used by by `Print::Object` (a method useful during the debugging phase of a project) in order to directly print an object to a stream.

See also: `Print::Object`, `PrintValue::Object`, `StringRepValue::Link`

`StringRepValue::Link` [StringRepValue]

```
s = gStringRepValue(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the value associated with `i`. This is often used to print or display the value. It is also used by `PrintValue::Object` and indirectly by `Print::Object` (two methods useful during the debugging phase of a project) in order to directly print an object's value.

Example:

```
object  x;
object  s;

x = gNew(Link);
s = gStringRepValue(x);
```

See also: `PrintValue::Object`, `Print::Object`

5.22 LinkList Class

Instances of this class are used to coordinate and handle doubly linked lists as a whole. It is used with instances of the `Link` class (or one of its subclasses) which represent the individual links of the list.

The `Link` class is a superclass of this class. See that class for functionality which is inherited by this class.

See the examples included with the Dynace system for an illustration of the use of the doubly linked list related classes.

5.22.1 LinkList Class Methods

There is only one class method associated with this class and it is used to create new instances of the class.

`New::LinkList` [New]

```
i = gNew(LinkList);

object  i;
```

This class method creates a new and empty linked list object.

The value returned is the instance object created (the new LinkList).

See also: `Dispose::LinkList`

5.22.2 LinkList Instance Methods

The instance methods associated with this class perform functions to add, change, inquire, enumerate and dispose of links associated with the linked list. They also provide information such as how many links are on the list.

`AddFirst::LinkList` [AddFirst]

```
i = gAddFirst(i, lnk);

object  i;
object  lnk;
```

This method is used to add a new link to the beginning of the linked list. `i` is the linked list, `lnk` is the new link (an instance of the `Link` class or one of its subclasses).

The value returned is the linked list passed (`i`).

See also: `AddLast::LinkList, DisposeFirst::LinkList`

AddLast::LinkList [AddLast]

```
i = gAddLast(i, lnk);

object  i;
object  lnk;
```

This method is used to add a new link to the end of the linked list. `i` is the linked list, `lnk` is the new link (an instance of the `Link` class or one of its subclasses).

The value returned is the linked list passed (`i`).

See also: `AddFirst::LinkList`, `DisposeLast::LinkList`

Copy::LinkList [Copy]

```
c = gCopy(i);

object  i, c;
```

This method is used to create a new instance of the `LinkList` class with copies of all the links which were on `i`. These copies are made via `Copy` so any values associated with the link are the same objects as those on `i`.

The new `LinkList` instance is returned.

See also: `DeepCopy::LinkList`, `Copy::Object`

DeepCopy::LinkList [DeepCopy]

```
c = gDeepCopy(i);

object  i, c;
```

This method is used to create a new instance of the `LinkList` class with copies of all the links and their associated values which were on `i`. These copies are made via `DeepCopy` so the values associated with the links are also copied.

The new `LinkList` instance is returned.

See also: `Copy::LinkList`, `Copy::Object`

`DeepDispose::LinkList` [DeepDispose]

```
r = gDeepDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to dispose of the linked list and all its associated links. The `DeepDispose` method is used to dispose of each link on the linked list.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `Dispose::LinkList`, `Dispose::Object`

`DeepDisposeAllNodes::LinkList` [DeepDisposeAllNodes]

```
i = gDeepDisposeAllNodes(i);

object  i;
```

This method is used to dispose of all the links on a linked list without disposing of the linked list object itself. The `DeepDispose` method is used to dispose of the links so all the objects contained on the links will be deep disposed as well.

This method returns the object passed.

See also: `DeepDispose::LinkList`, `DisposeAllNodes::LinkList`

`DeepDisposeFirst::LinkList` [DeepDisposeFirst]

```
i = gDeepDisposeFirst(i);

object  i;
```

This method is used to unlink and dispose of the first link in the linked list `i`. If there is no first link this method simply returns.

The first link is disposed of by executing the `DeepDispose` method on it.

This method returns the linked list passed.

See also: `DeepDisposeLast::LinkList`, `DisposeFirst::LinkList`

DeepDisposeLast::LinkList [DeepDisposeLast]

```
i = gDeepDisposeLast(i);

object  i;
```

This method is used to unlink and dispose of the last link in the linked list `i`. If there are no links this method simply returns.

The last link is disposed of by executing the `DeepDispose` method on it.

This method returns the linked list passed.

See also: `DeepDisposeFirst::LinkList`, `DisposeLast::LinkList`

Dispose::LinkList [Dispose]

```
r = gDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to dispose of the linked list and all its associated links. The `Dispose` method is used to dispose of the links without disposing of the objects contained on the links.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `DeepDispose::LinkList`, `DisposeAllNodes::LinkList`

DisposeAllNodes::LinkList [DisposeAllNodes]

```
i = gDisposeAllNodes(i);

object  i;
```

This method is used to dispose of all the links on a linked list without disposing of the linked list object itself. The `Dispose` method is used to dispose of the links without disposing of the objects contained on the links.

This method returns the object passed.

See also: `Dispose::LinkList`, `DeepDisposeAllNodes::LinkList`

`DisposeFirst::LinkList` [DisposeFirst]

```
i = gDisposeFirst(i);

object  i;
```

This method is used to unlink and dispose of the first link in the linked list `i`. If there is no first link this method simply returns.

The first link is disposed of by executing the `Dispose` method on it.

This method returns the linked list passed.

See also: `DisposeLast::LinkList`, `DeepDispose::LinkList`, `RemoveFirst::LinkList`

`DisposeLast::LinkList` [DisposeLast]

```
i = gDisposeLast(i);

object  i;
```

This method is used to unlink and dispose of the last link in the linked list `i`. If there are no links this method simply returns.

The last link is disposed of by executing the `Dispose` method on it.

This method returns the linked list passed.

See also: `DisposeFirst::LinkList`, `DeepDispose::LinkList`, `RemoveLast::LinkList`

`First::LinkList` [First]

```
lnk = gFirst(i);

object  i;
object  lnk;
```

This method is used to obtain the first link on linked list `i`. If there are no links, this method returns `NULL`.

This method does not modify the linked list or the link returned.

See also: `Last::LinkList`, `RemoveFirst::LinkList`

`IncNelm::LinkList` [IncNelm]

```
i = gIncNelm(i, inc);

object  i;
int     lnc;
```

Each linked list automatically keeps track of the number of links on it. This method is used to change that value by `inc`. It is mainly used internally to keep the number accurate and would not normally be used externally.

`Last::LinkList` [Last]

```
lnk = gLast(i);

object  i;
object  lnk;
```

This method is used to obtain the last link on linked list `i`. If there are no links, this method returns `NULL`.

This method does not modify the linked list or the link returned.

See also: `First::LinkList`, `RemoveLast::LinkList`

`Remove::LinkList` [Remove]

This method has been superseded by `Remove::Link`.

See also: `Dispose::Link`, `RemoveFirst::LinkList`

`RemoveFirst::LinkList` [RemoveFirst]

```
lnk = gRemoveFirst(i);

object  i;
object  lnk;
```

This method is used to unlink the first link from the linked list `i` and return it. The link will no longer be linked on the list. However, the link is not disposed. If there are no links this method simply returns `NULL`.

This method returns the link removed.

See also: `RemoveLast::LinkList`, `DisposeFirst::LinkList`

RemoveLast::LinkList [RemoveLast]

```
lnk = gRemoveLast(i);

object  i;
object  lnk;
```

This method is used to unlink the last link from the linked list `i` and return it. The link will no longer be linked on the list. However, the link is not disposed. If there are no links this method simply returns `NULL`.

This method returns the link removed.

See also: `RemoveFirst::LinkList, DisposeLast::LinkList`

Sequence::LinkList [Sequence]

```
s = gSequence(i);

object  i;
object  s;
```

This method takes an instance of the `LinkList` class (`i`) and returns an instance of the `LinkSequence` class (`s`). The link list represented by `i` is not effected by this operation. The link sequence item, `s`, is used to enumerate through all the links in link list `i`. See the `LinkSequence` class for documentation of other methods needed to use `s`.

Example:

```
object  ll;  /*  linked list    */
object  s;   /*  link sequence  */
object  lnk; /*  one link       */

/*  ll must be initialized previously  */

for (s=gSequence(ll) ; lnk = gNext(s) ; )  {
        /*  do something with link lnk  */
}
```

See also: `Next::LinkSequence`

`Size::LinkList` [Size]

```
sz = gSize(i);

object  i;
int     sz;
```

This method is used to obtain the number of links on the linked list.

`StringRep::LinkList` [StringRep]

```
s = gStringRep(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the object `i` and its value. This is often used to print or display a representation of an object. It is also used by by `Print::Object` (a method useful during the debugging phase of a project) in order to directly print an object to a stream.

This method obtains the values of each link by calling `StringRepValue` on each link.

See also: `Print::Object`, `PrintValue::Object`

5.23 LinkObject Class

The `LinkObject` class is a subclass of the `LinkList` class. Since this class inherits much of its functionality from the `LinkList` and `Link` classes, see those classes for additional functionality. It is used in conjunction with the `LinkValue` class in order to manage arbitrary objects on a doubly linked list data structure.

The difference between this class and the `LinkList` class is that the `LinkList` class deals with instances of the `Link` class (or one of its subclasses) which may contain arbitrary objects and the `LinkObject` class deals with arbitrary objects directly. There is no need to deal with link objects at all. While this simplifies the usage of a linked lists quite a bit it comes at the expense of flexibility. However, since the `LinkList` class is a superclass of the `LinkObject` class you can often use the inherited functionality of the `LinkList` class in conjunction with the `LinkObject` object. It is, therefore, recommended that the `LinkObject` class be used whenever possible to simplify linked list operations

This class (`LinkObject`) inherits much of its functionality from its superclass (`LinkList`), therefore, in addition to the specific documentation on this class please also refer to the documentation of its superclass.

See the examples included with the Dynace system for an illustration of the use of the doubly linked list related classes.

5.23.1 LinkObject Class Methods

The `LinkObject` class has no specific class methods. It inherits all of its abilities from its superclass, `LinkList`.

5.23.2 LinkObject Instance Methods

The instance methods associated with this class perform functions to add, inquire and enumerate objects associated with the linked list. Since this class is a subclass of the `LinkList` class, additional functionality is inherited from it. The `LinkList` class documentation should, therefore, be consulted.

`AddFirst::LinkObject` [AddFirst]

```
i = gAddFirst(i, obj);

object  i;
object  obj;
```

This method is used to add a new link to the beginning of the list, `i`, containing the arbitrary object `obj`. The list object, `i`, is returned.

Example:

```
object  ll, obj;

ll = gNew(LinkObject);
obj = gNewWithInt(ShortInteger, 44);
gAddFirst(ll, obj);
```

See also: `AddLast::LinkObject`, `First::LinkObject`

`AddLast::LinkObject` [AddLast]

```
i = gAddLast(i, obj);

object  i;
object  obj;
```

This method is used to add a new link to the end of the list, `i`, containing the arbitrary object `obj`. The list object, `i`, is returned.

Example:

```
object  ll, obj;

ll = gNew(LinkObject);
obj = gNewWithInt(ShortInteger, 44);
gAddLast(ll, obj);
```

See also: `AddFirst::LinkObject`, `Last::LinkObject`

`First::LinkObject` [First]

```
obj = gFirst(i);

object  i;
object  obj;
```

This method is used to obtain the object stored in the first link of the list `i`. If there are no links on the list this method will return `NULL`. The list, `i`, is not effected by this operation.

Example:

```
object  ll, obj, obj2;

ll = gNew(LinkObject);
obj = gNewWithInt(ShortInteger, 44);
gAddFirst(ll, obj);
obj2 = gFirst(ll);
/* obj and obj2 are the same object  */
```

See also: `AddFirst::LinkObject`, `Last::LinkObject`

`GetValues::LinkObject` [GetValues]

```
ll = vGetValues(ll, ...);

object  ll;
```

This method is used to extract numerous values from a `LinkObject` instance. It does this without modifing the list. Each argument must be a pointer to a variable which is to hold one of the values. The first variable gets the first object on the list and the second gets the second, etc. The argument list must end in `NULL` to signify the end.

If there are more variables than links on the list the remaining variables will be set to `NULL`. If there are more elements on the list than variables the remiaining list elements are ignored.

The `LinkObject` instance passed is returned unaltered.

Example:

```
object  ll, obj1, obj2;
object  obj3, obj4;

obj1 = gNewWithInt(ShortInteger, 44);
obj2 = gNewWithInt(ShortInteger, 66);
ll = vMakeList(LinkObject, obj1, obj2, NULL);
vGetValues(ll, &obj3, &obj4, NULL);
```

See also: `MakeList::LinkObject`, `Nth::LinkObject`

`Last::LinkObject` [Last]

```
obj = gLast(i);

object  i;
object  obj;
```

This method is used to obtain the object stored in the last link of the list `i`. If there are no links on the list this method will return `NULL`. The list, `i`, is not effected by this operation.

Example:

```
object  ll, obj, obj2;

ll = gNew(LinkObject);
obj = gNewWithInt(ShortInteger, 44);
gAddLast(ll, obj);
obj2 = gLast(ll);
/* obj and obj2 are the same object  */
```

See also: `AddLast::LinkObject`, `First::LinkObject`

`MakeList::LinkObject` [MakeList]

```
obj = vMakeList(LinkObject, ...);

object  obj;
```

This method is used to create a new `LinkObject` containing all the objects listed in the method call. For example, if there are five objects listed, the returned `LinkObject` will be a list containing five links, each with an object associated with it.

Since this is a variable argument method you must tell the system when there are no more arguments by using a last argument of `NULL`. `GetValues` can be used to take the list back apart. All the other `LinkObject` methods may be used as well. The value returned is the new `LinkObject` instance.

Example:

```
object  ll, obj1, obj2;

obj1 = gNewWithInt(ShortInteger, 44);
obj2 = gNewWithInt(ShortInteger, 66);
ll = vMakeList(LinkObject, obj1, obj2, NULL);
```

See also: `GetValues::LinkObject`

`Nth::LinkObject` [Nth]

See [Nth::Link], page 229,

Pop::LinkObject [Pop]

```
obj = gPop(lst);

object  lst;
object  obj;
```

This method is used to obtain the first element of a list while removing it from the list at the same time. It works in conjunction with `gPush` to utilize a list as a stack. `obj` is either the first object on the list or `NULL` if the list was empty.

Example:

```
object  ll, obj, obj2;

ll = gNew(LinkObject);
obj = gNewWithInt(ShortInteger, 44);
gPush(ll, obj);
obj2 = gPop(ll);
```

See also: `Push::LinkObject`

Push::LinkObject [Push]

```
lst = gPush(lst, obj);

object  lst;
object  obj;
```

This method is used to utilize a linklist as a stack. Pushing is the same as adding a new link to the beginning of the list. Object `obj` is added (pushed) on list `lst`. The list object, `lst`, is returned. This method is the same as `gAddFirst`.

Example:

```
object  ll, obj;

ll = gNew(LinkObject);
obj = gNewWithInt(ShortInteger, 44);
gPush(ll, obj);
```

See also: `Pop::LinkObject`

`Sequence::LinkObject` [Sequence]

```
s = gSequence(i);

object  i;
object  s;
```

This method takes an instance of the `LinkObject` class (`i`) and returns an instance of the `LinkObjectSequence` class (`s`). The link list represented by `i` is not effected by this operation. The link sequence item, `s`, is used to enumerate through all the objects in link list `i`. See the `LinkObjectSequence` class for documentation of other methods needed to use `s`.

The difference between the `Sequence` and `SequenceLinks` methods in this class is that the `Sequence` method provides a means to enumerate over the objects in the linked list and the `SequenceLinks` provides a means to enumerate over the links which hold the objects in the list.

Example:

```
object  ll;  /*  linked list           */
object  s;   /*  link object sequence  */
object  obj; /*  an object             */

/*  ll must be initialized previously  */

for (s=gSequence(ll) ; obj = gNext(s) ; )  {
        /*  do something with obj  */
}
```

See also: `Next::LinkObjectSequence, SequenceLinks::LinkObject`

`SequenceLinks::LinkObject` [SequenceLinks]

```
s = gSequenceLinks(i);

object  i;
object  s;
```

This method takes an instance of the `LinkObject` class (`i`) and returns an instance of the `LinkSequence` class (`s`). The link list represented by `i` is not effected by this operation. The link sequence item, `s`, is used to enumerate through all the links in link list `i`. See the `LinkSequence` class for documentation of other methods needed to use `s`.

The difference between the `Sequence` and `SequenceLinks` methods in this class is that the `Sequence` method provides a means to enumerate over the objects in the linked

list and the `SequenceLinks` provides a means to enumerate over the links which hold the objects in the list.

Example:

```
object  ll;  /*  linked list    */
object  s;   /*  link sequence  */
object  lnk; /*  a link         */

/*  ll must be initialized previously  */

for (s=gSequenceLinks(ll) ; lnk = gNext(s) ; )  {
        /*  do something with link lnk  */
}
```

See also: `Next::LinkSequence`, `Sequence::LinkObject`

5.24 LinkObjectSequence Class

This class is a subclass of `Sequence` and is used to provide a mechanism to enumerate through all the objects on the links in a linked list structure without effecting the structure.

The linked list structure is usually represented by instances of the `LinkObject` class. Typically, the `Sequence` instance method of this class is used to create the instances of the `LinkObjectSequence` class.

5.24.1 LinkObjectSequence Class Methods

There is only one class method associated with this class. It is used to create new instances of itself and is normally only called by sequence methods associated with the `LinkObject` class.

`NewWithObj::LinkObjectSequence` [NewWithObj]

```
i = gNewWithObj(LinkObjectSequence, lnk);

object  lnk;
object  i;
```

This class method creates instances of the `LinkObjectSequence` class. It is normally only called by the sequence method associated with the `LinkObject` class. `lnk` is the first link in the list, and `i` is the sequence instance.

See also: `Next::LinkObjectSequence, Sequence::LinkObject`

5.24.2 LinkObjectSequence Instance Methods

This class only has a single instance method which is used to enumerate through object on a list.

`Next::LinkObjectSequence` [Next]

```
obj = gNext(i);

object  i;
object  obj;
```

This method is used to enumerate through the objects contained within the links on a linked list. Each time `Next` is called the following object on the linked list is returned. It does this in a non-destructive way so the associated linked list is not effected.

When `Next` is called after the last object has been returned (or if there are no links on the list) it will return `NULL` and automatically dispose of the sequence object `i`. Therefore, the only time `Dispose` would be needed (if the garbage collector weren't being used) would be if the entire list was not enumerated.

Example:

```
object  ll;  /*  linked object list */
object  s;   /*  link sequence      */
object  obj; /*  an object          */

/*  ll must be initialized previously  */

for (s=gSequence(ll) ; obj = gNext(s) ; )  {
        /*  do something with object obj  */
}
```

See also: `Sequence::LinkObject`

5.25 LinkSequence Class

This class is a subclass of `Sequence` and is used to provide a mechanism to enumerate through all the links in a linked list structure without effecting the structure.

The linked list structure is usually represented by instances of the `LinkList` or `LinkObject` classes. Typically, instance methods of these classes are used to create the instances of the `LinkSequence` class. These methods have names which begin with `Sequence`.

5.25.1 LinkSequence Class Methods

There is only one class method associated with this class. It is used to create new instances of itself and is normally only called by sequence methods associated with the `LinkList` or `LinkObject` classes.

`NewWithObj::LinkSequence` [NewWithObj]

```
i = gNewWithObj(LinkSequence, lnk);

object  lnk;
object  i;
```

This class method creates instances of the `LinkSequence` class. It is normally only called by sequence methods associated with the `LinkList` or `LinkObject` classes. `lnk` is the first link in the list, and `i` is the sequence instance.

See also: `Next::LinkSequence, Sequence::LinkList`

5.25.2 LinkSequence Instance Methods

This class only has a single instance method which is used to enumerate through links on a list.

`Next::LinkSequence` [Next]

```
lnk = gNext(i);

object  i;
object  lnk;
```

This method is used to enumerate through the links on a linked list. Each time `Next` is called the following link on the linked list is returned. It does this in a non-destructive way so the associated linked list is not effected.

When `Next` is called after the last link has been returned (or if there are no links on the list) it will return `NULL` and automatically dispose of the sequence object `i`. Therefore, the only time `Dispose` would be needed (if the garbage collector weren't being used) would be if the entire list was not enumerated.

Example:

```
object  ll;  /*  linked list    */
object  s;   /*  link sequence  */
object  lnk; /*  one link       */

/*  ll must be initialized previously  */

for (s=gSequence(ll) ; lnk = gNext(s) ; )  {
        /*  do something with link lnk  */
}
```

See also: `Sequence::LinkList`, `SequenceLinks::LinkObject`

5.26 LinkValue Class

The `LinkValue` class is a subclass of the `Link` class. It is used in conjunction with the `LinkList` class in order to manage arbitrary objects on a doubly linked list data structure.

Like the `Link` class, it is used to represent a single link on a doubly linked list. However, the `LinkValue` class has the additional capability to store a single arbitrary object at each link. Therefore, the `LinkValue` class may be used without creating a subclass of it.

This class (`LinkValue`) inherits much of its functionality from its superclass (`Link`), therefore, in addition to the specific documentation on this class please also refer to the documentation of its superclass.

