INTRODUCTION

These are the release notes for Revision 3.0 of the NIH Class Library, the version of the library described by our book *Data Abstraction and Object-Oriented Programming in C++* by Keith E. Gorlen, Sanford M. Orlow, and Perry S. Plexico (ISBN 0471 92346 X), published by John Wiley and Sons.

This release of the NIH Class Library contains the following classes:

- **NIHCL**---Library Static Member Variables and Functions
  - **Object**---Root of the NIH Class Library Inheritance Tree
    - **Bitset**---Set of Small Integers (like Pascal's type SET)
    - **Class**---Class Descriptor
    - **Collection**---Abstract Class for Collections
      - **Arraychar**---Byte Array
      - **ArrayOb**---Array of Object Pointers
      - **Bag**---Unordered Collection of Objects
      - **SeqCltn**---Abstract Class for Ordered, Indexed Collections
        - **Heap**---Min-Max Heap of Object Pointers
        - **LinkedList**---Singly-Linked List
        - **OrderedCltn**---Ordered Collection of Object Pointers
          - **SortedCltn**---Sorted Collection
            - **KeySortCltn**---Keyed Sorted Collection
        - **Stack**---Stack of Object Pointers
      - **Set**---Unordered Collection of Non-Duplicate Objects
        - **Dictionary**---Set of Associations
          - **IdentDict**---Dictionary Keyed by Object Address
        - **IdentSet**---Set Keyed by Object Address
    - **Date**---Gregorian Calendar Date
    - **FDSet**---Set of File Descriptors for Use with `select(2)` System Call
    - **Float**---Floating Point Number
    - **Fraction**---Rational Arithmetic
    - **Integer**---Integer Number Object
    - **Iterator**---Collection Iterator
    - **Link**---Abstract Class for LinkedList Links
      - **LinkOb**---Link Containing Object Pointer
      - **Process**---Co-routine Process Object
        - **HeapProc**---Process with Stack in Free Store
        - **StackProc**---Process with Stack on `main()` Stack
    - **LookupKey**---Abstract Class for Dictionary Associations
      - **Assoc**---Association of Object Pointers
      - **AssocInt**---Association of Object Pointer with Integer
    - **Nil**---The Nil Object
    - **Point**---X-Y Coordinate Pair
    - **Random**---Random Number Generator
    - **Range**---Range of Integers

- **Rectangle**---Rectangle Object
- **Scheduler**---Co-routine Process Scheduler
- **Semaphore**---Process Synchronization
- **SharedQueue**---Shared Queue of Objects
- **String**---Character String
  - **Regex**---Regular Expression
- **Time**---Time of Day
- **Vector**---Abstract Class for Vectors
  - **BitVec**---Bit Vector
  - **ByteVec**---Byte Vector
  - **ShortVec**---Short Integer Vector
  - **IntVec**---Integer Vector
  - **LongVec**---Long Integer Vector
  - **FloatVec**---Floating Point Vector
  - **DoubleVec**---Double-Precision Floating Point Vector

OIOifd---File Descriptor Object I/O `readFrom()` Formatting
OIOin---Abstract Class for Object I/O `readFrom()` Formatting
- **OIOistream**---Abstract Class for Stream Object I/O `readFrom()` Formatting
  - **OIOnihin**---Stream Object I/O `readFrom()` Formatting

OIOofd---File Descriptor Object I/O `storeOn()` Formatting
OIOout---Abstract Class for Object I/O `storeOn()` Formatting
- **OIOostream**---Abstract Class for Stream Object I/O `storeOn()` Formatting
  - **OIOnihout**---Stream Object I/O `storeOn()` Formatting

ReadFromTbl---Tables used by Object I/O `readFrom()`
StoreOnTbl---Tables used by Object I/O `storeOn()`

CHANGES BETWEEN OOPS V2R2 AND NIHCL R3.0

This section highlights the most significant changes that have been made since the previous release. It is by no stretch of the imagination complete.

Library name changed from "OOPS" to "NIH Class Library" (NIHCL)

Since there's too many things called "OOPS" these days, we've changed the name of our library to the "NIH Class Library". All file and C++ names containing "OOPS" have been changed, often just by substituting "NIHCL" for "OOPS".