See the examples included with the Dynace system for an illustration of the use of the doubly linked list related classes.

5.26.1 LinkValue Class Methods

The `LinkValue` class has only a single class method. That method is used to create and initialize new instances of itself.

`NewWithObj::LinkValue` [NewWithObj]

```
i = gNewWithObj(LinkValue, item);

object  item;
object  i;
```

This class method creates instances of the LinkValue class. `item` is the object to be associated with the new link.

The value returned is the instance object created (the new link).

See also: `Dispose::Link, DeepDispose::LinkValue`

5.26.2 LinkValue Instance Methods

The instances methods associated with the `LinkValue` class are used to link, unlink, move and dispose of particular instances of the `LinkValue` class. The majority of its linking type functionality is inherited from its superclass, `Link`.

`ChangeValue::LinkValue` [ChangeValue]

```
i = gChangeValue(i, obj);

object  i;
object  obj;
```

This method is used to change the object associated with a link. `i` is the instance of the `LinkValue` class, and `obj` is the new object to be associated with it. Any previous association is simply disassociated (not disposed).

This method returns `i`.

See also: `Value::LinkValue, NewWithObj::LinkValue`

`DeepCopy::LinkValue` [DeepCopy]

```
c = gDeepCopy(i);

object  i, c;
```

This method is used to create a new instance of the `LinkValue` class which contains a *copy* of the value associated with the link. `DeepCopy` is used to create the copied value. The new `LinkValue` instance is returned.

Note that the new link returned will not be associated with any list. It will just be an unlinked link.

See also: `Copy::Object`

`DeepDispose::LinkValue` [DeepDispose]

```
r = gDeepDispose(i);

object  i;
object  r;     /*  NULL  */
```

This instance method is used to perform a deep disposal of an instance of the `LinkValue` class. It does this by calling `gDeepDispose` on the item associated with the link and then calling the link's superclass to dispose of the `LinkValue` instance.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `Dispose::Link`

`StringRep::LinkValue` [StringRep]

```
s = gStringRep(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the object `i` and its value. This is often used to print or display a representation of an object. It is also used by by `Print::Object` (a method useful during the debugging phase of a project) in order to directly print an object to a stream.

See also: `Print::Object`, `PrintValue::Object`, `StringRepValue::LinkValue`

`StringRepValue::LinkValue` [StringRepValue]

```
s = gStringRepValue(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the value associated with `i`. This is often used to print or display the value. It is also used by `PrintValue::Object` and indirectly by `Print::Object` (two methods useful during the debugging phase of a project) in order to directly print an object's value.

See also: `PrintValue::Object`, `Print::Object`

`Value::LinkValue` [Value]

```
obj = gValue(i);

object  i;
object  obj;
```

This method is used to obtain the object associated with a link. `i` is the instance of the `LinkValue` class, and `obj` is the object associated with it.

See also: `ChangeValue::LinkValue`, `NewWithObj::LinkValue`

5.27 LongArray Class

This class, which is a subclass of `NumberArray`, is used to represent arbitrary shaped arrays of the C language `long` data type in an efficient manner. Although this class implements much of its own functionality it is documented within the `NumberArray` and `Array` classes because the interface is shared by all subclasses of the `NumberArray` class.

5.28 LongInteger Class

The `LongInteger` class is used to represent the C language `long` data type as a Dynace object. It is a subclass of the `Number` class. Even though the `LongInteger` class implements most of its own functionality, it is documented as part of the `Number` class because most of the interface is the same for all subclasses of the `Number` class. Differences are documented in this section.

5.28.1 LongInteger Class Methods

The `LongInteger` class has only one class method and it is used to create new instances of itself.

`NewWithLong::LongInteger` [NewWithLong]

```
i = gNewWithLong(LongInteger, val);

long    val;
object  i;
```

This class method creates instances of the LongInteger class. `val` is the initial value of the integer being represented.

The value returned is a Dynace instance object which represents the integer passed.

Note that the default disposal methods are used by this class since there are no special storage allocation requirements.

Example:

```
object  x;

x = gNewWithLong(LongInteger, 55L);
```

See also: `LongValue::Number`, `Dispose::Object`

5.28.2 LongInteger Instance Methods

The instance methods associated with the `LongInteger` class provide a means of changing and obtaining the value associated with an instance of the `LongInteger` class. All of the `LongInteger` class instance methods are documented in the `Number` class because of their common interface with all other subclasses of the `Number` class.

5.29 LookupKey Class

The `LookupKey` class, which is a subclass of `Association`, is used to represent an object which contains a key. It is normally used in conjunction with the `ObjectAssociation`, `Set` and `Dictionary` classes in order to manage a group of arbitrary objects.

Since instances of the `LookupKey` class only contain enough information to describe the key, the `LookupKey` class can be thought of as an abstract class. It is never used by itself (instances of it are never created). Instead a new class would be created which would have the `LookupKey` class as a superclass. This new class would describe the data items which would be stored at each link. The `LookupKey` class would keep track of the key to the object. Alternatively, a new class with the same methods as this class could be created and used with the `Set` and `Dictionary` classes.

The `LookupKey` class also has a subclass called `ObjectAssociation` which associates a single arbitrary object with the key.

See the examples included with the Dynace system for an illustration of the use of the `Set`/`Dictionary` related classes.

5.29.1 LookupKey Class Methods

This class has only a single class method which is used to create and initialize instances of itself. Although this class can handle `NULL` key values, their use is discouraged.

`NewWithObj::LookupKey` [NewWithObj]

```
i = gNewWithObj(LookupKey, key);

object  key;
object  i;
```

This class method creates instances of the `LookupKey` class. `key` is used to initialize the key associated with the instance created. The new instance is returned.

5.29.2 LookupKey Instance Methods

The instance methods associated with this class are used access, change, dispose and print the key associated with the instances of this class. Additional functionality, although implemented by this class, is documented in `LookupKey`'s superclass, `Association`. This is done because of the common interface these methods share with all subclasses of `Association`.

`ChangeKey::LookupKey` [ChangeKey]

```
old = gChangeKey(i, key);

object  i;
object  key;
object  old;
```

This method is used to change the key associated with instance `i`. `key` is the new key. The old key is simply replaced, it is not disposed. This method returns the old key object.

See also: `Key::LookupKey`

`DeepCopy::LookupKey` [DeepCopy]

```
c = gDeepCopy(i);

object  i, c;
```

This method is used to create a new instance of the `LookupKey` class which contains a *copy* of the key value from the original `LookupKey`. `DeepCopy` is used to create the copy of the value. The new `LookupKey` instance is returned.

See also: `Copy::Object`

`DeepDispose::LookupKey` [DeepDispose]

```
r = gDeepDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to dispose of instance `i` as well as the key associated with it. It disposes of the key by calling the `DeepDispose` method on it.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `Dispose::Object`

`Key::LookupKey` [Key]

```
key = gKey(i);

object  i;
object  key;
```

This method is used to obtain the key associated with instance `i`.

See also: `ChangeKey::LookupKey`

`StringRepValue::LookupKey` [StringRepValue]

```
s = gStringRepValue(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the value associated with `i`. This is often used to print or display the value. It is also used by `PrintValue::Object` and indirectly by `Print::Object` (two methods useful during the debugging phase of a project) in order to directly print an object's value.

See also: `PrintValue::Object`, `Print::Object`

5.30 LowFile Class

This class is used to encapsulate the standard C low level IO facility. It is a subclass of `Stream` and enables access to these routines through an interface which is common to all subclasses of `Stream`.

Although this class implements most of its own functionality, it is documented as part of the `Stream` class because most of the interface is the same for all subclasses of `Stream`. Differences are documented in this section.

5.30.1 LowFile Class Methods

The one class method associated with this class is the method used to open or create a file.

`OpenLowFile::LowFile` [OpenLowFile]

```
i = gOpenLowFile(LowFile, name, oflag, pmode);

char    *name;  /*  file name  */
int     oflag;
int     pmode;
object  i;
```

This class method us used to open or create a normal file using the C low level IO facility. `name` is the file name to be opened and may also be an instance of the `String` class. The `name`, `oflag` and `pmode` parameters correspond to the local C library function `open`. The local library manuals can more completely describe those arguments. The value returned is an object which refers to the opened file. If the file cannot be opened a `NULL` will be returned.

Example:

```
object  f;

f = gOpenLowFile(LowFile, "myfile", O_CREAT|O_WRONLY,
                 S_IREAD|S_IWRITE);
```

See also: `Dispose::LowFile`, `Read::Stream`

5.30.2 LowFile Instance Methods

The instance methods associated with this class are used to read and write data to the file represented by the instance.

Although this class implements most of its own functionality, it is documented as part of the `Stream` class because most of the interface is the same for all subclasses of `Stream`. Differences are documented in this section.

`DeepDispose::LowFile` [DeepDispose]

```
r = gDeepDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to close the file associated with the instance and dispose of the instance. It performs the same function as `Dispose::LowFile`.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `Dispose::LowFile, OpenLowFile::LowFile`

`Dispose::LowFile` [Dispose]

```
r = gDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to close the file associated with the instance and dispose of the instance. It performs the same function as `DeepDispose::LowFile`.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `DeepDispose::LowFile, OpenLowFile::LowFile`

`FileHandle::LowFile` [FileHandle]

```
h = gFileHandle(i);

object  i;
int     h;
```

This method is used to obtain the file handle associated with `i`.

`Name::LowFile` [Name]

```
n = gName(i);

object  i;
char    *n;  /*  the file name  */
```

This method is used to obtain the name of the file represented by `i`.

5.31 Number Class

The `Number` class is an abstract class used to combine common functionality of the primitive C numeric data types. Although most of the functionality is actually implemented in the subclasses of the `Number` class, they are documented here because of their common interface.

5.31.1 Number Class Methods

The `Number` class has no specific class methods.

5.31.2 Number Instance Methods

Although most of the instance methods are actually implemented in the subclasses of the `Number` class, they are documented here because of their common interface. Regardless of the numeric type being represented there has been every effort made to make a standard set of interfaces to these data elements. Therefore, you are able to change the value associated with an instance of the `ShortInteger` class to a `double` value of 3.141, however, when you retrieve the `double` value you will only get back 3.0. (Of course you wouldn't loose the fractional part if you were dealing with an instance of the `DoubleFloat` class.)

`ChangeCharValue::Number` [ChangeCharValue]

```
i = gChangeCharValue(i, c);

object  i;
int     c;
```

This method is used to change the value associated with an instance of a subclass of the `Number` class. Notice that this method returns the instance being passed. `c` is the new value. If it is not the same type as `i` it will be converted.

Note that `c` is of type `int`. That is because if you pass a character to a generic function (which uses a variable argument declaration) the C language will automatically promote it to an `int`.

Example:

```
object  x;

x = gNewWithChar(Character, 'a');
gChangeCharValue(x, 'b');
```

See also: `CharValue::Number`

`ChangeDoubleValue::Number` [ChangeDoubleValue]

```
i = gChangeDoubleValue(i, val);

object  i;
double  val;
```

This method is used to change the value associated with an instance of a subclass of the `Number` class. Notice that this method returns the instance being passed. `val` is the new value. If it is not the same type as `i` it will be converted.

Example:

```
object  x;

x = gNewWithDouble(DoubleFloat, 3.14);
gChangeDoubleValue(x, 7.56);
```

See also: `NewWithDouble::DoubleFloat, DoubleValue::Number`

`ChangeLongValue::Number` [ChangeLongValue]

```
i = gChangeLongValue(i, val);

object  i;
long    val;
```

This method is used to change the value associated with an instance of a subclass of the `Number` class. Notice that this method returns the instance being passed. `val` is the new value. If it is not the same type as `i` it will be converted.

Example:

```
object  x;

x = gNewWithLong(LongInteger, 55L);
gChangeLongValue(x, 66L);
```

See also: `NewWithLong::LongInteger, LongValue::Number`

`ChangeShortValue::Number` [ChangeShortValue]

```
i = gChangeShortValue(i, s);

object  i;
int     s;
```

This method is used to change the value associated with an instance of a subclass of the `Number` class. Notice that this method returns the instance being passed. `s` is the new value. If it is not the same type as `i` it will be converted.

Note that `s` is of type `int`. That is because if you pass a `short` to a generic function (which uses a variable argument declaration) the C language will automatically promote it to an `int`.

Example:

```
object  x;

x = gNewWithInt(ShortInteger, 55);
gChangeShortValue(x, 66);
```

See also: `NewWithInt::ShortInteger`, `ShortValue::Number`

`ChangeUShortValue::Number` [ChangeUShortValue]

```
i = gChangeUShortValue(i, s);

object  i;
unsigned  s;
```

This method is used to change the value associated with an instance of a subclass of the `Number` class. Notice that this method returns the instance being passed. `s` is the new value. If it is not the same type as `i` it will be converted.

Note that `s` is of type `unsigned`. That is because if you pass a `short` to a generic function (which uses a variable argument declaration) the C language will automatically promote it to an `unsigned`.

Example:

```
object  x;

x = gNewWithUnsigned(UnsignedShortInteger, 55);
gChangeUShortValue(x, 66);
```

See also: `NewWithUnsigned::UnsignedShortInteger`

`ChangeValue::Number` [ChangeValue]

```
i = gChangeValue(i, v);

object  i;
object  v;
```

This method is used to change the value associated with an instance of a subclass of the `Number` class. Notice that this method returns the instance being passed. `v` is an object which represents the new value. If it is not the same type as `i` it will be converted. Note that this conversion does not effect the object `v`, just the value changed in `i`.

Example:

```
object  x, y;

x = gNewWithInt(ShortInteger, 88);
y = gNewWithDouble(DoubleFloat, 3.141);
gChangeValue(x, y);    /*  x is changed to 3  */
```

See also: `ChangeCharValue::Number, ChangeShortValue::Number, etc.`

`CharValue::Number` [CharValue]

```
c = gCharValue(i);

object  i;
char    c;
```

This method is used to obtain the `char` value associated with an instance of a subclass of the `Number` class. Note that this is one of the few generics which doesn't return a Dynace object. It returns a `char`. If the instance does not represent an instance of the `Character` class, the returned value of whatever is represented will be converted to a `char`.

Example:

```
object  x;
char    c;

x = gNewWithChar(Character, 'a');
c = gCharValue(x);
```

See also: `NewWithChar::Character, ChangeValue::Number`

`Compare::Number` [Compare]

```
r = gCompare(i, obj);

object  i;
object  obj;
int     r;
```

This method is used by the generic container classes to determine the relationship of the values represented by `i` and `obj`. `r` is -1 if the value represented by `i` is less than the value represented by `obj`, 1 if the value of `i` is greater than `obj`, and 0 if they are equal. `obj` may be any type of object.

See also: `Hash::Number`

`DoubleValue::Number` [DoubleValue]

```
val = gDoubleValue(i);

object  i;
double  val;
```

This method is used to obtain the `double` value associated with an instance of a subclass of the `Number` class. Note that this is one of the few generics which doesn't return a Dynace object. It returns a `double`. If the instance does not represent an instance of the `DoubleFloat` class, the returned value of whatever is represented will be converted to a `double`.

Example:

```
object  x;
double  val;

x = gNewWithDouble(DoubleFloat, 7.62);
val = gDoubleValue(x);
```

See also: `NewWithDouble::DoubleFloat`, `ChangeValue::Number`

`FormatNumber::Number` [FormatNumber]

```
val = gFormatNumber(i, msk, wth, dp);

object  i;      /*  object to be formatted      */
char    *msk;   /*  format mask                 */
int     wth;    /*  field width                 */
int     dp;     /*  number of decimal places    */
object  val;    /*  String value                */
```

This method is used to convert a number into a nicely formatted string. `i` is the numeric object to be formatted.

`msk` is a character string which tells the formatter specific which formatting options to use. Each character in the string represents a different option. The order is not significant. The following table lists the available options:

```
B   blank if zero
C   add commas every 3 digits to the left of the decimal point
L   left justify
P   put parentheses around negative numbers
Z   zero fill
D   floating dollar sign
U   upper-case letters in conversion (used in bases greater
    than 10)
R   add a percent sign to the end of the number
```

`wth` indicates the width of the field and hence the length of the resultant `String` object. If `wth` <= 0 `FormatNumber` will calculate and use the minimum width which will represent `i`.

`dp` indicates the number of decimal places that should appear in the string. If `dp` is less than the number of decimal places in `i` rounding will occur. However if `dp` < 0 Nfmt will calculate and use the minimum number of decimal places required to fully represent `i`.

The value returned by this method is an instance of the `String` class.

Example:

```
object  x, y;

x = gNewWithDouble(DoubleFloat, 1234567.127);
y = gFormatNumber(x, "C", 0, 2);    /*  y = "1,234,567.13"  */
```

See also: `StringRepValue::Number`

`Hash::Number` [Hash]

```
val = gHash(i);

object  i;
int     val;
```

This method is used by the generic container classes to obtain hash values for the object. `val` is a hash value between 0 and a large integer value.

See also: `Compare::Number`

`LongValue::Number` [LongValue]

```
val = gLongValue(i);

object  i;
long    val;
```

This method is used to obtain the `long` value associated with an instance of a subclass of the `Number` class. Note that this is one of the few generics which doesn't return a Dynace object. It returns a `long`. If the instance does not represent an instance of the `LongInteger` class, the returned value of whatever is represented will be converted to a `long`.

Example:

```
object  x;
long    val;

x = gNewWithLong(LongInteger, 55L);
val = gLongValue(x);
```

See also: `NewWithLong::LongInteger`, `ChangeLongValue::Number`

`ShortValue::Number` [ShortValue]

```
s = gShortValue(i);

object  i;
short   s;
```

This method is used to obtain the `short` value associated with an instance of a subclass of the `Number` class. Note that this is one of the few generics which doesn't return a Dynace object. It returns a `short`. If the instance does not represent an instance of the `ShortInteger` class, the returned value of whatever is represented will be converted to a `short`.

Example:

```
object  x;
short   s;

x = gNewWithInt(ShortInteger, 55);
s = gShortValue(x);
```

See also: `NewWithInt::ShortInteger`, `ChangeShortValue::Number`

`StringRepValue::Number` [StringRepValue]

```
s = gStringRepValue(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the value associated with `i`. This is often used to print or display the value. It is also used by `PrintValue::Object` and indirectly by `Print::Object` (two methods useful during the debugging phase of a project) in order to directly print an object's value.

Example:

```
object  x;
object  s;

x = gNewWithInt(ShortInteger, 55);
s = gStringRepValue(x);      /*  s represents "55"   */
```

See also: `PrintValue::Object`, `Print::Object`, `FormatNumber::Number`

`UnsignedShortValue::Number` [UnsignedShortValue]

```
s = gUnsignedShortValue(i);

object  i;
unsigned short  s;
```

This method is used to obtain the `unsigned short` value associated with an instance of a subclass of the `Number` class. Note that this is one of the few generics which doesn't return a Dynace object. It returns an `unsigned short`. If the instance does not represent an instance of the `UnsignedShortInteger` class, the returned value of whatever is represented will be converted to an `unsigned short`.

Example:

```
object  x;
unsigned short  s;

x = gNewWithUnsigned(UnsignedShortInteger, 55);
s = gUnsignedShortValue(x);
```

See also: `NewWithUnsigned::UnsignedShortInteger`,
`ChangeUShortValue::Number`

5.32 NumberArray Class

The `NumberArray` class, which is a subclass of `Array`, is an abstract class used to combine common functionality of numeric arrays. Although most of the functionality is actually implemented in the subclasses of the `NumberArray` class, they are documented here because of their common interface.

5.32.1 NumberArray Class Methods

The only class method associated with this class is the `New` method which is used to create new instances of numeric arrays. This method is actually implemented by subclasses of `NumberArray`.

`New::NumberArray` [New]

```
ary = vNew(cls, rnk, ...)

object    cls, ary;
unsigned  rnk, ...
```

This class method is used to create a new numeric array. This method is actually implemented by and should only be used with subclasses of the `NumberArray` class. It is documented here because it is the same for all subclasses of the `NumberArray` class.

`cls` would be one of the subclasses of the `NumberArray` class (such as `CharacterArray ShortArray UnsignedShortArray LongArray FloatArray` or `DoubleFloatArray`). This will indicate the type of array to be created.

`rnk` is the number of dimensions the new array should have. The remaining arguments (each of type unsigned) indicates the size of each consecutive dimension. Note that the number of arguments following `rnk` *must* be the same as the value in `rnk`.

`ary` is the new array object created and will be initialized to all zeros.

Example:

```
object  ary;

ary = vNew(DoubleFloatArray, 2, 5, 4);
/*  ary is a 5x4 double matrix  */
```

5.32.2 NumberArray Instance Methods

The instance methods associated with this class are actually implemented by subclasses of `NumberArray`. They are documented here because of their common interface with all subclasses of `NumberArray`.

Regardless of the array type, each subclass of `NumberArray` implements a full complement of methods necessary to deal with all C language data types. In all cases, if the value being stored into an array is not the same as the array, the value being stored will be converted to the correct type prior to the assignment. Also, if the desired return type is not the same

as the array type, the returned value will be converted. The type of the array will remain unchanged. This naturally means that if you store the number 3.141 into an element of a `ShortArray` you will only get 3.0 when attempting to get back a `double` value.

`ChangeCharValue::NumberArray` [ChangeCharValue]

```
ary = vChangeCharValue(ary, val, ...);

object    ary;
int       val;
unsigned  ...
```

This method is used to change the value of one element of array `ary` which should be an instance of a subclass of `NumberArray`.

`val` is the value which the element of the array should be changed to. The reason it is of type `int` is because C promotes `char` to `int` when variable arguments are used.

The arguments after the `val` argument (each an `unsigned`) are used to specify the exact index into each consecutive dimension of the array. The number of arguments after the `val` argument *must* be equal to the number of dimensions (or rank) of array `ary`. See `IndexOrigin::Array` for more information.

If `ary` and `val` do not represent the same types, normal data type conversion will take place during the assignment without changing the type of `ary`.

The value returned is the modified array passed.

Example:

```
object  ary;

ary = vNew(CharacterArray, 2, 5, 4);
vChangeCharValue(ary, 'C', 1, 2);
/*  ary[1][2] <- 'C'  */
```

See also: `ChangeValue::NumberArray`, `ChangeDoubleValue::NumberArray`, `ChangeCharValue::NumberArray` ...

`ChangeDoubleValue::NumberArray` [ChangeDoubleValue]

```
ary = vChangeDoubleValue(ary, val, ...);

object    ary;
double    val;
unsigned  ...
```

This method is used to change the value of one element of array `ary` which should be an instance of a subclass of `NumberArray`. This method should also be used for arrays of type `FloatArray`.

`val` is the value which the element of the array should be changed to.

The arguments after the `val` argument (each an `unsigned`) are used to specify the exact index into each consecutive dimension of the array. The number of arguments after the `val` argument *must* be equal to the number of dimensions (or rank) of array `ary`. See `IndexOrigin::Array` for more information.

If `ary` and `val` do not represent the same types, normal data type conversion will take place during the assignment without changing the type of `ary`.

The value returned is the modified array passed.

Example:

```
object  ary;

ary = vNew(DoubleFloatArray, 2, 5, 4);
vChangeDoubleValue(ary, 3.141, 1, 2);
/*  ary[1][2] <- 3.141  */
```

See also: `ChangeValue::NumberArray`, `ChangeCharValue::NumberArray`, `ChangeShortValue::NumberArray` ...

`ChangeLongValue::NumberArray` [ChangeLongValue]

```
ary = vChangeLongValue(ary, val, ...);

object    ary;
long      val;
unsigned  ...
```

This method is used to change the value of one element of array `ary` which should be an instance of a subclass of `NumberArray`.

`val` is the value which the element of the array should be changed to.

The arguments after the `val` argument (each an `unsigned`) are used to specify the exact index into each consecutive dimension of the array. The number of arguments after the `val` argument *must* be equal to the number of dimensions (or rank) of array `ary`. See `IndexOrigin::Array` for more information.

If `ary` and `val` do not represent the same types, normal data type conversion will take place during the assignment without changing the type of `ary`.

The value returned is the modified array passed.

Example:

```
object  ary;

ary = vNew(LongArray, 2, 5, 4);
vChangeLongValue(ary, 27L, 1, 2);
/*  ary[1][2] <- 27L  */
```

See also: `ChangeValue::NumberArray`, `ChangeDoubleValue::NumberArray`, `ChangeShortValue::NumberArray` ...

`ChangeShortValue::NumberArray` [ChangeShortValue]

```
ary = vChangeShortValue(ary, val, ...);

object    ary;
int       val;
unsigned  ...
```

This method is used to change the value of one element of array `ary` which should be an instance of a subclass of `NumberArray`.

`val` is the value which the element of the array should be changed to. The reason it is of type `int` is because C promotes `short` to `int` when variable arguments are used.

The arguments after the `val` argument (each an `unsigned`) are used to specify the exact index into each consecutive dimension of the array. The number of arguments after the `val` argument *must* be equal to the number of dimensions (or rank) of array `ary`. See `IndexOrigin::Array` for more information.

If `ary` and `val` do not represent the same types, normal data type conversion will take place during the assignment without changing the type of `ary`.

The value returned is the modified array passed.

Example:

```
object  ary;

ary = vNew(ShortArray, 2, 5, 4);
vChangeShortValue(ary, 27, 1, 2);
/*  ary[1][2] <- 27  */
```

See also: `ChangeValue::NumberArray`, `ChangeDoubleValue::NumberArray`, `ChangeShortValue::NumberArray` ...

`ChangeUShortValue::NumberArray` [ChangeUShortValue]

```
ary = vChangeUShortValue(ary, val, ...);

object    ary;
unsigned  val;
unsigned  ...
```

This method is used to change the value of one element of array `ary` which should be an instance of a subclass of `NumberArray`.

`val` is the value which the element of the array should be changed to. The reason it is of type `unsigned` is because C promotes `unsigned short` to `unsigned` when variable arguments are used.

The arguments after the `val` argument (each an `unsigned`) are used to specify the exact index into each consecutive dimension of the array. The number of arguments after the `val` argument *must* be equal to the number of dimensions (or rank) of array `ary`. See `IndexOrigin::Array` for more information.

If `ary` and `val` do not represent the same types, normal data type conversion will take place during the assignment without changing the type of `ary`.

The value returned is the modified array passed.

Example:

```
object  ary;

ary = vNew(UnsignedShortArray, 2, 5, 4);
vChangeUShortValue(ary, 27U, 1, 2);
/*  ary[1][2] <- 27U  */
```

See also: `ChangeValue::NumberArray`, `ChangeDoubleValue::NumberArray`, `ChangeShortValue::NumberArray` ...

`ChangeValue::NumberArray` [ChangeValue]

```
ary = vChangeValue(ary, val, ...);

object    ary;
object    val;
unsigned  ...
```

This method is used to change the value of one element of array `ary` which should be an instance of a subclass of `NumberArray`.

`val` should be an instance of the `Number` class and is the value which the element of the array should be changed to.

The arguments after the `val` argument (each an `unsigned`) are used to specify the exact index into each consecutive dimension of the array. The number of arguments after the `val` argument *must* be equal to the number of dimensions (or rank) of array `ary`. See `IndexOrigin::Array` for more information.

If `ary` and `val` do not represent the same types, normal data type conversion will take place during the assignment without changing the types of `ary` or `val`.

The value returned is the modified array passed.

Example:

```
object  ary;
object  v;

ary = vNew(DoubleFloatArray, 2, 5, 4);
v = gNewWithDouble(Double, 3.141);
vChangeValue(ary, v, 1, 2);
/*  ary[1][2] <- 3.141  */
```

See also: `ChangeDoubleValue::NumberArray`, `ChangeCharValue::NumberArray ...`

`CharValue::NumberArray` [CharValue]

```
v = vCharValue(ary, ...);

object    ary;
unsigned  ...
char      v;
```

This method is used to obtain the `char` value associated with a particular element of an instance of a subclass of the `NumberArray` class.

The arguments after the `ary` argument (each an `unsigned`) are used to specify the exact index into each consecutive dimension of the array. The number of arguments after the `ary` argument *must* be equal to the number of dimensions (or rank) of array `ary`. See `IndexOrigin::Array` for more information.

Note that this is one of the few generics which doesn't return a Dynace object. It returns a `char`. If the instance does not represent an instance of the `CharacterArray` class, the returned value of whatever is represented will be converted to a `char`.

Example:

```
object  ary;
char    v;

ary = vNew(CharacterArray, 2, 5, 4);
v = vCharValue(ary, 1, 2);
/*  v has ary[1][2]  */
```

See also: `ChangeCharValue::NumberArray`

`DoubleValue::NumberArray` [DoubleValue]

```
v = vDoubleValue(ary, ...);

object    ary;
unsigned  ...
double    v;
```

This method is used to obtain the `double` value associated with a particular element of an instance of a subclass of the `NumberArray` class.

The arguments after the `ary` argument (each an `unsigned`) are used to specify the exact index into each consecutive dimension of the array. The number of arguments after the `ary` argument *must* be equal to the number of dimensions (or rank) of array `ary`. See `IndexOrigin::Array` for more information.

Note that this is one of the few generics which doesn't return a Dynace object. It returns a `double`. If the instance does not represent an instance of the `DoubleFloatArray` class, the returned value of whatever is represented will be converted to a `double`.

Example:

```
object  ary;
double  v;

ary = vNew(DoubleFloatArray, 2, 5, 4);
v = vDoubleValue(ary, 1, 2);
/*  v has ary[1][2]  */
```

See also: `ChangeDoubleValue::NumberArray`

`LongValue::NumberArray` [LongValue]

```
v = vLongValue(ary, ...);

object    ary;
unsigned  ...
long      v;
```

This method is used to obtain the `long` value associated with a particular element of an instance of a subclass of the `NumberArray` class.

The arguments after the `ary` argument (each an `unsigned`) are used to specify the exact index into each consecutive dimension of the array. The number of arguments after the `ary` argument *must* be equal to the number of dimensions (or rank) of array `ary`. See `IndexOrigin::Array` for more information.

Note that this is one of the few generics which doesn't return a Dynace object. It returns a `long`. If the instance does not represent an instance of the `LongArray` class, the returned value of whatever is represented will be converted to a `long`.

Example:

```
object  ary;
long    v;

ary = vNew(LongArray, 2, 5, 4);
v = vLongValue(ary, 1, 2);
/*  v has ary[1][2]  */
```

See also: `ChangeLongValue::NumberArray`

`ShortValue::NumberArray` [ShortValue]

```
v = vShortValue(ary, ...);

object    ary;
unsigned  ...
short     v;
```

This method is used to obtain the `short` value associated with a particular element of an instance of a subclass of the `NumberArray` class.

The arguments after the `ary` argument (each an `unsigned`) are used to specify the exact index into each consecutive dimension of the array. The number of arguments after the `ary` argument *must* be equal to the number of dimensions (or rank) of array `ary`. See `IndexOrigin::Array` for more information.

Note that this is one of the few generics which doesn't return a Dynace object. It returns a `short`. If the instance does not represent an instance of the `ShortArray` class, the returned value of whatever is represented will be converted to a `short`.

Example:

```
object  ary;
short   v;

ary = vNew(ShortArray, 2, 5, 4);
v = vShortValue(ary, 1, 2);
/*  v has ary[1][2]  */
```

See also: `ChangeShortValue::NumberArray`

`UnsignedShortValue::NumberArray` [UnsignedShortValue]

```
v = vUnsignedShortValue(ary, ...);

object    ary;
unsigned  ...
unsigned short     v;
```

This method is used to obtain the `unsigned short` value associated with a particular element of an instance of a subclass of the `NumberArray` class.

The arguments after the `ary` argument (each an `unsigned`) are used to specify the exact index into each consecutive dimension of the array. The number of arguments after the `ary` argument *must* be equal to the number of dimensions (or rank) of array `ary`. See `IndexOrigin::Array` for more information.

Note that this is one of the few generics which doesn't return a Dynace object. It returns an `unsigned short`. If the instance does not represent an instance of the `UnsignedShortArray` class, the returned value of whatever is represented will be converted to a `unsigned short`.

Example:

```
object  ary;
unsigned short   v;

ary = vNew(UnsignedShortArray, 2, 5, 4);
v = vUnsignedShortValue(ary, 1, 2);
/*  v has ary[1][2]  */
```

See also: `ChangeUShortValue::NumberArray`

5.33 ObjectArray Class

This class, which is a subclass of `Array`, is used to represent arbitrary shaped arrays of Dynace objects in an efficient manner. By using `ObjectArray`s it is possible to have arrays where each element contains an arbitrary object including another arrays. Therefore it is possible to create nested or general arrays which are nested to arbitrary levels.

Much of the functionality of this class is implemented and documented in the `Array` class. Differences are documented in this section.

5.33.1 ObjectArray Class Methods

The only class method implemented by this class is one used to create new `ObjectArray` instances.

`New::ObjectArray` [New]

```
ary = vNew(ObjectArray, rnk, ...)

unsigned  rnk, ...
object    ary;
```

This class method is used to create a new instance of `ObjectArray`.

`rnk` is the number of dimensions the new array should have. The remaining arguments (each of type unsigned) indicates the size of each consecutive dimension. Note that the number of arguments following `rnk` *must* be the same as the value in `rnk`.

`ary` is the new array object created and will be initialized to all `NULL`'s.

Example:

```
object  ary;

ary = vNew(ObjectArray, 2, 5, 4);
/*  ary is a 5x4 matrix  */
```

5.33.2 ObjectArray Instance Methods

Most instance functionality is obtained and documented in the `Array` class, however, functionality which is particular to this class is documented in this section.

`Value::ObjectArray` [Value]

```
v = vValue(ary, ...);

object    ary;
unsigned  ...
object    v;
```

This method is used to obtain the object (or NULL) value associated with a particular element of an instance of the `ObjectArray` class.

The arguments after the `ary` argument (each an `unsigned`) are used to specify the exact index into each consecutive dimension of the array. The number of arguments after the `ary` argument *must* be equal to the number of dimensions (or rank) of array `ary`. See `IndexOrigin::Array` for more information.

Example:

```
object  ary;
object  v;

ary = vNew(ObjectArray, 2, 5, 4);
v = vValue(ary, 1, 2);
/*  v has ary[1][2]  */
```

See also: `ChangeValue::ObjectArray`

`ChangeValue::ObjectArray` [ChangeValue]

```
ary = vChangeValue(ary, val, ...);

object    ary;
object    val;
unsigned  ...
```

This method is used to change the value of one element of `ObjectArray` `ary`.

`val` is the value which the element of the array should be changed to (or NULL).

The arguments after the `val` argument (each an `unsigned`) are used to specify the exact index into each consecutive dimension of the array. The number of arguments after the `val` argument *must* be equal to the number of dimensions (or rank) of array `ary`. See `IndexOrigin::Array` for more information.

The value returned is the modified array passed.

Example:

```
object  ary, v;

ary = vNew(ObjectArray, 2, 5, 4);
v = gNewWithInt(ShortInteger, 6);
vChangeValue(ary, v, 1, 2);
/*  ary[1][2] <- v  */
```

See also: `Value::ObjectArray`

5.34 ObjectAssociation Class

This class is a subclass of the `LookupKey` class. It is used to associate a value (an arbitrary object) with a key. The documentation on the `LookupKey` should be consulted since much of this class's functionality is inherited from it.

See the examples included with the Dynace system for an illustration of the use of the `Set/Dictionary` related classes.

5.34.1 ObjectAssociation Class Methods

The only class method associated with this class is the one used to create instances of the `ObjectAssociation` class.

`NewWithObjObj::ObjectAssociation` [NewWithObjObj]

```
i = gNewWithObjObj(ObjectAssociation, key, val);

object  key;    /*  or NULL   */
object  val;    /*  or NULL   */
object  i;
```

This class method creates instances of the `ObjectAssociation` class. `key` is used to initialize the key associated with the instance created and `val` is used to initialize its associated value. The new instance is returned.

5.34.2 ObjectAssociation Instance Methods

The instance methods associated with this class are used to obtain, change and print the value associated with its instances. Much additional functionality is achieved through its superclass, `LookupKey`.

`ChangeValue::ObjectAssociation` [ChangeValue]

```
old = gChangeValue(i, val);

object  i;
object  val;    /*  or NULL   */
object  old;
```

This method is used to change the value associated with `ObjectAssociation i` to a value, `val`. The old value is simply no longer associated with the instance. It is not disposed. This method returns the old value being replaced.

See also: `Value::ObjectAssociation, ChangeKey::LookupKey`

DeepCopy::ObjectAssociation [DeepCopy]

```
c = gDeepCopy(i);

object  i, c;
```

This method is used to create a new instance of the `ObjectAssociation` class which contains a *copy* of the key and value from `i`. `DeepCopy` is used to create the copy of the key and value. The new `ObjectAssociation` instance is returned.

See also: `Copy::Object`

DeepDispose::ObjectAssociation [DeepDispose]

```
r = gDeepDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to dispose of the instance passed as well as associated key and value. The key and value are disposed of by using the `DeepDispose` method.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `DeepDispose::LookupKey`

StringRepValue::ObjectAssociation [StringRepValue]

```
s = gStringRepValue(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the value associated with `i`. This is often used to print or display the value. It is also used by `PrintValue::Object` and indirectly by `Print::Object` (two methods useful during the debugging phase of a project) in order to directly print an object's value.

See also: `PrintValue::Object`, `Print::Object`

`Value::ObjectAssociation` [Value]

```
val = gValue(i);

object  i;
object  val;
```

This method is used to obtain the value associated with `i`.

See also: `ChangeValue::ObjectAssociation`, `Key::LookupKey`

5.35 ObjectPool Class

The `ObjectPool` class provides a means to group arbitrary objects in a pool. This pool may later be disposed causing all the objects contained in the pool to be disposed as well. This is useful when running a particular functional module. All the new objects can be added to a pool and then disposed en-mass when the functional unit is complete. This makes object disposal a little easier to organize.

This class can maintain numerous pools as well as global pools.

5.35.1 ObjectPool Class Methods

Class methods for this class are used to create new pools as well as manage global pools.

`New::ObjectPool` [New]

```
i = gNew(ObjectPool);

object  i;
```

This class method creates instances of the ObjectPool class. It is usually only used by subclasses of the `ObjectPool` class. The new link will not point to anything. It will just be an empty link.

This method is actually just inherited from `Object` in order to create a null link.

The value returned is the instance object created (the new link).

See also: `Dispose::ObjectPool`

5.36 Pipe Class

The `Pipe` class is used to provide a mechanism for threads to transfer information via a uni-directional byte stream. If bi-directional communications is needed two pipes may be used.

The `Pipe` class is a subclass of `Stream` and as such shares much of its functionality with other subclasses of the `Stream` class. Although this class implements most of its own functionality, it is documented as part of the `Stream` class because most of the interface is the same for all subclasses of `Stream`. Differences are documented in this section.

See the relevant text for a detailed description of pipes.

5.36.1 Pipe Class Methods

There are only two class methods associated with the `Pipe` class. They are used to create new pipes and find existing ones.

`FindStr::Pipe` [FindStr]

```
p = gFindStr(Pipe, name)

char    *name;
object  p;
```

This method is used to obtain a pipe object from its name. `p` is the pipe object found or `NULL` if not found.

See also: `New::Pipe`

`New::Pipe` [New]

```
p = gNew(Pipe)

object  p;
```

This class method is used to create and initialize a new unnamed pipe with a default buffer size.

The value returned (`p`) is the pipe object created.

See also: `NewWithStrInt::Pipe`, `FindStr::Pipe`, `Dispose::Pipe`

`NewWithStrInt::Pipe` [NewWithStrInt]

```
p = gNewWithStrInt(Pipe, name, bufsiz)

char    *name;
int     bufsiz;
object  p;
```

This class method is used to create and initialize a new pipe.

`name` is the name of the new pipe and may be `NULL` if no name is to be associated with the pipe.

`bufsiz` is the size of the buffer associated with the pipe. The size of the buffer determines how many bytes of information may be buffered before a thread has to held. If set too small the system will become inefficient because of the constant switching between threads. Typical values are 128 or 512. It depends on the quantity and size of the data being transferred.

The value returned (`p`) is the pipe object created. Other threads may obtain the thread object by the use of the `FindStr::Pipe` method.

See also: `New::Pipe`, `FindStr::Pipe`, `Dispose::Pipe`

5.36.2 Pipe Instance Methods

The instance methods associated with this class are used to put bytes on the pipe, take bytes off, dispose of, and obtain information about particular pipes.

Although this class implements most of its own functionality, it is documented as part of the `Stream` class because most of the interface is the same for all subclasses of `Stream`. Differences are documented in this section.

`DeepDispose::Pipe` [DeepDispose]

```
r = gDeepDispose(p)

object  p;
object  r;     /*  NULL  */
```

This method is used to dispose of a pipe instance.
It is the same as `Dispose::Pipe`.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

`Dispose::Pipe` [Dispose]

```
r = gDispose(p)

object  p;
object  r;     /*  NULL  */
```

This method is used to dispose of a pipe instance.
It is the same as `DeepDispose::Pipe`.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

`Length::Pipe` [Length]

```
n = gLength(p)

object  p;
long    n;
```

This method is used to determine the number of bytes currently on the pipe's buffer. Note that this method returns a `long`. This is to retain consistency with `Length::Stream`.

See also: `Room::Pipe`, `Size::Pipe`

`Mode::Pipe` [Mode]

```
p = gMode(p, rblock, wblock)

object  p;
int     rblock;   /*  read block   */
int     wblock;   /*  write block  */
```

This method is used to set the blocking attributes of a particular `Pipe`. A 1 turns on the blocking and a 0 turns it off.

If blocking is turned on when a particular request for reading or writing occurs and there is insufficient bytes or room available the thread requesting the read or write will go into a hold state until another thread can either supply the requested bytes or make the necessary room available. If blocking is not turned on then the particular read or write request will only perform the operation with as many bytes is available at the time. There will be no waiting.

The `rblock` flag effects read requests and the `wblock` effects write requests.

This method returns the `Pipe` instance passed.

See also: `Room::Pipe`, `Length::Pipe`

`Room::Pipe` [Room]

```
n = gRoom(p)

object  p;
int     n;
```

This method is used to determine the number of bytes currently available on the pipe's buffer.

See also: `Length::Pipe`

`Size::Pipe` [Size]

```
n = gSize(p)

object  p;
int     n;
```

This method is used to determine the size of the buffer associated with a `Pipe`.

See also: `Length::Pipe`

5.37 Pointer Class

The `Pointer` class is used to represent the C language `void *` data type as a Dynace object.

5.37.1 Pointer Class Methods

The Pointer class has only one class method and it is used to create new instances of itself.

`NewWithPtr::Pointer` [NewWithPtr]

```
i = gNewWithPtr(Pointer, val);

void    *val;
object  i;
```

This class method creates instances of the Pointer class. `val` is the initial value of the pointer being represented.

The value returned is a Dynace instance object which represents the pointer passed.

Note that the default disposal methods are used by this class since there are no special storage allocation requirements.

Example:

```
object  x;
int     v = 3;

x = gNewWithPtr(Pointer, &v);
```

See also: `PointerValue::Pointer`, `Dispose::Object`

5.37.2 Pointer Instance Methods

The instance methods associated with the Pointer class provide a means of changing and obtaining the value associated with an instance of the Pointer class. Methods are also included to help the generic container classes to quickly access members of this class.

`ChangeValue::Pointer` [ChangeValue]

```
i = gChangeValue(i, val);

object  i;
void    *val;
```

This method is used to change the value associated with an instance of the Pointer class. Notice that this method returns the instance being passed. `val` is the new value.

Example:

```
object  x;
int     v1 = 4, v2 = 5;

x = gNewWithPtr(Pointer, &v1);
gChangeValue(x, &v2);
```

See also: `NewWithPtr::Pointer`, `PointerValue::Pointer`

`Compare::Pointer` [Compare]

```
r = gCompare(i, obj);

object  i;
object  obj;
int     r;
```

This method is used by the generic container classes to determine the equality of the values represented by `i` and `obj`. `r` is -1 if the value represented by `i` is less than the value represented by `obj`, 1 if the value of `i` is greater than `obj`, and 0 if they are equal.

See also: `Hash::Pointer`

`Hash::Pointer` [Hash]

```
val = gHash(i);

object  i;
int     val;
```

This method is used by the generic container classes to obtain hash values for the object. `val` is a hash value between 0 and a large integer value.

See also: `Compare::Pointer`

`PointerValue::Pointer` [PointerValue]

```
val = gPointerValue(i);

object  i;
void    *val;
```

This method is used to obtain the `void *` value associated with an instance of the Pointer class. Note that this is one of the few generics which doesn't return a Dynace object. It returns a void * pointer.

Example:

```
object  x;
int     a = 4, *b;
void    *val;

x = gNewWithPtr(Pointer, &a);
b = (int *) gPointerValue(x);
/*  b now points to a  */
```

See also: `NewWithPtr::Pointer`, `ChangeValue::Pointer`

`StringRepValue::Pointer` [StringRepValue]

```
s = gStringRepValue(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the value associated with `i`. This is often used to print or display the value. It is also used by `PrintValue::Object` and indirectly by `Print::Object` (two methods useful during the debugging phase of a project) in order to directly print an object's value.