Class NIHCL

With the introduction of static member functions and (useful) static member variables in R2.0, it is now possible to eliminate most global names. In R3.0, we've gathered many previously global functions and variables and made them static members of a new class, `NIHCL`, which is the base class of `Object`. Here's a list of the public functions:

```
static class NIHCL {
// ...
```

```
public:             // static member functions
    static unsigned char charBitMask(int i);
    static unsigned short shortBitMask(int i);
    static unsigned int intBitMask(int i);
    static unsigned char bitCount(int i);
    static unsigned char bitReverse(int i);
    static void initialize();          // library initialization
    static bool initialized();         // library initialized?
    static void setError(int error, int sev ...);
} NIHCL_init;
```

Since all NIH Library classes inherit these members, their member functions can use these names without needing to specify a scope qualifier, except to resolve ambiguities. However, non-member functions and member functions of classes not derived from `NIHCL` can access them with the `NIHCL` scope qualifier; for example:

```
NIHCL::setError(ERROR_CODE, DEFAULT);
```

Optional support for multiple inheritance

The NIH Class Library can be compiled to support Multiple Inheritance (MI) by defining the preprocessor symbol `MI`. *All classes linked together in a program must all have been compiled with the same* `MI` *option setting*. The major effect of this switch is that all classes derived from `Object` specify it as a virtual base class. Since C++ does not permit a pointer to a virtual base class to be cast down to a pointer to a derived class, the new `DECLARE_MEMBERS` macro defines an overloaded family of static member functions named `castDown()` that can perform this conversion. (If `MI` is not enabled, `castDown()` becomes an ordinary pointer cast.)

The `castDown()` functions all call the function `_castDown()` to perform the pointer conversion. If a class has only a single base class, it uses the `DEFINE_CLASS` and `DEFINE_ABSTRACT_CLASS` preprocessor macros as before, and these generate an implementation of `_castDown()` suitable for the single inheritance case. If a class has multiple base classes, it uses the new `DEFINE_CLASS_MI` and `DEFINE_ABSTRACT_CLASS_MI` macros, which do *not* generate `_castDown()`---the class provider must supply a definition as described in `Template_c`.

All `readFrom()` constructors must specify the `readFrom()` constructor for the virtual base class `Object` in their initialization lists when `MI` is enabled. See `Template_h` for details.

If you use virtual base classes in conjunction with the NIH Class Library, you must take care when implementing the `deepCopy()` and `storeOn()` operations that a virtual base class's member variables are only deepened or stored once. The library provides the functions `deepenVBase()` and `storeVBaseOn()` to help with this. Call `deepenVBase()` instead of `deepenShallowCopy()` to deepen the member variables of a virtual base class, and call `storeVBaseOn()` instead of `storer()` to store the member variables of a virtual base class.

`DECLARE_MEMBERS` macro

The new DECLARE_MEMBERS preprocessor macro generates the declarations for the class descriptor and most of the member functions that all NIH Library classes must provide, including:

```
private:
        static Class classDesc;         // class descriptor
public:
        classname* castDown(Object*);
        const classname* castDown(const Object*);
        classname& castDown(Object&);
        const classname& castDown(const Object&);
        static const Class* desc();             // return class
descriptor
        static classname* readFrom(OIOin&);
        static classname* readFrom(OIOifd&);
        classname(OIOin&);                      // readFrom()
constructors
        classname(OIOout&);
        virtual const Class* isA() const;
        virtual Object* shallowCopy() const;
        virtual void* _castdown(const Class&) const;
protected:
   void deepenVBase();
        void storeVBaseOn(OIOofd&) const;       // store virtual base
class
        void storeVBaseOn(OIOout&) const;
private:
        static Object* reader(OIOin& strm);
        static Object* reader(OIOifd& fd);
```

The DECLARE_MEMBERS macro takes a single argument, the name of the class being declared.