Example:

```
object  x;
object  s;

x = gNewWithPtr(Pointer, NULL);
s = gStringRepValue(x);
```

See also: `PrintValue::Object`, `Print::Object`

5.38 PointerArray Class

This class, which is a subclass of `Array`, is used to represent arbitrary shaped arrays of arbitrary C language pointers in an efficient manner.

Much of the functionality of this class is implemented and documented in the `Array` class. Differences are documented in this section.

5.38.1 PointerArray Class Methods

The only class method implemented by this class is one used to create new `PointerArray` instances.

`New::PointerArray` [New]

```
ary = vNew(PointerArray, rnk, ...)

unsigned  rnk, ...
object    ary;
```

This class method is used to create a new instance of `PointerArray`.

`rnk` is the number of dimensions the new array should have. The remaining arguments (each of type unsigned) indicates the size of each consecutive dimension. Note that the number of arguments following `rnk` *must* be the same as the value in `rnk`.

`ary` is the new array object created and will be initialized to all `NULL`'s.

Example:

```
object  ary;

ary = vNew(PointerArray, 2, 5, 4);
/*  ary is a 5x4 matrix  */
```

5.38.2 PointerArray Instance Methods

Most instance functionality is obtained and documented in the `Array` class, however, functionality which is particular to this class is documented in this section.

`PointerValue::PointerArray` [PointerValue]

```
v = vPointerValue(ary, ...);

object    ary;
unsigned  ...
void      *v;
```

This method is used to obtain the pointer value associated with a particular element of an instance of the `PointerArray` class.

The arguments after the `ary` argument (each an `unsigned`) are used to specify the exact index into each consecutive dimension of the array. The number of arguments after the `ary` argument *must* be equal to the number of dimensions (or rank) of array `ary`. See `IndexOrigin::Array` for more information.

Example:

```
object  ary;
void    *v;

ary = vNew(PointerArray, 2, 5, 4);
v = vPointerValue(ary, 1, 2);
/*  v has ary[1][2]  */
```

See also: `ChangeValue::PointerArray`

`ChangeValue::PointerArray` [ChangeValue]

```
ary = vChangeValue(ary, val, ...);

object    ary;
void     *val;
unsigned  ...
```

This method is used to change the value of one element of `PointerArray` `ary`.

`val` is the value which the element of the array should be changed to.

The arguments after the `val` argument (each an `unsigned`) are used to specify the exact index into each consecutive dimension of the array. The number of arguments after the `val` argument *must* be equal to the number of dimensions (or rank) of array `ary`. See `IndexOrigin::Array` for more information.

The value returned is the modified array passed.

Example:

```
object  ary;
int     v = 5;

ary = vNew(PointerArray, 2, 5, 4);
vChangeValue(ary, &v, 1, 2);
/*  ary[1][2] <- &v  */
```

See also: `PointerValue::PointerArray`

5.39 PropertyList Class

Property lists allow arbitrary string name / Dynace object pairs to be associated with any Dynace object. The Dynace objects being associated are associated and retrieved via a unique string name. Any number of these associations may be applied to an instance of a class with this capability.

This class is used as a mixin class. A mixin class is one which is sub-classed with another class (multiple inheritance) in order to give additional functionality to the first class. This class gives any class which inherits from it the ability to set and retrieve arbitrary Dynace objects which are accessed by string names (properties).

For example, if you had a class called `MyClass` and wanted to give it the ability to have property lists you would create a new class called `MyClassWithProperties` (for example) which would inherit from `MyClass` and `PropertyList`. The new class (`MyClassWithProperties`) would contain all the functionality of `MyClass` plus it would be able to handle property lists.

5.39.1 PropertyList Class Methods

There are no class methods implemented by this class.

5.39.2 PropertyList Instance Methods

`DisposePropertyList::PropertyList` [DisposePropertyList]

```
r = gDisposePropertyList(ins);

object  ins;   /*  the instance */
object  r;     /*  ins  */
```

This method is used to remove all properties previously associated with an object. The properties will either be deep disposed or simply disassociated depending on how they were added (see `PropertyPut`). The instance passed is returned.

Example:

```
gDisposePropertyList(ins);
```

See also: `PropertyRemove::PropertyList`

`PropertyGet::PropertyList` [PropertyGet]

```
r = gPropertyGet(ins, pr);

object  ins;   /*  the instance */
char   *pr;    /*  the property name  */
object  r;     /*  value  */
```

This method is used to retrieve a property previously associated with an object. The property is returned otherwise, if no property with that name was added, then `NULL` is returned.

Example:

```
object  val;
val = gPropertyGet(ins, "Color");
```

See also: `PropertyPut::PropertyList`, `PropertyRemove::PropertyList`

`PropertyPut::PropertyList` [PropertyPut]

```
r = gPropertyPut(ins, pr, ad, val);

object  ins;   /*  the instance */
char   *pr;    /*  the property name  */
int     ad;    /*  auto dispose flag  */
object  val;   /*  value to be associated  */
object  r;     /*  val  */
```

This method is used to associate an object (`val`) to instance `ins` under the property name given by `pr`. If the given property has already been used it will be overwritten by `val`.

`ad` is a flag used to determine what happens to `val` once `ins` is disposed or if the property is overwritten. If `ad` is 1 the object will be deep disposed otherwise (if it's zero) it will be simply disassociated with the object and not disposed.

Example:

```
object  ins = gNew(MyClass);
gPropertyPut(ins, "Color", 1, gNewWithStr(String, "Red"));
```

See also: `PropertyGet::PropertyList`, `PropertyRemove::PropertyList`

`PropertyRemove::PropertyList` [PropertyRemove]

```
r = gPropertyRemove(ins, pr);

object  ins;   /*  the instance */
char   *pr;    /*  the property name  */
object  r;     /*  ins  */
```

This method is used to remove a property previously associated with an object. The property will either be deep disposed or simply disassociated depending on how it was added (see `PropertyPut`). The instance passed is returned.

Example:

```
gPropertyRemove(ins, "Color");
```

See also: `DisposePropertyList::PropertyList`, `PropertyRemove::PropertyList`

5.40 Semaphore Class

This class provides a mechanism to coordinate the activity of multiple threads. It provides counting, named and unnamed semaphores. See the manual for a detailed description of semaphores.

See the examples included with the Dynace system for an illustration of the use of the thread / pipe / semaphore related classes.

5.40.1 Semaphore Class Methods

The only two class methods associated with this class provide a means to create new semaphore instances and to find existing semaphores by name.

`FindStr::Semaphore` [FindStr]

```
s = gFindStr(Semaphore, name)

char    *name;
object  s;
```

This method is used to obtain a semaphore object from its associated name. If found it is returned, otherwise `NULL` is returned.

See also: `New::Semaphore`

`New::Semaphore` [New]

```
s = gNew(Semaphore)

object  s;
```

This class method is used to create and initialize a new unnamed semaphore with an initial and maximum count of 1.

The value returned (`s`) is the semaphore object created.

See also: `NewSemaphore::Semaphore`, `FindStr::Semaphore`, `Dispose::Semaphore`

`NewSemaphore::Semaphore` [NewSemaphore]

```
s = gNewSemaphore(Semaphore, name, cnt, max)

char    *name;
int     cnt, max;
object  s;
```

This class method is used to create and initialize a new semaphore.

`name` is the name of the new semaphore and may be `NULL` if no name is to be associated with the semaphore.

`cnt` is the initial count associated with the counting semaphore and is used to indicate the maximum number of threads which may hold this semaphore at one time. A typical value would be 1 and indicates that only a single thread may hold it at a time.

`max` is the maximum value the count of the semaphore is allowed to achieve. If set higher (with `Release`) it will be reset to `max`. This is typically set to the same value as `cnt`.

The value returned (`s`) is the semaphore object created. If not kept it may be obtained later by the use of the `FindStr::Semaphore` method.

See also: `NewSemaphore::Semaphore`, `FindStr::Semaphore`, `Dispose::Semaphore`

5.40.2 Semaphore Instance Methods

The instance methods associated with this class provide a means for locking, releasing, disposing and getting information about semaphore instances.

`Count::Semaphore` [Count]

```
cnt = gCount(s)

object  s;
int     cnt;
```

This method is used to obtain the current lock count associated with semaphore `s`.

See also: `New::Semaphore`, `Name::Semaphore`

`DeepDispose::Semaphore` [DeepDispose]

```
r = gDeepDispose(s)

object  s;
object  r;     /*  NULL  */
```

This method is used to dispose of a semaphore. All locks will be released and the semaphore is disposed. This method is the same as `Dispose::Semaphore`.

See also: `Release::Semaphore`

Dispose::Semaphore [Dispose]

```
r = gDispose(s)

object  s;
object  r;     /*  NULL  */
```

This method is used to dispose of a semaphore. All locks will be released and the semaphore is disposed. This method is the same as `DeepDispose::Semaphore`.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `Release::Semaphore`

Name::Semaphore [Name]

```
nam = gName(s)

object  s;
char    *nam;
```

This method is used to obtain the name of semaphore `s`.

See also: `New::Semaphore, Count::Semaphore`

Release::Semaphore [Release]

```
s = gRelease(s, cnt)

object  s;
int     cnt;
```

This method is used to release a lock on a semaphore. The semaphore lock count will be incremented by `cnt`, which is normally 1. As soon as a semaphore's lock count goes above 0 any threads which are waiting for the semaphore will be taken off hold and receive the lock (limited to the number of locks available). This method returns the semaphore instance passed.

See also: `WaitFor::Semaphore`

`WaitFor::Semaphore` [WaitFor]

```
r = gWaitFor(s)

object  s;
int     r;
```

This method is used to obtain a lock on semaphore `s`. If the semaphore has already exceeded its lock count the current thread will be placed on hold until the semaphore is available. Each call to `WaitFor` uses up a single count associated with semaphore `s`. This method always returns 0.

See also: `Release::Semaphore`

5.41 Sequence Class

The `Sequence` class is an abstract class which serves the sole purpose of grouping several subclasses which have common functionality. There is no specific class or instance methods directly associated with this class.

The subclasses of this class (`LinkSequence`, `LinkObjectSequence` and `SetSequence`) are used to provide a convenient mechanism to enumerate (or sequence) through all the elements of a collection.

5.42 Set Class

The `Set` class is used to store a collection of objects which contain some means of unique identification for quick access. The objects stored may be any Dynace object. `gHash` and `gCompare` will be used on the object to be added in order to store and compare it amongst other objects in the `Set`.

Since Dynace contains default methods for `Hash` and `Compare`, any group of Dynace objects may be stored in any `Set` instance.

If values are to be associated with the keys one of the `Dictionary` classes would be preferred. The `Set` class, however, provides most of the functionality needed by the `Dictionary` classes.

See the examples included with the Dynace system for an illustration of the use of the `Set`/`Dictionary` related classes.

5.42.1 Set Class Methods

The only class method used in this class is one to create instances of itself.

`New::Set` [New]

```
i = gNew(Set);

object  i;
```

This class method creates instances of the `Set` class. The set created is designed to accommodate rather small sets of about 30 elements. If a `Set` becomes too full it automatically re-sizes itself. See `NewWithInt::Set` for a method of creating a set while specifying its initial size.

The new instance is returned.

See also: `NewWithInt::Set, Resize::Set`

`NewWithInt::Set` [NewWithInt]

```
i = gNewWithInt(Set, size);

int     size;
object  i;
```

This class method creates instances of the `Set` class. `size` is used to give the instance some idea of the initial maximum number of objects to be stored in the `Set`. The actual number of objects may exceed this number, however, some loss of efficiency will occur.

Since a `Set` is implemented as a hash table it is best if `size` is actually larger than the number of objects to be held. It should be an odd number or better yet a prime

number. Keep in mind, however, that there is storage requirements associated with this number and the use of a very large number will consume large memory resources. In reality most reasonable numbers will work fine.

Regardless of the size specified the set object will never overflow. It will just be less efficient if many more elements then `size` are added.

The new instance is returned.

See also: `New::Set`, `Resize::Set`

5.42.2 Set Instance Methods

The instance methods associated with this class provide all the means to add, inquire, remove and print instances of the `Set` class.

`Add::Set` [Add]

```
r = gAdd(i, luk);

object  i;
object  luk;
object  r;
```

This method is used to add a new object (`luk`) to the `Set` instance (`i`). If an object with the same key already exists in the `Set` it will be left as is and `Add` will return `NULL`. If `luk` is added it will also be returned.

`luk` may be any Dynace object. The `gHash` and `gCompare` generics are used to find and compare the various objects in the `Set`.

See also: `Find::Set`, `FindAdd::Set`, `RemoveObj::Set`

`Copy::Set` [Copy]

```
so = gCopy(i);

object  i, so;
```

This method is used to create a new instance of the `Set` class with the *same* elements as `Set i`. Or put more simply, it makes a copy. The new `Set` instance is returned.

Note that this method is designed to also work correctly for the `Dictionary` subclasses of `Set`.

Example:

```
object  x, y;

x = gNewWithInt(Set, 101);
y = gCopy(x);
```

See also: `NewWithInt::Set`, `DeepCopy::Set`

`DeepCopy::Set` [DeepCopy]

```
so = gDeepCopy(i);

object  i, so;
```

This method is used to create a new instance of the `Set` class which contains *copies* of all the elements from the original `Set`. `DeepCopy` is used on each element to create the copies. The new `Set` instance is returned.

Note that this method is designed to also work correctly for the `Dictionary` subclasses of `Set`.

Example:

```
object  x, y;

x = gNewWithInt(Set, 100);
y = gDeepCopy(x);
```

See also: `NewWithInt::Set`, `Copy::Set`

`DeepDispose::Set` [DeepDispose]

```
r = gDeepDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to dispose of an entire `Set` instance. It also disposes of all the objects or associations in the `Set` by use of the `DeepDispose` method.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `Dispose::Set`, `Dispose1::Set`, `DisposeAllNodes::Set`

`DeepDisposeAllNodes::Set` [DeepDisposeAllNodes]

```
i = gDeepDisposeAllNodes(i);

object  i;
```

This method is used to remove all objects in a `Set` instance without disposing of the set itself. Each object or association is disposed of via `DeepDispose`.

The value returned is always the instance passed.

See also: `DisposeAllNodes::Set, Dispose::Set, DisposeAllNodes1::Set`

`DeepDisposeGroup::Set` [DeepDisposeGroup]

```
i = gDeepDisposeGroup(i, fun);

object  i;
int     (*fun)(object);
```

This method is used to remove and dispose of a group of objects from the `Set i`. The function, `fun`, is executed for each object in the `Set`. With each call `fun` is passed the object at that point in the `Set`. If `fun` returns a 1 that particular object will be removed, or else if `fun` returns a 0 the object will not be removed. `fun` is used to decide which objects will be removed.

The objects removed are disposed of by use of the `DeepDispose` method.

See also: `GroupRemove::Set, DisposeGroup::Set, DisposeAllNodes::Set`

`DeepDisposeObj::Set` [DeepDisposeObj]

```
r = gDeepDisposeObj(i, luk);

object  i;
object  luk;
object  r;
```

This method is used to remove and dispose of an object (`luk`) from the `Set` instance (`i`). If it is not found `NULL` is returned, otherwise `i` is returned. `luk` is disposed of by use of the `DeepDispose` method.

`luk` may be any Dynace object. The `gHash` and `gCompare` generics are used to find and compare the various objects in the `Set`.

See also: `RemoveObj::Set, DisposeObj::Set`

Dispose::Set [Dispose]

```
r = gDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to dispose of an entire `Set` instance. It does not dispose of any of the objects or associations in the `Set`.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `DeepDispose::Set`, `Dispose1::Set`, `DisposeAllNodes::Set`

Dispose1::Set [Dispose1]

```
gDispose1(i);

object  i;
```

This method is used to dispose of an entire `Set` instance. It also disposes of all of the objects or associations in the `Set` by the use of the `Dispose` method applied to each object. There is no value returned.

See also: `DeepDispose::Set`, `Dispose::Set`, `DisposeAllNodes1::Set`

DisposeAllNodes::Set [DisposeAllNodes]

```
i = gDisposeAllNodes(i);

object  i;
```

This method is used to remove all objects in a `Set` instance without disposing of the set itself. It does not dispose of any of the objects or associations in the `Set`, they are simply disassociated with the `Set`.

The value returned is always the instance passed.

See also: `DeepDisposeAllNodes::Set`, `DisposeAllNodes1::Set`, `Dispose::Set`

`DisposeAllNodes1::Set` [DisposeAllNodes1]

```
i = gDisposeAllNodes1(i);

object  i;
```

This method is used to remove all objects in a `Set` instance without disposing of the set itself. Each object or association in the set are disposed of via `Dispose`.

The value returned is always the instance passed.

See also: `DisposeAllNodes::Set, DeepDisposeAllNodes::Set, Dispose::Set`

`DisposeGroup::Set` [DisposeGroup]

```
i = gDisposeGroup(i, fun);

object  i;
int     (*fun)(object);
```

This method is used to remove and dispose of a group of objects from the `Set i`. The function, `fun`, is executed for each object in the `Set`. With each call `fun` is passed the object at that point in the `Set`. If `fun` returns a 1 that particular object will be removed, or else if `fun` returns a 0 the object will not be removed. `fun` is used to decide which objects will be removed.

The objects removed are disposed of by use of the `Dispose` method.

See also: `GroupRemove::Set, DeepDisposeGroup::Set`

`DisposeObj::Set` [DisposeObj]

```
r = gDisposeObj(i, luk);

object  i;
object  luk;
object  r;
```

This method is used to remove and dispose of an object (`luk`) from the `Set` instance (`i`). If it is not found `NULL` is returned, otherwise `i` is returned. `luk` is disposed of by use of the `Dispose` method.

`luk` may be any Dynace object. The `gHash` and `gCompare` generics are used to find and compare the various objects in the `Set`.

See also: `RemoveObj::Set, DeepDisposeObj::Set`

`Find::Set` [Find]

```
r = gFind(i, luk);

object  i;
object  luk;
object  r;
```

This method is used to find an existing object (`luk`) in the `Set` instance (`i`). If the object is found in the `Set` it will be returned otherwise `Find` will return `NULL`.

`luk` may be any Dynace object. The `gHash` and `gCompare` generics are used to find and compare the various objects in the `Set`.

See also: `Add::Set`, `FindAdd::Set`, `RemoveObj::Set`

`FindAdd::Set` [FindAdd]

```
r = gFindAdd(i, luk);

object  i;
object  luk;
object  r;
```

This method is used to find an existing object (`luk`) in the `Set` instance (`i`) and return it. If the object is not found it will be added and returned.

`luk` may be any Dynace object. The `gHash` and `gCompare` generics are used to find and compare the various objects in the `Set`.

See also: `Add::Set`, `Find::Set`, `RemoveObj::Set`

`First::Set` [First]

```
r = gFirst(i);

object  i;
object  r;
```

This method returns one of the objects in the set. The returned object will be an arbitrary one in the set. This method is useful in cases where you want to enumerate through each object in the set where you will perform some operation on the returned object and then remove or dispose it from the set. In that case the next call to `gFirst` will simply return a different object from the set.

If no objects are left in the set, `gFirst` returns `NULL`.

See also: `RemoveObj::Set, Sequence::Set`

`ForAll::Set` [ForAll]

```
r = gForAll(i, fun);

object  i;
object  (*fun)(object);
object  r;
```

This method is used to execute function `fun` on each object in a `Set`.

Each time `fun` is called it is passed the next object in the `Set` as its first and only argument. If `fun` returns a non-NULL object, `ForAll` returns immediately with the value returned by `fun`. As long as `fun` returns NULL it will be passed successive objects in the `Set` until there are no more objects. At that time `ForAll` will just return NULL indicating that all objects have been enumerated.

See also: `Sequence::Set`

`GroupRemove::Set` [GroupRemove]

```
i = gGroupRemove(i, fun);

object  i;
int     (*fun)(object);
```

This method is used to remove a group of objects from the `Set i`. The function, `fun`, is executed for each object in the `Set`. With each call `fun` is passed the object at that point in the `Set`. If `fun` returns a 1 that particular object will be removed, or else if `fun` returns a 0 the object will not be removed. `fun` is used to decide which objects will be removed.

The objects removed are not disposed.

See also: `DisposeGroup::Set, DeepDisposeGroup::Set`

`RemoveObj::Set` [RemoveObj]

```
r = gRemoveObj(i, luk);

object  i;
object  luk;
object  r;
```

This method is used to remove an object (`luk`) from the `Set` instance (`i`) and return it. If the object is not found `NULL` will be returned. `luk` is not disposed.

`luk` may be any Dynace object. The `gHash` and `gCompare` generics are used to find and compare the various objects in the `Set`.

See also: `DisposeObj::Set, DeepDisposeObj::Set`

`Resize::Set` [Resize]

```
r = gResize(i, size);

object  i;
int     size;
object  r;
```

This method is used to change the size of the hash table used to implement the `Set`. It may be used if during runtime it is discovered that the number of objects stored or to be stored in the `Set` are very different than the number used when the `Set` was created.

Since a `Set` is implemented as a hash table it is best if `size` is actually larger than the number of objects to be held. It should be an odd number or better yet a prime number. Keep in mind, however, that there is storage requirements associated with this number and the use of a very large number will consume large memory resources. The `Set` will support more objects than defined by `size` at a lesser efficiency. In reality most reasonable numbers will work fine.

`r` is the re-sized `Set`.

This method is very time consuming and should only be used when absolutely necessary.

See also: `NewWithInt::Set`

`Sequence::Set` [Sequence]

```
s = gSequence(i);

object  i;
object  s;
```

This method takes an instance of the `Set` class (`i`) and returns an instance of the `SetSequence` class (`s`). The `Set` represented by `i` is not effected by this operation. The `Set` sequence item, `s`, is used to enumerate through all the objects in `Set i`. See the `SetSequence` class for documentation of other methods needed to use `s`.

Example:

```
object  set; /*  Set           */
object  s;   /*  Set sequence  */
object  obj; /*  an object     */

/*  set must be initialized previously  */

for (s=gSequence(set) ; obj = gNext(s) ; )  {
        /*  do something with obj  */
}
```

See also: `Next::SetSequence`, `First::Set`

`Size::Set` [Size]

```
sz = gSize(i);

object  i;
int     sz;
```

This method is used to obtain the number of objects in the `Set`.

`StringRep::Set` [StringRep]

```
s = gStringRep(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the object `i` and its value. This is often used to print or display a representation of an object. It is also used by by `Print::Object` (a method useful during the debugging phase of a project) in order to directly print an object to a stream.

This method obtains the values of each object by calling `StringRepValue` on each element.

See also: `Print::Object`, `PrintValue::Object`

5.43 SetSequence Class

This class is a subclass of `Sequence` and is used to provide a mechanism to enumerate through all the objects in a `Set` (or one of its subclasses) without effecting the structure.

Typically, the `Sequence` instance method of the `Set` class is used to create the instances of the `SetSequence` class.

5.43.1 SetSequence Class Methods

There is only one class method associated with this class. It is used to create new instances of itself and is only called by the `Sequence` method associated with the `Set` class.

5.43.2 SetSequence Instance Methods

This class only has a single instance method which is used to enumerate through objects in the `Set`.

`Next::SetSequence` [Next]

```
obj = gNext(i);

object  i;
object  obj;
```

This method is used to enumerate through all the objects in a `Set`. Each time `Next` is called the following object in the `Set` is returned. It does this in a non-destructive way so the associated `Set` is not effected.

When `Next` is called after the last object has been returned (or if there are no objects in the `Set`) it will return `NULL` and automatically dispose of the sequence object `i`. Therefore, the only time `Dispose` would be needed (if the garbage collector weren't being used) would be if the entire list was not enumerated.

Note that if this method is used with one of the `Dictionary` classes each call to `Next` returns the appropriate `Association` type. The `gValue` and `gKey` generics may be used to obtain the desired values from the `Association` returned.

Example:

```
object  set;  /*  Set                         */
object  s;    /*  Set sequence                */
object  obj;  /*  one object or association  */

/*  set must be initialized previously  */

for (s=gSequence(set) ; obj = gNext(s) ; )  {
        /*  do something with object obj  */
}
```

See also: `Sequence::Set`

5.44 ShortArray Class

This class, which is a subclass of `NumberArray`, is used to represent arbitrary shaped arrays of the C language `short` data type in an efficient manner. Although this class implements much of its own functionality most of it is documented within the `NumberArray` and `Array` classes because the interface is shared by all subclasses of the `NumberArray` class.

5.44.1 ShortArray Class Methods

There is only one class method which is particular to this class.

`Iota::ShortArray` [Iota]

```
ary = gIota(ShortArray, n);

object    ary;
int       n;
```

This method is used to create a `ShortArray` which has 1 dimension of length `n`. The first element is initialized with the value set by `IndexOrigin::Array` and each successive element is given a value one greater than the previous element.

The value returned is the new array.

Example:

```
object  ary;

ary = gIota(ShortArray, 5);
/*  ary = (0, 1, 2, 3, 4)  */
```

See also: `IndexOrigin::Array`

5.45 ShortInteger Class

The `ShortInteger` class is used to represent the C language `short` data type as a Dynace object. It is a subclass of the `Number` class. Even though the `ShortInteger` class implements most of its own functionality, it is documented as part of the `Number` class because most of the interface is the same for all subclasses of the `Number` class. Differences are documented in this section.

5.45.1 ShortInteger Class Methods

The `ShortInteger` class has only one class method and it is used to create new instances of itself.

`NewWithInt::ShortInteger` [NewWithInt]

```
i = gNewWithInt(ShortInteger, s);

int     s;
object  i;
```

This class method creates instances of the ShortInteger class. `s` is the initial value of the integer being represented. Note that `s` is of type `int`. That is because if you pass a short type to a generic function (which uses a variable argument declaration) the C language will automatically promote it to an `int`.

The value returned is a Dynace instance object which represents the integer passed.

Note that the default disposal methods are used by this class since there are no special storage allocation requirements.

Example:

```
object  x;

x = gNewWithInt(ShortInteger, 55);
```

See also: `ShortValue::Number`, `Dispose::Object`

5.45.2 ShortInteger Instance Methods

The instance methods associated with the `ShortInteger` class provide a means of changing and obtaining the value associated with an instance of the `ShortInteger` class. All of the `ShortInteger` class instance methods are documented in the `Number` class because of their common interface with all other subclasses of the `Number` class.

5.46 Socket Class

The `Socket` class is a subclass of `Stream` and provides facilities which make utilizing sockets easy and portable. Sockets provide a mechanism for network communication over TCP or the Internet.

At present only the client-side methods are implemented.

5.46.1 Socket Class Methods

The only class method is used to open a new socket which equates to forming a connection with a server.

`SocketConnect::Socket` [SocketConnect]

```
skt = gSocketConnect(Socket, addr, port);

char    *addr;
int     port;
object  skt;
```

This method is used to form a connection between a client and server. The application calling this method would be the client side. In order to form a connection the IP address ("123.456.789.012") or domain name ("prep.ai.mit.edu") as well as the port number of the server application must be known. `addr` is a character string representing the IP address or domain name, and `port` is the port number for the server application. This information would always be known by the organization responsible for the server end.

`gSocketConnect` returns an object which represents the connection or `NULL` if the connection was not made.

Example:

```
object  skt = gSocketConnect(Socket, "192.44.32.44", 4099);
```

See also: `Dispose::Socket`

5.46.2 Socket Instance Methods

Instance methods associated with the `Socket` class enable data to be sent to and from a client / server pair of applications.

`Dispose::Socket` [Dispose]

```
r = gDispose(skt);

object  skt;
object  r;     /*  NULL  */
```

This method is used to close a socket connection and dispose of its associated object. There is also a `gDeepDispose` which performs the same function.

`GetErrorCode::Socket` [GetErrorCode]

```
r = gGetErrorCode(skt);

object  skt;
int     r;
```

This method is used to obtain an error code which describes the last error on more detail.

`Read::Socket` [Read]

```
r = gRead(skt, buf, n);

object   skt;
char     *buf;
unsigned n;
int      r;
```

This method is used to read `n` bytes into `buf` from `Socket skt`. The value returned is the actual number of bytes read or -1 on error. An error can occur for two reasons: a severed connection, or a time out error. In the case of a severed connection one would not want to continue use of that socket since it is probably not valid anymore.

`gRead` normally waits up to 60 seconds for reception of the requested bytes. If part of the data is received `gRead` will again wait up to 60 seconds for the remaining data. `gRead` continues to wait for *all* the data unless a connection error occurs or no data is received for at least 60 seconds.

Example:

```
char    buf[80];

if (10 != gRead(skt, buf, 10))
        error code;
```

See also: `gTimedRead::Socket`

`RecvFile::Socket` [RecvFile]

```
r = gRecvFile(skt, tf);

object  skt;
char    *tf;
int     r;
```

This method is used to receive a whole file over a socket connection. `ff` is the name (and path) of the local file to where the received file is to be saved.

This method returns 0 on success or non-zero upon failure. This method makes some degree of effort (via acknowledgments and check sums) to assure that the file was in fact correctly received.

This method is unique to Dynace which means that the server end would have to have the correct receive function in order for this facility to be used.

Example:

```
if (gRecvFile(skt, "myFile"))
      handle error;
```

See also: `SendFile::Socket`

`SendFile::Socket` [SendFile]

```
r = gSendFile(skt, ff, tf);

object  skt;
char    *ff;
char    *tf;
int     r;
```

This method is used to transmit a whole file over a socket connection. `ff` is the name (and path) of the local file to be transmitted. `tf` is the name of the file that the receiving end will see (i.e. the name of the file the receiving end will think you are sending). If `tf` is `NULL` then `ff` is used.

This method returns 0 on success or non-zero upon failure. This method makes some degree of effort (via acknowledgments and check sums) to assure that the file was in fact correctly received by the other end.

This method is unique to Dynace which means that the server end would have to have the correct receive function in order for this facility to be used.

Example:

```
if (gSendFile(skt, "../myFile", NULL))
      handle error;
```

See also: `RecvFile::Socket`

`TimedRead::Socket` [TimedRead]

```
r = gTimedRead(skt, buf, n, w);

object   skt;
char     *buf;
unsigned n;
int      w;
int      r;
```

This method is used to read `n` bytes into `buf` from `Socket skt`. The value returned is the actual number of bytes read or -1 on error. An error can occur for two reasons: a severed connection, or a time out error. In the case of a severed connection one would not want to continue use of that socket since it is probably not valid anymore.

`gTimedRead` normally waits up to `w` seconds for reception of the requested bytes. If part of the data is received `gTimedRead` will again wait up to `w` seconds for the remaining data. `gTimedRead` continues to wait for *all* the data unless a connection error occurs or no data is received for at least `w` seconds.

Example:

```
char    buf[80];

if (10 != gTimedRead(skt, buf, 10, 60))
        error code;
```

See also: `gRead::Socket`

`TimedWrite::Socket` [TimedWrite]

```
r = gTimedWrite(skt, buf, n, w);

object   skt;
char     *buf;
unsigned n;
int      w;
int      r;
```

This method is used to send `n` bytes from `buf` to `Socket skt`. The value returned is the actual number of bytes sent or -1 on error. An error can occur for two reasons: a severed connection, or a time out error. In the case of a severed connection one would not want to continue use of that socket since it is probably not valid anymore.

`gTimedWrite` normally waits up to `w` seconds for the ability to send the requested bytes. If part of the data is sent `gTimedWrite` will again wait up to `w` seconds for the remaining data. `gTimedWrite` continues to wait until *all* the data is sent unless a connection error occurs or no data is able to send for at least `w` seconds.

Successfully sending data does not guarantee that the data was received on the other end. You must send back some kind of acknowledgement to assure that.

Example:

```
if (12 != gTimedWrite(skt, "Hello, World", 12, 60))
        error code;
```

See also: `gWrite::Socket`

`Write::Socket` [Write]

```
r = gWrite(skt, buf, n);

object   skt;
char     *buf;
unsigned n;
int      r;
```

This method is used to send `n` bytes from `buf` to `Socket skt`. The value returned is the actual number of bytes sent or -1 on error. An error can occur for two reasons: a severed connection, or a time out error. In the case of a severed connection one would not want to continue use of that socket since it is probably not valid anymore.

`gWrite` normally waits up to 60 seconds for the ability to send the requested bytes. If part of the data is sent `gWrite` will again wait up to 60 seconds for the remaining data. `gWrite` continues to wait until *all* the data is sent unless a connection error occurs or no data is able to send for at least 60 seconds.

Successfully sending data does not guarantee that the data was received on the other end. You must send back some kind of acknowledgement to assure that.

Example:

```
if (12 != gWrite(skt, "Hello, World", 12))
        error code;
```

See also: `gTimedWrite::Socket`

5.47 Stream Class

The `Stream` class is an abstract class used to group other classes which poses the ability to read and write characters sequentially. Although much of the actual functionality is implemented by its subclasses, the methods are documented here because of their common interface.

5.47.1 Stream Class Methods

There are no class methods associated with this class.

5.47.2 Stream Instance Methods

Although most of the following methods are actually implemented in the subclasses of `Stream`, they are documented here because of their common interface. Note that some methods are not available for all subclasses of the `Stream` class. This would occur if the method is inappropriate for a particular `Stream` subclass (such as seeking backward on a `String`).

`Advance::Stream` [Advance]

```
r = gAdvance(i, n);

object  i;
long    n;
long    r;
```

This method is used to advance the position of the next read of `Stream` `i` forward by `n` bytes. The value returned is the actual number of bytes advanced or 0 on error.

See also: `Seek::Stream`, `Retreat::Stream`

`EndOfStream::Stream` [EndOfStream]

```
r = gEndOfStream(i);

object  i;
int     r;
```

This method returns 1 if `Stream` `i` is at the end of available characters and 0 otherwise. by `i`.

See also: `Length::Stream`, `Position::Stream`

`Gets::Stream` [Gets]

```
r = gGets(i, buf, sz);

object  i;
char    *buf;
int     sz;
char    *r;
```

This method is used to read a new-line terminated string from `Stream` `i` into `buf`. It will read no more than `sz-1` bytes, but room permitting, the new-line will be included. `r` will be the same as `buf` unless there is an error, in which case `r` will be `NULL`.

Example:

```
char    buf[80];

gGets(stdinStream, buf, sizeof buf);
```

See also: `Read::Stream`, `Puts::Stream`

`Length::Stream` [Length]

```
r = gLength(i);

object  i;
long    r;
```

This method is used obtain the total length of the `Stream` represented by `i`.

See also: `Seek::Stream`, `Position::Stream`

`Position::Stream` [Position]

```
r = gPosition(i);

object  i;
long    r;
```

This method is used obtain the current position in the stream relative to its beginning.

See also: `Seek::Stream`, `Length::Stream`

`Printf::Stream` [Printf]

```
r = vPrintf(i, fmt, ...);

object  i;
char    *fmt;
int     r;
```

This method is used to write a formatted string to stream `i`. `fmt` specifies the format of the output as well as the types and number of remaining arguments. Since this method is implemented by the use of the `vsprintf` function supplied by the host system you may find more detailed description of the arguments to this method in the manual supplied by your host compiler (under `fprintf`). The only difference would be the first argument which would now be a `Stream` instance.

The value returned is the number of bytes actually written to the stream.

Example:

```
vPrintf(stdoutStream, "I am %d years old.\n", 10);
```

See also: `Puts::Stream`

`Putc::Stream` [Putc]

```
r = gPutc(i, ch);

object  i;
char    ch;
int     r;
```

This method is used to write a character (`ch`) to stream `i`. The value returned is the character written or `EOF` if error.

Example:

```
gPutc(stdoutStream, 'x');
```

See also: `Puts::Stream`

`Puts::Stream` [Puts]

```
r = gPuts(i, str);

object  i;
char    *str;  /*  or:  object str;  */
int     r;
```

This method is used to write a string (`str`) to stream `i`. `str` may also be an instance of the `String` class in which case the associated string will be written. All the characters up to the first null byte will be written. The value returned is the number of bytes written to the stream.

Example:

```
gPuts(stdoutStream, "Hello\n");
```

See also: `Putc::Stream`, `Printf::Stream`

`Read::Stream` [Read]

```
r = gRead(i, buf, n);

object  i;
char    *buf;
unsigned n;
int     r;
```

This method is used to read `n` bytes into `buf` from `Stream` `i`. The value returned is the actual number of bytes read or -1 on error.

Example:

```
char    buf[80];

gRead(stdinStream, buf, 10);
```

See also: `Gets::Stream`

`Retreat::Stream` [Retreat]

```
r = gRetreat(i, n);

object  i;
long    n;
long    r;
```

This method is used to retreat the position of the next read of `Stream` `i` by `n` bytes. The value returned is the actual number of bytes retreated or 0 on error.

See also: `Seek::Stream`, `Advance::Stream`

`Seek::Stream` [Seek]

```
r = gSeek(i, n);

object  i;
long    n;
long    r;
```

This method is used to seek to position `n` of `Stream` `i` relative to the beginning of the stream. The value returned is the actual position seeked to or 0 on error.

See also: `Retreat::Stream`, `Advance::Stream`

`Write::Stream` [Write]

```
r = gWrite(i, buf, n);

object  i;
char    *buf;
unsigned n;
int     r;
```

This method is used to write `n` bytes from `buf` into `Stream` `i`. The value returned is the actual number of bytes written or -1 on error.

Example:

```
gWrite(stdoutStream, "Hello", 5);
```

See also: `Puts::Stream`, `Read::Stream`

5.47.3 Stream Global Variables

There are several global variables associated with the `Stream` class. These variables represent standard streams used throughout Dynace. They are used for diagnostic output as well as any other input or output which may be necessary on a global basis. These variables may be assigned different streams in order to redirect the output. The remainder of this section documents all the global stream variables.

`stderrStream::Stream` [stderrStream]

```
object  stderrStream;
```

The `stderrStream` global variable is used to direct error messages to the correct output medium. It is initialized to an instance of the `File` class using the C language `stderr` stream. It may be set to any instance of a subclass of `Stream`.

See also: `stdoutStream::Stream`

`stdinStream::Stream` [stdinStream]

```
object  stdinStream;
```

The `stdinStream` global variable is used as a global mechanism for obtaining input. It is initialized to an instance of the `File` class using the C language `stdin` stream. It may be set to any instance of a subclass of `Stream`.

See also: `stdoutStream::Stream`

`stdoutStream::Stream` [stdoutStream]

```
object  stdoutStream;
```

The `stdoutStream` global variable is used to direct normal output to the correct output medium. It is initialized to an instance of the `File` class using the C language `stdout` stream. It may be set to any instance of a subclass of `Stream`.

See also: `stderrStream::Stream`

`traceStream::Stream` [traceStream]