New implementation of Process classes

In R3.0, class `Process` has two derived classes, `StackProc` and `HeapProc`, which serve as the base classes for client processes. `StackProc` and `HeapProc` differ in where a process's stack is located when the process is running: a `StackProc` has its stack located on the real stack in the stack segment, while a `HeapProc` has its stack in the free storage area in the data segment. A context switch of a `StackProc` involves copying the active part of the current process's stack into a save area, then copying the saved stack of the new process onto the real stack. A context switch of a `HeapProc` involves simply resetting the processor's stack pointer and frame pointer registers to point to the new stack, so a `StackProc` context switch is much slower than a `HeapProc` context switch. However, using `HeapProc`s tends to break debuggers, which usually can't cope with the bizarre stack location, so programs using them are difficult to debug. Also, the stack area for a `HeapProc` must be specified when it is constructed, and must be large enough to hold the largest stack that can occur anytime during execution. In contrast, the stack save area for a `StackProc` grows in size if necessary and must only be large enough to hold the largest stack in use when the process is suspended. Thus, the tradeoff is debuggability and reduced memory requirement vs. speed.

Fiddling with the stack area and machine registers is something you can't do directly from C++, so processes are inherently non-portable. The R3.0 implementation of the `Process` classes attempts to use the C library routines `setjmp()`, `longjmp()`, and `alloca()` to do context switching. While this works on many machines, you may need to write your own versions of these routines for machines on which it doesn't. See the section on *PORTING THE PROCESS CLASSES* for instructions.

External `class_`*classname* identifiers eliminated

The class descriptor for each class no longer has an external identifier of the form `class_`*classname*. Instead, the class descriptor is a static member variable of each class, and the inline static member function `desc()` returns its address.

Changes to Object I/O

Previous releases of OOPS have a fundamental problem in the way they handle object I/O for classes with member variables that are class instances. For example, consider an OOPS class `X` with a member variable of class `M`, where `M` is also an OOPS class:

```
class X : public BASE {
    M m;    // a member class instance
    M* p;   // a member pointer to a class instance
    int i;  // a fundamental type
// ...
};
```

Previous releases implement `X::storer()` and `X::X(istream&,X&)` as follows:

```
void X::storer(ostream& strm)
{
    BASE::storer(strm);
    m.storeOn(strm);
    p->storeOn(strm);
    strm << i << " ";
}

void X::X(istream&strm, X& where)
    : (strm,where)
```

```
{
    this = &where;
    readFrom(strm,"M",m);
    p = (M*)readFrom(strm,"M");
    strm >> i;
}

void X::storer(FileDescTy& fd)
{
    BASE::storer(fd);
    m.storeOn(fd);
    p->storeOn(fd);
    storeBin(fd,i);
}

void X::X(FileDescTy& fd, X& where)
    : (fd,where)
{
    this = &where;
    readFrom(fd,"M",m);
    p = (M*)readFrom(fd,"M");
    readBin(fd,i);
}
```

The problem is that this constructor first initializes `m` using the `M::M()` constructor, then calls `readFrom()`, which overwrites this initialized instance with an instance constructed by reading `strm`. We didn't notice this bug earlier because, in practice, the problem occurs only in classes `Rectangle` and `SharedQueue`, and has no obvious consequences. The worst that is likely to happen is that `M::M()` allocates some memory that never gets reclaimed.

Unfortunately, the fix requires some widespread changes. But, it turns out that numerous other improvements become possible. The new format for `storer()` functions and `readFrom()` constructors when not using MI is:

```
void X::storer(OIOout& strm)
{
    BASE::storer(strm);
    m.storeMemberOn(strm);
    p->storeOn(strm);
    strm << i;
}

void X::X(OIOin& strm)
    : (strm), m(strm)
{
    p = M::readFrom(strm);
```

```
        strm >> i;
    }

    void X::storer(OIOofd& fd)
    {
        BASE::storer(fd);
        m.storeMemberOn(fd);
        p->storeOn(fd);
        fd << i;
    }

    void X::X(OIOifd& fd)
        : (fd), m(fd)
    {
        p = M::readFrom(fd);
        fd >> i;
    }
```

The new format is simpler and consistent---`storer()` functions always call `BASE::storer()` and `X::X(OIOin&)` constructors always call `BASE::BASE(OIOin&)`.