```
object  traceStream;
```

The `traceStream` global variable is used to direct Dynace trace output to the correct output medium. It is initialized to an instance of the `File` class using the C language `stdout` stream. It may be set to any instance of a subclass of `Stream`.

See also: `stderrStream::Stream`

5.48 String Class

The `String` class is used to represent the C language `char *` data type as a Dynace object. It is a subclass of the `Stream` class and it provides an array of methods to operate on the string objects. Unlike C strings this class will dynamically expand a string buffer as needed so you should never have to worry about the buffer size. You can also use the `MA_compact` function to compact the string memory and reduce memory fragmentation.

Most of the `String` methods which take string arguments (except the first argument) will accept a C language string (`char *`) or an instance of the `String` class arbitrarily. The system can tell the difference between them at runtime. If it is a valid Dynace object the system will assure that it is a kind of `String`, otherwise it is assumed to be a C string.

Instances of this class will also operate like other subclasses of `Stream`. That is you can write (append), read (copy and remove), advance (remove leading characters) and obtain its length. This comes in handy when you wish to keep a small (and fast) sequential file in memory.

5.48.1 String Class Methods

The String class has several class methods which create new `String` instances. They are provided to lend various levels of flexibility.

`Build::String` [Build]

```
i = vBuild(String, ...);

object  i;
```

This class method creates a new instance of the String class. The initial value of the new instance will be the concatenation of all the `String` or `char *` arguments up to an `END`.

Each argument may be a C language `char *` or an instance of the `String` class. Dynace can tell the difference between them at runtime. If it is a valid Dynace object the system will assure that it is a kind of `String`, otherwise it is assumed to be a C string.

Note that the variable argument list *MUST* end with an `END`. There is no other way in C to tell when the last argument has been reached. If you forget the `END` the system will probably hang! (If you have a problem with the system hanging you might want to check all the `vBuild` generic calls.)

The value returned is the new `String` instance.

Example:

```
object  x, y;

x = gNewWithStr(String, " World");
y = vBuild(String, "Hello", x, "!", END);
/*  y = "Hello World!"  */
```

See also: `StringValue::String`, `Dispose::String`, There is also an instance method `Build::String`

`MaskFunction::String` [MaskFunction]

```
fun = gMaskFunction(String, ch);

char    ch;
int     (*fun)();
```

This method is used to return the masking function associated with the mask character `ch`.

Currently there are only two masking characters used in the system, `` `#' `` (numbers) and `` `@' `` (letters). The `` `#' `` character's associated function is `isdigit` while the `` `@' `` character's associated function is `isalpha`.

The masking function associated with a masking character takes a character as its only parameter and returns non-zero if the character is the masking character associated with the function and returns zero if it is not.

Example:

```
int     (*fun)();
char    *str = "A1";
int     r;

fun = gMaskFunction(String, '#');       /*  Uses isdigit  */
r = fun(str[0]);                        /*  r == 0        */
r = fun(str[1]);                        /*  r == 1        */
```

See also: `gSetMaskFiller::String`, `gApplyMask::String`, `gRemoveMask::String`

`New::String` [New]

```
i = gNew(String);

object  i;
```

This class method creates instances of the String class. The returned String object is initialized to "".

The value returned is a Dynace instance object which represents a null string.

Example:

```
object  x;

x = gNew(String);
```

See also: `Copy::String`, `StringValue::String`, `Dispose::String`, `NewWithStr::String`, `NewWithObj::String`

`NewWithStr::String` [NewWithStr]

```
i = gNewWithStr(String, val);

char *val;  /*  or object  */
object  i;
```

This class method creates instances of the String class. `val` is the initial value of the string being represented.

`val` may be a C language `char *` or an instance of the `String` class. Dynace can tell the difference between them at runtime. If it is a valid Dynace object the system will assure that it is a kind of `String`, otherwise it is assumed to be a C string.

The value returned is a Dynace instance object which represents the string passed.

Example:

```
object  x, y;

x = gNewWithStr(String, "Hello World!");
y = gNewWithStr(String, (char *) x);
```

See also: `Copy::String`, `StringValue::String`, `Dispose::String` `New::String`, `NewWithObj::String`

`NewWithObj::String` [NewWithObj]

```
i = gNewWithObj(String, val);

object  val;  /*  or char *  */
object  i;
```

This class method creates instances of the String class. `val` is the initial value of the string being represented.

`val` may be a C language `char *` or an instance of the `String` class. Dynace can tell the difference between them at runtime. If it is a valid Dynace object the system will assure that it is a kind of `String`, otherwise it is assumed to be a C string.

The value returned is a Dynace instance object which represents the string passed.

Example:

```
object  x, y;

x = gNewWithObj(String, (object) "Hello World!");
y = gNewWithObj(String, x);
```

See also: `Copy::String`, `StringValue::String`, `Dispose::String`
`NewWithStr::String`, `New::String`

`Sprintf::String` [Sprintf]

```
i = vSprintf(String, fmt, ...);

char    *fmt;
object  i;
```

This class method creates a new instances of the String class. Except for the first argument and return value, this method takes the same arguments and performs the same function as the normal `sprintf` contained in your C library. See that documentation.

Note that except for the first argument, this method does not take Dynace objects.

The value returned is a Dynace instance object which represents the result of the operation.

Example:

```
object  x;

x = vSprintf(String, "%s is %d years old", "John", 66);
```

See also: `StringValue::String`, `Dispose::String`

5.48.2 String Instance Methods

The instance methods associated with the String class provide a means of changing and obtaining the value associated with an instance of the String class. Methods are also included to help the generic container classes to quickly access members of this class.

Append::String [Append]

```
i = gAppend(i, str);

object  i;
char    *str;  /*  or object  */
```

This method is used to append the string represented by `str` to the end of the string represented by `i`.

`str` may be a C language `char *` or an instance of the `String` class. Dynace can tell the difference between them at runtime. If it is a valid Dynace object the system will assure that it is a kind of `String`, otherwise it is assumed to be a C string.

The value returned is the `String` instance passed (`i`).

Example:

```
object  x;

x = gNewWithStr(String, "Hello");
gAppend(x, " World!");
/*  x = "Hello World!"  */
```

See also: `ChangeValue::String`, `Build::String`

ApplyMask::String [ApplyMask]

```
i = gApplyMask(i, mask, text);

object  i;
char    *mask;
char    *text;
```

This method is used to apply `mask` to `text` and set the resulting value to `i`. If `NULL` or an empty string (`""`) is passed in to `mask` then `i` has its string value changed to `text`. If `NULL` or an empty string (`""`) is passed in to `text` then `mask` is applied directly to the current string value in `i`.

Mask definitions contain any combination of mask characters and literal characters. The first character of a mask can optionally be either '`>`' or '`<`' indicating the direction which the mask is applied to the string. If the first character is '`<`' then the mask is applied from right to left. If the first character is '`>`' or is neither of the directional characters, then the mask is applied from left to right. The mask character '#' indicates that the character in this position must be a number while the mask character '@' indicates that the character in this position must be a letter. Any other character in the mask is a literal character and is left in the resulting string in its position unchanged.

The value returned is the `String` instance passed (`i`).

Example:

```
object  x;
char    mask = "<(###) ###-####";

x = gNewWithStr(String, "1234567890");
gApplyMask(x, mask, NULL);         /* x = "(123) 456-7890" */
gApplyMask(x, mask, "9876543210"); /* x = "(987) 654-3210" */
gApplyMask(x, NULL, "1234567890"); /* x = "1234567890"     */
```

See also: `gMaskFunction::String`, `gSetMaskFiller::String`, `gRemoveMask::String`

`Build::String` [Build]

```
i = vBuild(str, first, ...);

object  str;
char    *first;  /*  or object  */
object  i;
```

This method is used to modify the string associated with instance `str` by optionally overwriting and optionally appending a group of C strings or `String` objects.

`first` may be an instance of the `String` class, a C string `char *` or a `NULL`. If `first` is `NULL` the initial value of `str` will remain intact, otherwise, it will be replaced with the string represented by `first`.

After the `first` argument has been processed the remaining `String` or `char *` arguments, up to an `END`, will be concatenated to the end of `str`.

Each argument may be a C language `char *` or an instance of the `String` class. Dynace can tell the difference between them at runtime. If it is a valid Dynace object the system will assure that it is a kind of `String`, otherwise it is assumed to be a C string.

Note that the variable argument list *MUST* end with an `END`. There is no other way in C to tell when the last argument has been reached. If you forget the `END` the system will probably hang! (If you have a problem with the system hanging you might want to check all the `vBuild` generic calls.)

The value returned is the `String` instance passed (`str`).

Example:

```
object  x;

x = gNewWithStr(String, "Hello");
vBuild(x, NULL, " World", gNewWithStr(String, "!"), END);
/*  x = "Hello World!"  */

vBuild(x, "Goodbye", " World", "!", END);
/*  x = "Goodbye World!"  */
```

See also: `ChangeValue::String`, `Append::String`, There is also a class method `Build::String`

`ChangeCharAt::String` [ChangeCharAt]

```
i = gChangeCharAt(i, n, c);

object  i;
int     n;
int     c;
```

This method is used to change the character at position `n` within `String` instance `i` to character `c`. The index origin is zero.

If `n` is equal to the number of characters in instance `i` (that is one greater than the number of indexable characters), string `i` will be extended by one character (`c`). However, if any larger index is used Dynace will issue an error message and abort the program.

Example:

```
object  x;

x = gNewWithStr(String, "Hello");
gChangeCharAt(x, 1, 'E');
/*  x = "HEllo"  */
```

See also: `CharValueAt::String`

`ChangeStrValue::String` [ChangeStrValue]

```
i = gChangeStrValue(i, val);

object  i;
char   *val;
```

This method is used to change the value associated with an instance of the String class. The new value may be of any length.

The value returned is the instance `i`.

Example:

```
object  x, y;

x = gNewWithStr(String, "Hello World");
gChangeStrValue(x, "Are you all using Dynace?");
```

See also: `ChangeValue::String`, `Append::String`, `Build::String`

`ChangeValue::String` [ChangeValue]

```
i = gChangeValue(i, val);

object  i;
object  val;  /*  or char *  */
```

This method is used to change the value associated with an instance of the String class. The new value may be of any length.

`val` may be a C language `char *` or an instance of the `String` class. Dynace can tell the difference between them at runtime. If it is a valid Dynace object the system will assure that it is a kind of `String`, otherwise it is assumed to be a C string.

The value returned is the instance `i`.

Example:

```
object  x, y;

x = gNewWithStr(String, "Hello World");
y = gNewWithStr(String, "Are you all using Dynace?");
gChangeValue(x, y);
```

See also: `ChangeStrValue::String`, `Append::String`, `Build::String`

`CharValueAt::String` [CharValueAt]

```
c = gCharValueAt(i, n);

object  i;
int     n;
char    c;
```

This method is used to obtain character `n` from `String` instance `i`. The index origin is zero. If an index greater than the available characters is used Dynace will issue an error message and abort the program.

Example:

```
object  x;
char    c;

x = gNewWithStr(String, "Hello");
c = gCharValueAt(x, 1);
/*  c = 'e'  */
```

See also: `ChangeCharAt::String`

`Compare::String` [Compare]

```
r = gCompare(i, obj);

object  i;
object  obj;  /*  or char *  */
int     r;
```

This method is used to determine the equality of the values represented by `i` and `obj`.

`obj` may be a C language `char *` or an instance of the `String` class. Dynace can tell the difference between them at runtime. If it is a valid Dynace object the system will assure that it is a kind of `String`, otherwise it is assumed to be a C string.

`r` is -1 if the value represented by `i` is less than the value represented by `obj`, 1 if the value of `i` is greater than `obj`, and 0 if they are equal.

This method is also used by the generic container classes to determine object relationships.

See also: `Equal::String`, `CompareI::String`, `CompareN::String`

`CompareI::String` [CompareI]

```
r = gCompareI(i, obj);

object  i;
object  obj;  /*  or char *  */
int     r;
```

This method is used to determine the equality of the values represented by `i` and `obj` without regard for the case of the letters. Therefore, strings such as “abc” and “ABC” will be considered equal.

`obj` may be a C language `char *` or an instance of the `String` class. Dynace can tell the difference between them at runtime. If it is a valid Dynace object the system will assure that it is a kind of `String`, otherwise it is assumed to be a C string.

`r` is -1 if the value represented by `i` is less than the value represented by `obj`, 1 if the value of `i` is greater than `obj`, and 0 if they are equal.

See also: `Compare::String, CompareN::String`

`CompareN::String` [CompareN]

```
r = gCompareN(i, obj, n);

object  i;
object  obj;  /*  or char *  */
int     n;    /*  max number of characters to compare  */
int     r;
```

This method is used to determine the equality of the values represented by `i` and `obj`. No more than `n` characters will be compared. If the two strings only differ past `n` characters they will be considered equal.

`obj` may be a C language `char *` or an instance of the `String` class. Dynace can tell the difference between them at runtime. If it is a valid Dynace object the system will assure that it is a kind of `String`, otherwise it is assumed to be a C string.

`r` is -1 if the value represented by `i` is less than the value represented by `obj`, 1 if the value of `i` is greater than `obj`, and 0 if they are equal.

See also: `Compare::String, CompareNI::String`

`CompareNI::String` [CompareNI]

```
r = gCompareNI(i, obj, n);

object  i;
object  obj;  /*  or char *  */
int     n;    /*  max number of characters to compare  */
int     r;
```

This method is used to determine the equality of the values represented by `i` and `obj` without regard for the case of the letters. Therefore, strings such as “abc” and “ABC” will be considered equal. No more than `n` characters will be compared. If the two strings only differ past `n` characters they will be considered equal.

`obj` may be a C language `char *` or an instance of the `String` class. Dynace can tell the difference between them at runtime. If it is a valid Dynace object the system will assure that it is a kind of `String`, otherwise it is assumed to be a C string.

`r` is -1 if the value represented by `i` is less than the value represented by `obj`, 1 if the value of `i` is greater than `obj`, and 0 if they are equal.

See also: `CompareI::String, CompareN::String`

`Copy::String` [Copy]

```
so = gCopy(i);

object  i, so;
```

This method is used to create a new instance of the `String` class with the same string value as string `i`. Or put more simply, it makes a copy. The new `String` instance is returned.

This method performs the same function as `DeepCopy::String`.

Example:

```
object  x, y;

x = gNewWithStr(String, "Hello");
y = gCopy(x);
/*  x = "Hello"  */
```

See also: `New::String, DeepCopy::String`

`DeepCopy::String` [DeepCopy]

```
so = gDeepCopy(i);

object  i, so;
```

This method is used to create a new instance of the `String` class with the same string value as string `i`. Or put more simply, it makes a copy. The new `String` instance is returned.

This method performs the same function as `Copy::String`.

Example:

```
object  x, y;

x = gNewWithStr(String, "Hello");
y = gDeepCopy(x);
/*  x = "Hello"  */
```

See also: `New::String`, `Copy::String`

`DeepDispose::String` [DeepDispose]

```
r = gDeepDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to dispose of an instance of the String class. It performs the same operation as `Dispose`.

Example:

```
object  x;

x = gNewWithStr(String, "Hello World");
gDeepDispose(x);
```

`Dispose::String` [Dispose]

```
r = gDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to dispose of an instance of the String class. It performs the same operation as `DeepDispose`.

Example:

```
object  x;

x = gNewWithStr(String, "Hello World");
gDispose(x);
```

`Drop::String` [Drop]

```
i = gDrop(i, num);

object  i;
int     num;
```

This method is used to drop `num` characters from string `i`. If `num` is positive characters will be dropped from the beginning of the string, otherwise, characters will be dropped from the end of the string. An attempt to drop more characters than exist will lead to a zero length string.

The modified `String` instance passed is returned.

Example:

```
object  x, y;

x = gNewWithStr(String, "Hello World");
gDrop(x, 4);   /*  x = "o World"    */
x = gNewWithStr(String, "Hello World");
gDrop(x, -4);   /*  x = "Hello W"    */
x = gNewWithStr(String, "ABC");
gDrop(x, 5);   /*  x = ""    */
```

See also: `Take::String`, `SubString::String`

`Equal::String` [Equal]

```
r = gEqual(i, obj);

object  i;
object  obj;  /*  or char *  */
int     r;
```

This method is used to determine the equality of the values represented by `i` and `obj`. If they contain the same string values a 1 is returned and 0 otherwise.

`obj` may be a C language `char *` or an instance of the `String` class. Dynace can tell the difference between them at runtime. If it is a valid Dynace object the system will assure that it is a kind of `String`, otherwise it is assumed to be a C string.

Example:

```
object  x;
int     r;

x = gNewWithStr(String, "Hello");
r = gEqual(x, "Hello");                 /*  r = 1  */
r = gEqual(x, gNewWithStr(String, "Hello"));  /*  r = 1  */
r = gEqual(x, "World");                 /*  r = 0  */
```

See also: `Compare::String`

`Hash::String` [Hash]

```
val = gHash(i);

object  i;
int     val;
```

This method is used by the generic container classes to obtain hash values for the object. `val` is a hash value between 0 and a large integer value.

See also: `Compare::String`

`JustifyCenter::String` [JustifyCenter]

```
i = gJustifyCenter(i);

object  i;
```

This method is used to center justify the text within a string. The length of the string is not modified, only the position of the characters within the string. It is often used in conjunction with `Take::String` in order to format text.

The modified `String` instance passed is returned.

Example:

```
object  x;

x = gNewWithStr(String, "ABD  DEF  DD      ");
gJustifyCenter(x);   /*  x = "   ABD  DEF  DD   "    */
x = gNewWithStr(String, "ABD  DEF");
gJustifyCenter(gTake(x, 14));   /*  x = "   ABD  DEF   "    */
```

See also: `Take::String`, `JustifyRight::String`, `JustifyLeft::String`

`JustifyLeft::String` [JustifyLeft]

```
i = gJustifyLeft(i);

object  i;
```

This method is used to left justify the text within a string. The length of the string is not modified, only the position of the characters within the string. It is often used in conjunction with `Take::String` in order to format text.

The modified `String` instance passed is returned.

Example:

```
object  x;

x = gNewWithStr(String, "    ABD  DEF  DD");
gJustifyLeft(x);   /*  x = "ABD  DEF  DD    "    */
x = gNewWithStr(String, "ABD  DEF");
gJustifyLeft(gTake(x, 10));   /*  x = "ABD  DEF  "    */
```

See also: `Take::String`, `JustifyCenter::String`, `JustifyRight::String`

`JustifyRight::String` [JustifyRight]

```
i = gJustifyRight(i);

object  i;
```

This method is used to right justify the text within a string. The length of the string is not modified, only the position of the characters within the string. It is often used in conjunction with `Take::String` in order to format text.

The modified `String` instance passed is returned.

Example:

```
object  x;

x = gNewWithStr(String, "ABD  DEF  DD    ");
gJustifyRight(x);   /*  x = "    ABD  DEF  DD"    */
x = gNewWithStr(String, "ABD  DEF");
gJustifyRight(gTake(x, 10));   /*  x = "  ABD  DEF"    */
```

See also: `Take::String`, `JustifyCenter::String`, `JustifyLeft::String`

`NumbPieces::String` [NumbPieces]

```
n = gNumbPieces(i, d);

object  i;
char    d;  /*  delimiter         */
int     n;  /*  number of pieces  */
```

This method is used to determine the number of sub-strings located in `i`. It is assumed that `i` is a string which contains the delimiter character `d`. That delimiter character is used to logically split string `i` into several sub-strings.

Example:

```
object  x;
int     n;

x = gNewWithStr(String, "ABD,DEF,DD");
n = gNumbPieces(x, ',');  /*  n = 3  */
```

See also: `Piece::String`

`Piece::String` [Piece]

```
r = gPiece(i, d, n);

object  i;
char    d;  /*  delimiter     */
int     n;  /*  piece wanted  */
object  r;  /*  sub-string    */
```

This method is used to create a new string object which will contain a copy of a sub-string of `i`. It is assumed that `i` is a string which contains the delimiter character `d`. That delimiter character is used to logically split string `i` into several sub-strings. `n` is used to specify which sub-string is desired.

This method returns a new string object which will contain a copy of the desired sub-string. A `NULL` is returned if `n` is greater than the number of sub-strings in `i`.

Example:

```
object  x, y;

x = gNewWithStr(String, "ABD,DEF,DD");
y = gPiece(x, ',', 0);  /*  y="ABD"  */
y = gPiece(x, ',', 1);  /*  y="DEF"  */
y = gPiece(x, ',', 2);  /*  y="DD"   */
```

See also: `NumbPieces::String`

`PrintLength::String` [PrintLength]

```
sz = gPrintLength(i, ts);

object  i;
int     ts;
int     sz;
```

This method is used to obtain the length of the string associated with the instance of the String class passed when it is printed. This method takes into account tabs, backspaces and return characters in its calculations.

`ts` is the tab stop size that the method should use.

Note that this is one of the few generics which doesn't return a Dynace object. It returns an integer.

Example:

```
object  x;
int     sz;

x = gNewWithStr(String, "Hello\tWorld\bx");
sz = gPrintLength(x, 8);
/*  sz = 13   */
```

See also: `Size::String`

`Process::String` [Process]

```
i = gProcess(i);

object  i;
```

This method is used to process a string with embedded escape sequences. The escape sequences will be processed and converted into the appropriate control characters. For example `"\r"` will be converted in to a return control character. All the ANSI standard escape sequences are supported. In addition, `"\e"` gets converted into the escape character. `"\^L"` (where L may be any letter - upper or lower case) gets converted into the corresponding control character, for example `"\^G"` will get converted to the control-G character. `"\dnnn"` works like `"\xnnn"` except that `nnn` is treated as a decimal number.

Example:

```
object  x;

x = gNewWithStr(String, "Hello\\tWorld\\bx");
gProcess(x);
/*  x will now contain the real control characters  */
```

See also: `PrintLength::String`

RemoveMask::String [RemoveMask]

```
i = gRemoveMask(i, mask, text);

object  i;
char    *mask;
char    *text;
```

This method is used to remove `mask` from `text` and set the resulting value to `i`. If `NULL` or an empty string (`""`) is passed in to `mask` then `i` has its string value changed to `text`. If `NULL` or an empty string (`""`) is passed in to `text` then `mask` is removed from the current string value in `i`.

The value returned is the `String` instance passed (`i`).

Example:

```
object  x;
char    mask = "<(###) ###-####";

x = gNewWithStr(String, "(123) 456-7890");
gRemoveMask(x, mask, NULL);            /*x="1234567890"    */
gRemoveMask(x, mask, "(987)654-3210"); /*x="9876543210"    */
gRemoveMask(x, NULL, "(123) 456-7890");/*x="(123) 456-7890"*/
```

See also: `gMaskFunction::String`, `gSetMaskFiller::String`, `gApplyMask::String`

SetMaskFiller::String [SetMaskFiller]

```
r = gSetMaskFiller(i, ch);

object  i;
char    ch;
char    r;
```

This method is used to set the character to be used to fill the unused space in a masked string. This character defaults to be a space but is frequently set to be the underscore (`` `_' ``) character. This method returns the previous mask filler character.

Example:

```
object  x;
char    r;

x = gNewWithStr(String, "1234567");
gApplyMask(x, "<(###) ###-####", NULL);
      /* x = "(   ) 123-4567" */
r = gSetMaskFiller(x, '_');
      /* r = ' ' (space)      */
gApplyMask(x, "<(###) ###-####", NULL);
      /* x = "(___) 123-4567" */
```

See also: `gMaskFunction::String`, `gApplyMask::String`, `gRemoveMask::String`

`Size::String` [Size]

```
sz = gSize(i);

object  i;
int     sz;
```

This method is used to obtain the length of the string associated with the instance of the String class passed.

Note that this is one of the few generics which doesn't return a Dynace object. It returns an integer.

Example:

```
object  x;
int     sz;

x = gNewWithStr(String, "Hello World");
sz = gSize(x);
/*  sz = 11   */
```

See also: `PrintLength::String`

`StringRepValue::String` [StringRepValue]

```
s = gStringRepValue(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the value associated with `i`. This is often used to print or display the value. It is also used by `PrintValue::Object` and indirectly by `Print::Object` (two methods useful during the debugging phase of a project) in order to directly print an object's value.

Example:

```
object  x;
object  s;

x = gNewWithStr(String, "Hello");
s = gStringRepValue(x);
  /* s represents "Hello" (with quotes)  */
```

See also: `StringValue::String`, `PrintValue::Object`, `Print::Object`

`StringValue::String` [StringValue]

```
val = gStringValue(i);

object  i;
char    *val;
```

This method is used to obtain the `char *` value associated with an instance of the String class. Note that this is one of the few generics which doesn't return a Dynace object. It returns a char * string.

Example:

```
object  x;
char    *val;

x = gNewWithStr(String, "Hello World");
val = gStringValue(x);
```

See also: `New::String`, `ChangeValue::String`

`StripCenter::String` [StripCenter]

```
i = gStripCenter(i);

object  i;
```

This method is used to strip off blank characters (space, tab, new line, carriage return) from both sides of a string. The return value is the modified `String` instance passed.

Example:

```
object  x, y;

x = gNewWithStr(String, "    ABD  DEF  DD    ");
gStripCenter(x);   /*  x = "ABD  DEF  DD"    */
```

See also: `StripRight::String`, `StripLeft::String`

`StripLeft::String` [StripLeft]

```
i = gStripLeft(i);

object  i;
```

This method is used to strip off blank characters (space, tab, new line, carriage return) from the left side of a string. The return value is the modified `String` instance passed.

Example:

```
object  x, y;

x = gNewWithStr(String, "    ABD  DEF  DD    ");
gStripLeft(x);   /*  x = "ABD  DEF  DD    "    */
```

See also: `StripRight::String`, `StripCenter::String`

`StripRight::String` [StripRight]

```
i = gStripRight(i);

object  i;
```

This method is used to strip off blank characters (space, tab, new line, carriage return) from the right side of a string. The return value is the modified `String` instance passed.

Example:

```
object  x, y;

x = gNewWithStr(String, "    ABD  DEF  DD    ");
gStripRight(x);   /*  x = "    ABD  DEF  DD"    */
```

See also: `StripLeft::String`, `StripCenter::String`

SubString::String [SubString]

```
so = gSubString(i, beg, num);

object  i;
int     beg, num;
object  so;
```

This method is used to create a new instance of the `String` class which is initialized to a copy of a sub-string of instance `i`. `beg` indicates the beginning position number of the string to be copied, index origin zero. `num` is the number of characters to copy.

If `beg` is negative, `beg` will be converted to a beginning point relative to the end of the string. Therefore, a `-1` would indicate a starting point of the last character in the string and `-2` would indicate a starting point of the second to the last character in the string. Zero indicates to start at the beginning of the string.

If `num` is negative, `-num` characters *before* `beg` will be taken.

The new `String` instance is returned.

Example:

```
object  x, y;

x = gNewWithStr(String, "Hello World");
y = gSubString(x, 1, 3);   /*  y = "ell"    */
y = gSubString(x, -5, 5);  /*  y = "World"  */
```

See also: `Take::String`, `Drop::String`

Take::String [Take]

```
i = gTake(i, num);

object  i;
int     num;
```

This method is used to change the length of the string associated with the `String` instance `i`. `num` is the new length of the string. If `num` is less than the previous length of the string, the string will be terminated to the shorter length. If `num` is greater than the previous length, the string will be padded with spaces until it is `num` characters long.

If `num` is negative, characters will be taken from the end of the string and padding will be performed at the beginning of the string.

The modified `String` instance passed is returned.

Example:

```
object  x, y;

x = gNewWithStr(String, "Hello World");
gTake(x, 4);   /*  x = "Hello"    */
x = gNewWithStr(String, "Hello World");
gTake(x, -4);   /*  x = "orld"    */
x = gNewWithStr(String, "ABC");
gTake(x, 5);   /*  x = "ABC  "    */
x = gNewWithStr(String, "ABC");
gTake(x, -5);   /*  x = "  ABC"    */
```

See also: `Drop::String`, `SubString::String`

`ToLower::String` [ToLower]

```
i = gToLower(i);

object  i;
```

This method is used to convert all the characters in `String` instance `i` to lower case. Non-alphabetic characters are left unchanged. The instance passed is returned.

Example:

```
object  x;

x = gNewWithStr(String, "Hello World");
gToLower(x);
/*  x = "hello world"  */
```

See also: `ToUpper::String`

`ToUpper::String` [ToUpper]

```
i = gToUpper(i);

object  i;
```

This method is used to convert all the characters in `String` instance `i` to upper case. Non-alphabetic characters are left unchanged. The instance passed is returned.

Example:

```
object  x;

x = gNewWithStr(String, "Hello World");
gToUpper(x);
/*  x = "HELLO WORLD"  */
```

See also: `ToLower::String`

5.49 StringAssociation Class

This class is a subclass of `Association` and is used to represent an association between a C language string and an arbitrary Dynace object. It performs all the functions as the `LookupKey` and `ObjectAssociation` classes and is used in conjunction with the `Set` and `Dictionary` classes. The purpose of this class is to provide an efficient mechanism to form a common association.

See the examples included with the Dynace system for an illustration of the use of the `Set/Dictionary` related classes.

5.49.1 StringAssociation Class Methods

The only class method associated with this class is one needed to create new instances of itself.

`NewWithStrObj::StringAssociation` [NewWithStrObj]

```
i = gNewWithStrObj(StringAssociation, key, val);

char    *key;
object  val;    /*  or NULL   */
object  i;
```

This class method creates instances of the `StringAssociation` class. `key` is used to initialize the key associated with the instance created and `val` is used to initialize its associated value. Note that internally `key` is copied into an allocated buffer so that it is all right to reuse the buffer passed to this method.

The new instance is returned.

5.49.2 StringAssociation Instance Methods

The instance methods associated with this class are used to obtain, change and print the key and value associated with the instance. Additional functionality, although implemented by this class, is documented in `StringAssociation`'s superclass, `Association`. This is done because of the common interface these methods share with all subclasses of `Association`.

`ChangeStringKey::StringAssociation` [ChangeStringKey]

```
i = gChangeStringKey(i, key);

object  i;
char    *key;
```

This method is used to change the key associated with instance `i`. `key` is the new key. The old key is simply replaced. This method returns `i`.

See also: `StringKey::StringAssociation`,
`ChangeValue::StringAssociation`

`ChangeValue::StringAssociation` [ChangeValue]

```
old = gChangeValue(i, val);

object  i;
object  val;    /*  or NULL   */
object  old;
```

This method is used to change the value associated with an association, `i`, to a value, `val`. The old value is simply no longer associated with the instance. It is not disposed. This method returns the old value being replaced.

See also: `Value::StringAssociation`, `ChangeStringKey::StringAssociation`

`Copy::StringAssociation` [Copy]

```
so = gCopy(i);

object  i, so;
```

This method is used to create a new instance of the `StringAssociation` class with the *same* value as `StringAssociation` `i`. Or put more simply, it makes a copy. The new `StringAssociation` instance is returned.

See also: `DeepCopy::StringAssociation`

`DeepCopy::StringAssociation` [DeepCopy]

```
so = gDeepCopy(i);

object  i, so;
```

This method is used to create a new instance of the `StringAssociation` class which contains a *copy* of the value from the original `StringAssociation`. `DeepCopy` is used to create the copy of the value. The new `StringAssociation` instance is returned.

See also: `Copy::StringAssociation`

`DeepDispose::StringAssociation` [DeepDispose]

```
r = gDeepDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to dispose of a `StringAssociation` instance. It also disposes of the value associated with the instance by the use of its `DeepDispose` method.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `Dispose::StringAssociation`

`Dispose::StringAssociation` [Dispose]

```
r = gDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to dispose of a `StringAssociation` instance. It does not dispose of its associated value.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `DeepDispose::StringAssociation`

`Key::StringAssociation` [Key]

```
key = gKey(i);

object  i;
object  key;
```

This method is used to obtain the key associated with instance `i`.

See also: `StringKey::StringAssociation`

`StringKey::StringAssociation` [StringKey]

```
key = gStringKey(i);

object  i;
char    *key;
```

This method is used to obtain the key associated with instance `i`.

See also: `Key::StringAssociation`,
`ChangeStringKey::StringAssociation`, `Value::StringAssociation`

`StringRepValue::StringAssociation` [StringRepValue]

```
s = gStringRepValue(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the value associated with `i`. This is often used to print or display the value. It is also used by `PrintValue::Object` and indirectly by `Print::Object` (two methods useful during the debugging phase of a project) in order to directly print an object's value.

See also: `PrintValue::Object`, `Print::Object`

`Value::StringAssociation` [Value]

```
val = gValue(i);

object  i;
object  val;
```

This method is used to obtain the value associated with `i`.

See also: `ChangeValue::StringAssociation`,
`StringKey::StringAssociation`

5.50 StringDictionary Class

This class combines the functionality of the `Set` and `StringAssociation` classes in order to store a collection of arbitrary key/value pairs.

The difference between this class and the `Dictionary` class is that this class only accepts C language strings as keys. This class provides an efficient and simple to use means of representing a commonly needed collection.

This class is a subclass of the `Set` class and therefore inherits all of its functionality.

See the examples included with the Dynace system for an illustration of the use of the `Set/Dictionary` related classes.

5.50.1 StringDictionary Class Methods

There are no class methods for this class. It inherits the ability to create instances of itself through its superclass, `Set`.

5.50.2 StringDictionary Instance Methods

The instance methods associated with this class are used to add, retrieve and remove key/value pairs from the `Dictionary`. Note that additional functionality may be obtained through its superclass, `Set`.

`AddStr::StringDictionary` [AddStr]

```
r = gAddStr(i, key, value);

object  i;
char    *key
object  value;
object  r;
```

This method is used to add a new key/value pair to the `StringDictionary` instance (`i`). If an object with the same key already exists in the `Dictionary` it will be left as is and `AddStr` will return `NULL`. If the key/value objects are added `AddStr` will return the `StringAssociation` instance created to represent the key/value pair passed.

See also: `FindStr::StringDictionary`, `FindValueStr::StringDictionary`, `ChangeValueWithStr::StringDictionary`, `RemoveStr::StringDictionary`

`ChangeValueWithStr::StringDictionary` [ChangeValueWithStr]

```
r = gChangeValueWithStr(i, key, value);

object  i;
char    *key
object  value;
object  r;
```

This method is used to change the value associated with an existing key/value pair to the `StringDictionary` instance (`i`). `key` represents the identity of the pre-existing key/value pair and `value` represents the new value to be associated with the key.

Normally, this method changes the value part of the association and returns the previous value which is not disposed. If `key` doesn't identify a pre-existing association, this method simply returns `NULL`.

See also: `FindStr::StringDictionary`, `FindValueStr::StringDictionary`, `RemoveStr::StringDictionary`

`DeepDisposeStr::StringDictionary` [DeepDisposeStr]

```
r = gDeepDisposeStr(i, key);

object  i;
char    *key;
object  r;
```

This method is used to remove and dispose of a key/value pair from a `Dictionary`. If found and removed `i` is returned. If `key` is not found `NULL` is returned.

The value and `StringAssociation` used to bind the two are all deep disposed.

This method is the same as `DisposeStr::StringDictionary`.

See also: `RemoveStr::StringDictionary`

`Dispose::StringDictionary` [Dispose]

```
r = gDispose(i);

object  i;
object  r;     /*  NULL  */
```

This method is used to dispose of an entire `Dictionary`. It does not dispose of any of the values but does dispose of all the associations used to represent the pair.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `RemoveStr::StringDictionary`, `DeepDispose::Set`, `DisposeAllNodes::StringDictionary`

`DisposeAllNodes::StringDictionary` [DisposeAllNodes]

```
i = gDisposeAllNodes(i);

object  i;
```

This method is used to remove all objects in a `StringDictionary` instance without disposing of the instance itself. The objects in the `StringDictionary` are simply disassociated from the `StringDictionary` instance and are not disposed.

The value returned is always the instance passed.

See also: `DeepDisposeAllNodes::Set`, `Dispose::StringDictionary`

`DisposeStr::StringDictionary` [DisposeStr]

```
r = gDisposeStr(i, key);

object  i;
char    *key;
object  r;
```

This method is used to remove and dispose of a key/value pair from a `Dictionary`. If found and removed `i` is returned. If `key` is not found `NULL` is returned.

The value and `StringAssociation` used to bind the two are all deep disposed.

This method is the same as `DeepDisposeStr::StringDictionary`.

See also: `RemoveStr::StringDictionary`

`FindStr::StringDictionary` [FindStr]

```
r = gFindStr(i, key);

object  i;
char    *key;
object  r;
```

This method is used to find the instance of the `StringAssociation` class which is used to represent the key/value pair stored under `key` in `Dictionary i`. If `key` is not found `NULL` is returned.

See also: `FindValueStr::StringDictionary`

`FindAddStr::StringDictionary` [FindAddStr]

```
r = gFindAddStr(i, key, value);

object  i;
char    *key;
object  value;
object  r;
```

This method is used to find and return the instance of the `StringAssociation` class used to represent the key/value pair stored under the key `key`. If it is not found a new `StringAssociation` will be added and returned which represent the key/value pair representing `key` and `value`.

See also: `FindStr::StringDictionary`, `FindValueStr::StringDictionary`

`FindValueStr::StringDictionary` [FindValueStr]

```
r = gFindValueStr(i, key);

object  i;
char    *key;
object  r;
```

This method is used to find the value stored under `key` in `Dictionary i`. If `key` is not found `NULL` is returned.

See also: `FindStr::StringDictionary`

`RemoveStr::StringDictionary` [RemoveStr]

```
r = gRemoveStr(i, key);

object  i;
char    *key;
object  r;
```

This method is used to remove a key/value pair from a `Dictionary`. If found and removed `i` is returned. If `key` is not found `NULL` is returned.

The value is not disposed, however, the `StringAssociation` used to bind the two is.

See also: `DeepDisposeStr::StringDictionary`

5.51 Thread Class

The `Thread` class gives Dynace the ability to create, manage and execute multiple threads of execution in a timesharing fashion. See the manual for a detailed description of threads.

See the examples included with the Dynace system for an illustration of the use of the thread / pipe / semaphore related classes.

5.51.1 Thread Class Methods

The class methods associated with the `Thread` class provide a means to create new instances of the `Thread` class. Each instance represents a unique thread of execution. The class methods also provide other functionality which is not related to any particular thread, such as the ability to initialize the threading system.

`BlockingGetkey::Thread` [BlockingGetkey]

```
c = gBlockingGetkey(Thread)

int     c;
```

This method is used to get a key from the keyboard. It does it in a way that allows other threads to continue if there is no key ready. This allows the system to continue to operate while waiting for a user entry.

The thread which calls this method will be placed on hold until a key has been hit. At that time the thread will be enabled and the method will return the key hit.

Only one thread may use this method at a time.

See also: `StartThreader`

`FindStr::Thread` [FindStr]

```
t = gFindStr(Thread, name)

char    *name;
object  t;
```

This method is used to obtain a thread object from its associated name. If found it is returned, otherwise `NULL` is returned. Note that the name of the thread which started the thread system is "main". If `name` is `NULL` `Find` returns the currently running thread object.

See also: `NewThread::Thread`

`NewThread::Thread` [NewThread]

```
t = gNewThread(Thread, name, fun, priority, arg, run,
               autoDispose)

char    *name;
int     (*fun)(void *);
int     priority;
void    *arg;
int     run;
int     autoDispose;
object  t;
```

This class method creates instances of the `Thread` class. Each thread represents a unique thread of execution.

`name` is the name of the thread. This name may be used to access the thread object via its name at a future point. `name` may be null if no particular name is desired.

`fun` is the C language function which will implement the process to be run by the new thread. When `fun` is first executed it is passed the pointer defined by `arg` as its only argument. `fun` may return an integer result which may be accessed by other threads.

`priority` sets the initial priority of the thread. This is a number between 1 and 30000. The higher the number the greater the priority. Threads of higher priority (which are ready to run) always run before any threads of lesser priority. Only ready threads of the same priority share CPU time. There is a macro named `DEFAULT_PRIORITY` which may be used to create threads which share resources equally. Creating higher or lower priority threads may be accomplished relative to the `DEFAULT_PRIORITY` macro. Note that *all* ready threads of a higher priority must complete before threads of a lower priority will receive any CPU time.

`arg` provides a simple mechanism to pass data to the thread when it starts. When `fun` is started it is passed a single argument `arg`.

`run` is a flag to tell the system how to initialize the thread. If `run` is 1 the new thread will be created in a ready-to-run state. It will be placed on the queue of ready threads and will automatically start when it gets CPU time. If the thread is started at a higher priority than the thread which created the new thread then the current thread will be timed-out and the new thread will immediately start. If `run` is set to 0 the new thread will be created in a hold state and must be released from this state in order to start running.

`autoDispose` is used to control what happens to the thread object once the thread completes. If set to 1 the thread will be automatically disposed when it completes. If set to 0 the thread will remain in the system until manually disposed. This may be useful if the return value of the thread is desired after it has completed. Note that the garbage collector never collects thread objects. They must be handled manually. Note also that the way the system was implemented completed threads and threads on hold take up 0 CPU resources.

The value returned (`t`) is the thread object created. If not kept it may be obtained later by the use of the `FindStr::Thread` method. This thread object may be used for many purposes to control a thread. You may hold, release, change priority, kill or otherwise query the state of the thread via this object.

Note that the threader must be started prior to the creation of any new threads. This may be accomplished by the use of the `StartThreader` method.

See also: `StartThreader, FindStr::Thread`

5.51.2 Thread Instance Methods

The instance methods associated with this class are used to manage and query various aspects related to particular threads. Each instance represents a single unique thread.

`DeepDispose::Thread` [DeepDispose]

```
r = gDeepDispose(t)

object  t;
object  r;     /*  NULL  */
```

This method is used to permanently stop the execution of an arbitrary thread, `t` (if it is running) and dispose of it. `t` may be the current thread, in which case the next thread on the queue will start.

In order to give the thread a return value use `Kill::Thread` and then this method.

This method is the same as `Dispose::Thread`.

See also: `Kill::Thread, Hold::Thread`

`Dispose::Thread` [Dispose]

```
r = gDispose(t)

object  t;
object  r;     /*  NULL  */
```

This method is used to permanently stop the execution of an arbitrary thread, `t` (if it is running) and dispose of it. `t` may be the current thread, in which case the next thread on the queue will start.

In order to give the thread a return value use `Kill::Thread` and then this method.

This method is the same as `DeepDispose::Thread`.

The value returned is always `NULL` and may be used to null out the variable which contained the object being disposed in order to avoid future accidental use.

See also: `Kill::Thread`, `Hold::Thread`

`ChangePriority::Thread` [ChangePriority]