Public `readFrom()` constructor

Since other classes in general must have access to `X::X(OIOin&)`, it must be public instead of protected, requiring a change to all header files.

Changes to `readFrom()`

As explained previously, calls to `readFrom()` that specify the third argument overwrite an initialized instance of a class. Since this is generally a bad thing to do, the third argument to `readFrom()` has been eliminated; thus, `readFrom()` will only return a pointer to an object. This form of `readFrom()` was used to initialize member class instances. These must now be initialized via the `readFrom()` constructor in the initializer list, as shown in the example. When converting old programs, these calls to `readFrom()` will be flagged as errors because the three argument form of `readFrom()` is no longer defined. *Note that when you change the `readFrom()` constructor, you must also change the corresponding `storer()` function to store the member using `storeMemberOn()` rather than `storeOn()`.*

In previous releases, the second argument to `readFrom()` was an optional name of the class of object that was expected to be read. If the object read was of a different class, `readFrom()` raised an error. Beginning with this release, `readFrom(OIOin&)`, and `readFrom(OIOifd&)` are static member functions of each class which will also accept derived classes of the specified class, just as C++ allows a pointer to a derived class to be used instead of a pointer to a base class. The global functions `readFrom(istream, const char* classname)` and `readFrom(int fd, const char*` `classname)` have been eliminated.

`istream&` replaced by `OIOin&`

The type of the first (and now only) argument to the constructors called by `readFrom(istream&)` has been changed from an `istream&` to an `OIOin&`. This avoids naming conflicts with other constructors. Also, `OIOin` is an abstract base class, and all input operators are virtual functions, so you can customize the Object I/O format by defining your own derived classes. `OIOistream` and `OIOnihin` implement a format similar to the old OOPS format.

`ostream&` replaced by `OIOout&`

The type of the argument to the `storer(ostream&)` function has been changed to an `OIOout&`. `OIOout` is an abstract base class, and all output operators are virtual functions, so you can customize the Object I/O format by defining your own derived classes. `OIOostream` and `OIOnihout` implement a format similar to the old OOPS format.

Automatic separators output by `OIOostream::operator<<()`

It is no longer necessary to explicitly output a space after each number written in a `storer(OIOnihout&)` function. Class `OIOostream` reimplements `operator<<()` to supply the space automatically.

`FileDescTy&` replaced by `OIOifd&/OIOofd&`

The type of the argument to the `storer(FileDescTy&)` function has been changed to an `OIOofd&`. and the argument to the `readFrom(FileDescTy&)` function has been changed to an `OIOifd&`. `OIOifd` and `OIOofd` are not abstract classes, and their I/O operators are not virtual functions as in `OIOin` and `OIOout`, so using them does not incur the overhead of a virtual function call for each member variable.

Change to `readFrom(OIOin&)`

Encountering EOF during `readFrom(OIOin&)` is now always an error. Previous releases returned `nil` if the input stream was initially at the EOF. `readFrom(OIOin&)` now behaves like `readFrom(OIOifd&)` always has.

`storeBin()` replaced by `OIOofd::operator<<()` and `OIOofd::put()`

The function `storeBin(FileDescTy&,` *`type`*`)` has been replaced by `OIOofd::operator<<(`*`type`*`)` and the function `storeBin(FileDescTy&,` *`type`*`*, unsigned)` has been replaced by `OIOofd::put(`*`type`*`*, unsigned)`.

`readBin()` replaced by `OIOifd::operator>>()` and `OIOifd::get()`

The function `readBin(FileDescTy&,` *`type`*`)` has been replaced by

`OIOifd::operator>>(`*`type`*`)` and the function `readBin(FileDescTy&,` *`type`*`*, unsigned)` has been replaced by `OIOifd::get(`*`type`*`*, unsigned)`.

`read_Cstring()` replaced by `OIOin::getCstring()`

The function `read_Cstring()` has been replaced by `OIOin::getCstring()`.

`store_Cstring()` replaced by `OIOout::putCstring()`

The function `store_Cstring()` has been replaced by `OIOout::putCstring()`.

`READ_OBJECT_AS_BINARY` eliminated

The `READ_OBJECT_AS_BINARY` macro has been eliminated. Replace it with code to read member variables individually using `>>` and `get()`.

`STORE_OBJECT_AS_BINARY` eliminated

The `STORE_OBJECT_AS_BINARY` macro has been eliminated. Replace it with code to store member variables individually using `<<` and `put()`.

Changes to the `DEFINE_CLASS` macro

The new version of the `DEFINE_CLASS` macro has hooks for supporting multiple inheritance. Before calling the `DEFINE_CLASS` macro, you must define three preprocessor symbols: `BASE_CLASSES`, `MEMBER_CLASSES`, and `VIRTUAL_BASE_CLASSES`. As an example, suppose you are writing the implementation of a class with the following declaration:

```
class X: public A, public virtual B {
    C c;        // C is a class
    D d;        // D is a class
//...
};
```

Set the symbol `BASE_CLASSES` to a list of the addresses of the class descriptors for the base classes of the class you are defining. These must be in the same order as they appear in the class declaration:

```
#define BASE_CLASSES A::desc(), B::desc()
```

Set the symbol `MEMBER_CLASSES` to a list of the addresses of the class descriptors for any member variables of the class that are NIH Library classes. These must be in the same order as they appear in the class declaration:

```
#define MEMBER_CLASSES C::desc(), D::desc()
```

If a class has no class members, define MEMBER_CLASSES, but give it no value.

Set the symbol VIRTUAL_BASE_CLASSES to a list of the addresses of the class descriptors for the virtual base classes of the class you are defining. These must be in the same order as they appear in the class declaration:

```
#define VIRTUAL_BASE_CLASSES B::desc()
```

If a class has no virtual base classes, define VIRTUAL_BASE_CLASSES, but give it no value.

Now you are ready to call the DEFINE_CLASS macro:

DEFINE_CLASS(*classname*,*version*,*identification*,*initor1*,*initor2*)

Classname is the name of the class you are defining.

Version is the version number of the class you are defining. It should be changed whenever the format of the information written by the storer() function changes such that older versions of readFrom() can no longer interpret it correctly.

Identification is a character string that identifies the revision level of the implementation of the class. It is simply stored in the class descriptor where you can retrieve it by calling the function Class::ident(). The identification parameter is intended for use with a revision control system such as RCS or SCCS. NIH Library classes specify it as the string "$Header$", which RCS replaces with the revision identification.

Initor1 and *initor2* are pointers to functions you may supply to perform initialization for the class, for example, initializing static data that the class uses.

DEFINE_ABSTRACT_CLASS

Abstract classes should use the new macro DEFINE_ABSTRACT_CLASS instead of DEFINE_CLASS. DEFINE_ABSTRACT_CLASS has the same arguments as DEFINE_CLASS; the only difference is that the reader() functions it generates do not reference the class's object I/O constructors, and shallowCopy() is defined as a derivedClassResponsibility().

Change to Class::className()

The function className() returns the name of the class of the object to which it is applied. In previous releases Class::className() does not do this. Instead, it returns the name of the class described by the class object to which it is applied. This release eliminates this inconsistency: className() returns "Class" when applied to an instance of class Class. The new function Class::name() returns the name of the class described by an instance of class Class.

New member functions of class `Class`

`Class** baseClasses()` returns a zero-terminated array of pointers to the class descriptors of the base classes of this class.

`Class** memberClasses()` returns a zero-terminated array of pointers to the class descriptors of the member classes of this class.

`Class** virtualBaseClasses()` returns a zero-terminated array of pointers to the class descriptors of the virtual base classes of this class.

`unsigned long signature()` returns the signature of this class. The signature of a class is computed by hashing the signatures of this class's base and member classes and the version number of this class. It is currently used by `storeOn()`/`readFrom()` to prevent obsolete versions of objects from being read.

`const Class* Class::lookup(const char* name)` returns a pointer to the class descriptor object for the class with the specified name. `Class::lookup()` returns 0 if the name is not found.