```
t = gChangePriority(t, pri)

object  t;
int     pri;
```

This method is used to change the priority of thread `t`. Thread `t` may be the currently running thread. If thread `t` is changed to a higher priority than the currently running thread, the currently running will immediately release the CPU to thread `t`.

The priority is a number between 1 and 30000. The higher the number the greater the priority. Threads of higher priority (which are ready to run) always run before any threads of lesser priority. Only ready threads of the same priority share CPU time. There is a macro named `DEFAULT_PRIORITY` which may be used to create threads which share resources equally. Creating higher or lower priority threads may be accomplished relative to the `DEFAULT_PRIORITY` macro. Note that *all* ready threads of a higher priority must complete before threads of a lower priority will receive any CPU time.

See also: `Priority::Thread`, `NewThread::Thread`, `State::Thread`

`Hold::Thread` [Hold]

```
t = gHold(t)

object  t;
```

This method is used to place a temporary hold on a thread (`t`). The thread stops execution and waits until released. Threads on hold take no CPU time. Any number of threads may place any number of holds on any threads, however, an equal number of releases are required to release a thread which has been held multiple times. A thread may put itself on hold. The value returned is the thread passed.

See also: `Release::Thread`, `Kill::Thread`

`IntValue::Thread` [IntValue]

```
r = gIntValue(t)

object  t;
int     r;
```

This method is used to obtain the return value of a thread (`t`) which has completed or been killed. If the thread is still running (or on hold) this method will just return 0.

See also: `State::Thread, WaitFor::Thread, Kill::Thread`

`Kill::Thread` [Kill]

```
t = gKill(t, rtn)

object  t;
int     rtn;
```

This method is used to permanently stop the execution of an arbitrary thread, `t`. Once this method is called on a thread it cannot be run again. `t` may be the current thread, in which case the next thread on the queue will start. Thread `t` is not disposed. Attempting to `Kill` a thread which has already been killed or completed has no ill effect, `rtn` is just ignored.

`rtn` represents the return value of the thread, which may be queried by other threads. Any holds associated with the thread will be expunged. Thread `t` is returned.

Note that killed threads take up no CPU time.

See also: `Dispose::Thread, Hold::Thread`

`Name::Thread` [Name]

```
nam = gName(t)

object  t;
char    *nam;
```

This method is used to obtain the name of thread `t`.

See also: `NewThread::Thread, State::Thread, Priority::Thread`

`Priority::Thread` [Priority]

```
pri = gPriority(t)

object  t;
int     pri;
```

This method is used to obtain the priority of thread `t`.

The priority is a number between 1 and 30000. The higher the number the greater the priority. Threads of higher priority (which are ready to run) always run before any threads of lesser priority. Only ready threads of the same priority share CPU time. There is a macro named `DEFAULT_PRIORITY` which may be used to create threads which share resources equally. Creating higher or lower priority threads may be accomplished relative to the `DEFAULT_PRIORITY` macro. Note that *all* ready threads of a higher priority must complete before threads of a lower priority will receive any CPU time.

See also: `ChangePriority::Thread, NewThread::Thread, State::Thread`

`Release::Thread` [Release]

```
t = gRelease(t, yld)

object  t;
int     yld;
```

This method is used to release a hold placed by the `Hold` method on a thread (`t`). The thread will not really be released until the number of calls to `Release` is equal to the number of times `Hold` was called on thread `t`.

`yld` is a flag. It indicates whether the current thread should yield to the thread being released (if the thread being released has a higher priority). If `yld` is 1 and the thread being released has a higher priority than the current thread the system will immediately transfer the CPU to the thread being released. If `yld` is set to 0 the current thread will be allowed to complete its time slice. The value returned is the thread passed.

See also: `Hold::Thread`

`State::Thread` [State]

```
s = gState(t)

object  t;
int     s;
```

This method is used to obtain the current state of a thread (`t`). The valid states are macros defined in `dynl.h` and are defined as follows:

```
NEW_THREAD              New thread which hasn't started running
                        yet
RUNNING_THREAD          Thread ready to run
HOLD_THREAD             Thread placed on hold
DONE_THREAD             Thread has completed or been killed
WAITING_FOR_THREAD      Thread waiting for another thread to
                        complete
WAITING_FOR_SEMAPHORE   Thread waiting for semaphore
```

`WaitFor::Thread` [WaitFor]

```
r = gWaitFor(t)

object  t;
int     r;
```

This method is used to place the current thread on hold until thread `t` completes, gets killed or disposed. At that point the thread gets released and `WaitFor` returns the return value from thread `t`. Note that threads waiting for other threads take up no CPU time.

See also: `Kill::Thread`, `Dispose::Thread`, `Hold::Thread`

5.51.3 Thread Macros

The thread system includes a few macros which are used to provide some additional functionality without incurring much system overhead.

`ENABLE_THREADER` [ENABLE_THREADER]

```
ENABLE_THREADER;
```

This macro is used to re-enable the threader after the `INHIBIT_THREADER` has been used. Since consecutive calls to `INHIBIT_THREADER` stack, an equal number of `ENABLE_THREADER` calls must be used.

The two macros, `INHIBIT_THREADER` and `ENABLE_THREADER`, are used to surround code which you want to be absolutely certain that a context switch doesn't occur during. Since these macros nest it is ok if between a pair a function is called which also uses these macros.

Note that when the threader is enabled a context switch can only occur during a generic function call or by the use of the `YIELD` macro.

Example:

```
INHIBIT_THREADER;
        .
  (protected code)
        .
ENABLE_THREADER;
```

See also: `INHIBIT_THREADER, StartThreader`

`INHIBIT_THREADER` [INHIBIT_THREADER]

```
INHIBIT_THREADER;
```

This macro is used to temporarily disable the threader. Once disabled no context switches will occur until the threader is re-enabled by the use of the `ENABLE_THREADER` macro. Consecutive calls to `INHIBIT_THREADER` stack, therefore, an equal number of `ENABLE_THREADER` calls must be used to re-enable the threader.

The two macros, `INHIBIT_THREADER` and `ENABLE_THREADER`, are used to surround code which you want to be absolutely certain that a context switch doesn't occur during. Since these macros nest it is ok if between a pair a function is called which also uses these macros.

Note that when the threader is enabled a context switch can only occur during a generic function call or by the use of the `YIELD` macro.

Example:

```
INHIBIT_THREADER;
        .
  (protected code)
        .
ENABLE_THREADER;
```

See also: `ENABLE_THREADER, StartThreader`

`StartThreader` [StartThreader]

```
StartThreader(argc)

int     argc;
```

This macro is used to initialize the threading system. It must be called before any other thread operations (except for initializing the thread class). It *must* be called from the C function called `main`, and should only be called once. It is typically called immediately following the Dynace system initialization.

Once this macro is called all execution is performed by threads. The program (`main`) which called `StartThreader` becomes the single running thread with the name "main". Any future threads created at the default priority will share CPU time equally with the main thread. There is nothing special about the main thread.

`argc` is simply the argument normally passed to C language `main` functions. The value doesn't matter. It is used to obtain the address of the stack.

See also: `NewThread::Thread`

`YIELD` [YIELD]

```
YIELD;
```

The `YIELD` macro is used to force a context switch (unless the threader has been disabled with the `INHIBIT_THREADER` macro).

Since Dynace only provides automatic context switching during a generic function call, if a thread is running a long compute cycle without any generic function calls it will hog the system until a generic is called unless the `YIELD` macro is used. This macro may be placed at regular intervals in the code to be sure that it shares the CPU time. The `YIELD` macro will only cause an actual context switch when the threads time is complete, otherwise `YIELD` will do nothing.

Example:

```
while (some condition)  {
        do something;
        YIELD;
}
```

See also: `StartThreader`

5.52 Time Class

The `Time` class uses the same set of instance and class methods as the `LongInteger` class. The equivalent C language representation would be a long integer in the form `HHMMSSLLL` so 1:23:45.678 pm would be represented as 132345678L.

5.52.1 Time Class Methods

The `Time` class has two class methods to create instances.

`NewWithLong::Time` [NewWithLong]

```
i = gNewWithLong(Time, tm);

long    tm;
object  i;
```

This class method creates instances of the `Time` class. `tm` represents the initial time value represented.

Example:

```
object  x;

x = gNewWithLong(Time, 132345678L);  /* x = 1:23:45.678 pm  */
```

See also: `Now::Time`, `FormatTime::Time`, `Dispose::Object`

`Now::Time` [Now]

```
i = gNow(Time);

object  i;
```

This class method creates instances of the `Time` class which represents the current time contained within the system.

Example:

```
object  x;

x = gNow(Time);   /*  x = the current time  */
```

See also: `NewWithLong::Time`, `FormatTime::Time`, `Dispose::Object`

5.52.2 Time Instance Methods

A portion of the `Time` class's functionality is identical to the functionality of the `LongInteger` class. The remaining functionality is defined by the following specific methods.

`AddHours::Time` [AddHours]

```
i = gAddHours(i, hours);

object  i;
long    hours;
```

This method is used to add an arbitrary number of hours (`hours`) to time `i`. Adding an hour that goes past midnight will cause the time to cycle around to morning. The value returned is the object passed.

Example:

```
object  tm;

tm = gNewWithLong(Time, 234500000L);
gAddHours(tm, 2L);  /* tm = 14500000L (1:45:00.000 am) */
```

See also: `AddMilliseconds::Time`, `AddMinutes::Time`, `AddSeconds::Time`

`AddMilliseconds::Time` [AddMilliseconds]

```
i = gAddMilliseconds(i, m);

object  i;
long    m;
```

This method is used to add an arbitrary number of milliseconds (`m`) to time `i`. Adding a millisecond that goes past midnight will cause the time to cycle around to morning. The value returned is the object passed.

Example:

```
object  tm;

tm = gNewWithLong(Time, 234500000L);
gAddMilliseconds(tm, 20L);
    /* tm = 500020L (11:45:00.020 pm) */
```

See also: `AddHours::Time`, `AddMinutes::Time`, `AddSeconds::Time`

`AddMinutes::Time` [AddMinutes]

```
i = gAddMinutes(i, m);

object  i;
long    m;
```

This method is used to add an arbitrary number of minutes (`m`) to time `i`. Adding a minute that goes past midnight will cause the time to cycle around to morning. The value returned is the object passed.

Example:

```
object  tm;

tm = gNewWithLong(Time, 234500000L);
gAddMinutes(tm, 20L);  /* tm = 500000L (1:05:00.000 am) */
```

See also: `AddHours::Time`, `AddMilliseconds::Time`, `AddSeconds::Time`

`AddSeconds::Time` [AddSeconds]

```
i = gAddSeconds(i, s);

object  i;
long    s;
```

This method is used to add an arbitrary number of seconds (`s`) to time `i`. Adding a second that goes past midnight will cause the time to cycle around to morning. The value returned is the object passed.

Example:

```
object  dt;

tm = gNewWithLong(Time, 234500000L);
gAddSeconds(tm, 20L);  /* tm = 520000L (11:45:20.000 pm) */
```

See also: `AddHours::Time`, `AddMilliseconds::Time`, `AddMinutes::Time`

`ChangeLongValue::Time` [ChangeLongValue]

```
i = gChangeLongValue(i, val);

object  i;
long    val;
```

This method is used to change the time value associated with an instance of the `Time` class. Notice that this method returns the instance being passed. `val` is the new time value.

Example:

```
object  tm;

tm = gNewWithLong(Time, 234500000L);
gChangeLongValue(tm, 44500000L);   /*  tm = 4:45:00.000 am  */
```

See also: `ChangeTimeValue::Time, NewWithLong::Time`

`ChangeTimeValue::Time` [ChangeTimeValue]

```
i = gChangeTimeValue(i, val);

object  i;
long    val;
```

This method is used to change the time value associated with an instance of the `Time` class. Notice that this method returns the instance being passed. `val` is the new time value.

Example:

```
object  tm;

tm = gNewWithLong(Time, 234500000L);
gChangeTimeValue(tm, 44500000L);   /*  tm = 4:45:00.000 am  */
```

See also: `ChangeLongValue::Time, NewWithLong::Time`

`ChangeValue::Time` [ChangeValue]

```
i = gChangeValue(i, v);

object  i;
object  v;
```

This method is used to change the value associated with an instance of the `Time` class. Notice that this method returns the instance being passed. `v` is an object which represents the new time value.

Example:

```
object  x, y;

x = gNewWithLong(Time, 234500000L);
y = gNewWithLong(Time, 44500000L);
gChangeValue(x, y);    /*  x is changed to 4:45:00.000 am  */
```

See also: `ChangeLongValue::Time`, `ChangeTimeValue::Time`

`Compare::Time` [Compare]

```
r = gCompare(i, obj);

object  i;
object  obj;
int     r;
```

This method is used by the generic container classes to determine the relationship of the values represented by `i` and `obj`. `r` is -1 if the time represented by `i` is less than the time represented by `obj`, 1 if the time value of `i` is greater than `obj`, and 0 if they are equal.

See also: `Hash::Time`

`Difference::Time` [Difference]

```
r = gDifference(i, tm);

object  i;
object  tm;
long    r;
```

This method is used to obtain the difference, in milliseconds, between two time objects.

Example:

```
object  tm1, tm2;
long    r;

tm1 = gNewWithLong(Time, 234515023L);
tm2 = gNewWithLong(Time, 234515000L);
r = gDifference(dt2, dt1);    /*  r = 23L  */
```

See also: `TimeDifference::Time`

`FormatTime::Time` [FormatTime]

```
s = gFormatTime(i, fmt);

object  i;
char    *fmt;
object  s;
```

This method returns an instance of the `String` class which is a formatted representation of the time associated with instance `i`. `fmt` is the format specification used to determine the resulting `String` object. Each character in `fmt` is sequentially processed and except for the character '%', they are simply copied to the resulting `String` object. Whenever this method encounters the '%' character, the character following is used to determine what aspect and format of the time represented by `i` should be added to the resultant `String` object. It works much like the standard C `printf`. The following table indicates the available formatting option characters:

```
%       the % character
g       hour in military (government) time.
        Eleven o'clock is 23:00 (5)
G       the hour to two places in military (government)
        time (05)
h       hour (5)
H       the hour to two places (05)
m       minutes (5)
M       the minutes to two places (05)
s       seconds (5)
S       the seconds to two places (05)
l       milliseconds (20)
L       the milliseconds to three places (020)
p       the am / pm flag in lower case (pm)
p       the am / pm flag in upper case (PM)
```

Example:

```
object  s, tm;

tm = gNow(Time);
s = gFormatTime(tm, "%h:%M:%S.%L %p");
/*  s contains "5:45:23.678 pm" (or whatever)  */
```

See also: `StringRepValue::Time`

Hash::Time [Hash]

```
val = gHash(i);

object  i;
int     val;
```

This method is used by the generic container classes to obtain hash values for the time object. `val` is a hash value between 0 and a large integer value.

See also: `Compare::Time`

Hours::Time [Hours]

```
h = gHours(i);

object  i;
int     h;
```

This method returns the number of hours associated with an instance of the `Time` class.

Example:

```
object  tm;

tm = gToday(Time);
h = gHours(tm);   /*  h contains 14 (or whatever)  */
```

See also: `Milliseconds::Time`, `Minutes::Time`, `Seconds::Time`

LongValue::Time [LongValue]

```
val = gLongValue(i);

object  i;
long    val;
```

This method is used to obtain the `long` value that represents the time associated with an instance the `Time` class. Note that this is one of the few generics which doesn't return a Dynace object. It returns a `long`.

Example:

```
object  x;
long    val;

x = gNewWithLong(LongInteger, 234500000L);
val = gLongValue(x);    /*  val = 234500000L  */
```

See also: `NewWithLong::Time`, `ChangeLongValue::Time`, `TimeValue::Time`

`Milliseconds::Time` [Milliseconds]

```
m = gMilliseconds(i);

object  i;
int     m;
```

This method returns the number of milliseconds associated with an instance of the `Time` class.

Example:

```
object  tm;

tm = gToday(Time);
m = gMilliseconds(tm);   /*  m contains 154 (or whatever)  */
```

See also: `Hours::Time`, `Minutes::Time`, `Seconds::Time`

`Minutes::Time` [Minutes]

```
m = gMinutes(i);

object  i;
int     m;
```

This method returns the number of minutes associated with an instance of the `Time` class.

Example:

```
object  tm;

tm = gToday(Time);
m = gMinutes(tm);   /*  h contains 55 (or whatever)  */
```

See also: `Hours::Time`, `Milliseconds::Time`, `Seconds::Time`

`Seconds::Time` [Seconds]

```
m = gSeconds(i);

object  i;
int     s;
```

This method returns the number of seconds associated with an instance of the `Time` class.

Example:

```
object  tm;

tm = gToday(Time);
s = gSeconds(tm);   /*  s contains 30 (or whatever)  */
```

See also: `Hours::Time`, `Milliseconds::Time`, `Minutes::Time`

`StringRepValue::Time` [StringRepValue]

```
s = gStringRepValue(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the time associated with `i`. This is often used to print or display the time. It is also used by `PrintValue::Object` and indirectly by `Print::Object` (two methods useful during the debugging phase of a project) in order to directly print an object's value.

Example:

```
object  x;
object  s;

x = gNewWithLong(Time, 234500000L);
s = gStringRepValue(x);
   /*  s represents "11:45:00.000 pm"  */
```

See also: `PrintValue::Object`, `Print::Object`, `FormatTime::Time`, `TimeStringRepValue::Time`

`TimeDifference::Time` [TimeDifference]

```
r = gTmieDifference(i, tm);

object  i;
object  tm;
long    r;
```

This method is used to obtain the difference, in milliseconds, between two time objects.

Example:

```
object  tm1, tm2;
long    r;

tm1 = gNewWithLong(Time, 234515023L);
tm2 = gNewWithLong(Time, 234515000L);
r = gTimeDifference(dt2, dt1);    /*  r = 23L  */
```

See also: `Difference::Time`

`TimeStringRepValue::Time` [TimeStringRepValue]

```
s = gTimeStringRepValue(i);

object  i;
object  s;
```

This method is used to generate an instance of the `String` class which represents the time associated with `i`. This is often used to print or display the time.

Example:

```
object  x;
object  s;

x = gNewWithLong(Time, 234500000L);
s = gTimeStringRepValue(x);
    /* s represents "11:45:00.000 pm"  */
```

See also: `PrintValue::Object`, `Print::Object`, `FormatTime::Time`, `StringRepValue::Time`

`TimeValue::Time` [TimeValue]

```
val = gTimeValue(i);

object  i;
long    val;
```

This method is used to obtain the `long` value that represents the time associated with an instance the `Time` class. Note that this is one of the few generics which doesn't return a Dynace object. It returns a `long`.

Example:

```
object  x;
long    val;

x = gNewWithLong(LongInteger, 234500000L);
val = gTimeValue(x);    /*  val = 234500000L  */
```

See also: `NewWithLong::Time`, `ChangeLongValue::Time`, `LongValue::Time`

`ValidTime::Time` [ValidTime]

```
r = gValidTime(i);

object  i;
int     r;
```

This method is used to determine the validity of a time. If `i` is a valid time 1 is returned and 0 otherwise.

Example:

```
object  tm;
int     r;

tm = gNewWithLong(Time, 234515899L);
r = gValidTime(tm);   /*  r = 1  */
tm = gNewWithLong(Time, 254500899L);
r = gValidTime(tm);   /*  r = 0  */
tm = gNewWithLong(Time, 114567000L);
r = gValidTime(tm);   /*  r = 0  */
```

5.53 UnsignedShortArray Class

This class, which is a subclass of `NumberArray`, is used to represent arbitrary shaped arrays of the C language `unsigned short` data type in an efficient manner. Although this class implements much of its own functionality it is documented within the `NumberArray` and `Array` classes because the interface is shared by all subclasses of the `NumberArray` class.

5.54 UnsignedShortInteger Class

The `UnsignedShortInteger` class is used to represent the C language `unsigned short` data type as a Dynace object. It is a subclass of the `Number` class. Even though the `UnsignedShortInteger` class implements most of its own functionality, it is documented as part of the `Number` class because most of the interface is the same for all subclasses of the `Number` class. Differences are documented in this section.

5.54.1 UnsignedShortInteger Class Methods

The `UnsignedShortInteger` class has only one class method and it is used to create new instances of itself.

`NewWithUnsigned::UnsignedShortInteger` [NewWithUnsigned]

```
i = gNewWithUnsigned(UnsignedShortInteger, s);

unsigned s;
object  i;
```

This class method creates instances of the `UnsignedShortInteger` class. `s` is the initial value of the integer being represented. Note that `s` is of type `unsigned`. That is because if you pass an unsigned short type to a generic function (which uses a variable argument declaration) the C language will automatically promote it to an `unsigned`.

The value returned is a Dynace instance object which represents the integer passed.

Note that the default disposal methods are used by this class since there are no special storage allocation requirements.

Example:

```
object  x;

x = gNewWithUnsigned(UnsignedShortInteger, 55);
```

See also: `UnsignedShortValue::Number`, `Dispose::Object`

5.54.2 UnsignedShortInteger Instance Methods

The instance methods associated with the `UnsignedShortInteger` class provide a means of changing and obtaining the value associated with an instance of the `UnsignedShortInteger` class. All of the `UnsignedShortInteger` class instance methods are documented in the `Number` class because of their common interface with all other subclasses of the `Number` class.

Method, Macro, Function And Variable Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Y