Changes to `copy()`, `shallowCopy()`, and `deepCopy()`

In previous releases, `shallowCopy()` made a bitwise copy of an object, and `deepCopy()` first made a shallow copy of an object, and then called the virtual function `deepenShallowCopy()` to convert the shallow copy to a deep copy. Each class reimplemented `deepenShallowCopy()` to handle any pointer member variables contained in instances of its class.

The problems with this approach are that (1) it is usually unsafe to make a shallow copy of an object that contains pointers, and (2) with Release 2.0 of the AT&T C++ Translator, objects may contain compiler-generated pointers which `deepenShallowCopy()` cannot handle easily and portably.

The following changes have been made in an attempt to solve these problems:

Each class now reimplements the virtual function `shallowCopy()` to call the initialization constructor `X::X(const X&)` to make a shallow copy of an object. The implementation of `shallowCopy()` is the same for all classes and is generated automatically by the `DEFINE_CLASS` macro:

```
Object* classname::shallowCopy()
{
    return new classname(*this);
}
```

`Object::deepCopy()` still calls the virtual function `deepenShallowCopy()` to convert a shallow copy, now made by the initialization constructor, to a deep copy. Since the shallow copy is no longer a simple bitwise copy, you may need to change `deepenShallowCopy()` for some classes.

In previous releases, `copy()` defaulted to `deepCopy()` since it was unsafe for general use. Beginning with this release, `copy()` defaults to `shallowCopy()` as it does in Smalltalk-80. Also, `Object::deepCopy()` is no longer a virtual function.

New function `dumpOn()` and changes to `printOn()`

To make the `printOn()` function more useful in application programs, it has been changed to print minimal formatting information, the idea being that this can frequently be added by an application to suit its specific needs. The virtual function `dumpOn()` has been added to assist in debugging by printing more detailed information than `printOn()`. `Object::dumpOn()` prints the name of an object's class, a left square bracket (`"["`), calls `printOn()`, then prints a matching right square bracket and a newline (`"]\n"`). `Collection::dumpOn()` does the same, except that it applies `dumpOn()` to all objects in the collection instead of calling `printOn()`. Other classes reimplement `dumpOn()` to print more appropriate information.

By default, `dumpOn()` sends its output to `cerr`. A default argument has also been added to `printOn()` so that it writes to `cout` by default.

Changes to class `Link`

Constructor `Link(const Link&)` changed to `Link(Link*)`

The constructor `Link(const Link&)`, which constructs a `Link` that points to the argument `Link`, has been replaced by the constructor `Link(Link*)`. This change was necessary so that `shallowCopy()` could call the constructor `Link(const Link&)` to make a bitwise copy. We suggest temporarily commenting out the declaration of this constructor in the file `Link.h` and recompiling the programs that depend upon it so you can easily detect and change the code using `Link(const Link&)` to use the new `Link(Link*)` constructor instead.

As a result of this change, `Link::shallowCopy()` is now enabled and will return a bitwise copy of a `Link`.

New function `isListEnd()`

Class `Link` has a new member function `bool isListEnd()`, which you must use to check for the end of a `LinkedList` instead of checking for a pointer to `Nil`. Compile your program with `-DMI` to find the places where you need to make this change.

Changes to class `Iterator`

`Iterator::shallowCopy()` now produces a shallow copy with a pointer to the same collection bound to the original instead of to a shallow copy of the collection bound to the original, as in prior releases. `Iterator::deepCopy()` works as before: it produces a deep copy with a pointer to a deep copy of the collection bound to the original.

The member variable `Object* state` has been added, which collection classes can use to associate additional state information with an `Iterator`. For example, a collection class implemented as a tree structure can use `state` to point to a `Stack` used to maintain the state of a traversal of the tree. The destructor for class `Iterator` calls a new virtual function, `doFinish()`, which a class that uses `state` can reimplement to delete the state object when the `Iterator` is destroyed.

The `storeOn()` format has changed as a result of these modifications.

The function `Object* Iterator::operator()()` has been added to return a pointer to the current object, or 0 if there is none.

Changes to class `Dictionary`

The return type of `assocAt()`, `removeAssoc()`, and `removeKey()` has been changed from `LookupKey&` to `LookupKey*` for consistency with the return types of similar functions. `Dictionary::assocAt()` returns 0 instead of `nil` if the key is not found.

`const` arguments to member functions

In previous versions , it was possible to convert, or "widen", a pointer to a `const` object into a pointer to a non-`const` object by adding the `const` object to a collection class and then removing it:

```
Object* f(const Object& co)
{
    OrderedCltn c;
    c.add(co);
    return c.remove(co);
}
```

C++ R2.0 now issues error or warning messages when a `const` pointer is converted into a non-`const` pointer. To eliminate these problems, the const arguments to some functions such as `add()` have been changed to non-`const` arguments. These changes affect classes `Assoc`, `LinkOb`, and the collection classes.

If you need to add a `const` object to a collection class, you must use an explicit cast:

```
c.add((Object&)co);
```

Change to `Bag::remove(const Object&)`

`Bag::remove(const Object&)` now returns 0 until last occurrence removed instead of the address of the argument to eliminate "widening" of the `const` argument.

`shouldNotImplement()` functions now `private`

Virtual member functions that a class reimplements to call `shouldNotImplement()` have been made private so that the compiler can give an error message if a client program attempts to apply the function to an instance of the class.

Change to class Stack

When a `Stack` is converted to another type of collection, the objects in the `Stack` are added to the collection from the top of the stack down. Previous releases added them bottom-up.

Changes to class Heap

The new member function `Heap::removeId(const Object&)` allows you to remove the object that is the *same* (i.e. `isSame()`) as the argument object from a `Heap`.

Iterating over a `Heap` now visits the objects in the heap in sorted order, from smallest to largest. Previous implementations visited the objects in heap order. This affects the order in which `printOn()` lists the objects in a `Heap`, for example.

Changes to class LinkedList

The new member function `LinkedList::removeId(const Object&)` allows you to remove the object that is the *same* (i.e. `isSame()`) as the argument object from a `LinkedList`.

Changes to class LookupKey

The virtual function `Object* LookupKey::value() const` has been replaced by two virtual functions

```
virtual Object* value();
virtual const Object* value() const;
```

to prevent obtaining a non-const pointer from a const `LookupKey`.

Changes to class Arraychar

The constructor for class `Arraychar` now initializes each element of the the array to 0.

The virtual function `removeAll()` has been implemented, which resets each element of the array to 0.

Changes to class Assoc

The virtual function `Object* Assoc::value() const` has been replaced by two virtual functions

```
virtual Object* value();
virtual const Object* value() const;
```

to prevent obtaining a non-const pointer from a const `Assoc`.

Changes to class AssocInt

The virtual function `Object* AssocInt::value() const` has been replaced by two virtual functions

```
virtual Object* value();
virtual const Object* value() const;
```

to prevent obtaining a non-const pointer from a const `AssocInt`.

CHANGES BETWEEN OOPS V2R1 AND OOPS V2R2

Class name changes

The name of class `Arrayobid` is now `ArrayOb`, the name of class `Linkobid` is now `LinkOb`, and the typedef `obid` has been removed. Just change all occurrences of `Arrayobid` to `ArrayOb`, `Linkobid` to `LinkOb`, and `obid` to `Object*`.

Type `bool` now `int`

Type `bool` has been changed from `char` to `int` for compatibility with X V11.

New `String` class

There is a new, more efficient implementation of class `String`. The new `String` class is compatible with the old `String` class except for the following:

```
String(char c, unsigned l =1);
```

is now:

```
String(char& c, unsigned l=1,
    unsigned extra=DEFAULT_STRING_EXTRA);
```

The argument `unsigned extra` has been added to most of the `String::String()` constructors to allow the programmer to give a hint as to how much space to allocate in the string for additional characters. When properly used, this can reduce the number of calls made to the memory allocator.

Assignment to substrings has changed slightly. The old `String` class handled an assignment to a substring such as:

```
String s = "abcdef";
s(0,2) = "123";          // result is 12cdef
```

by truncating the source string to the length of the destination substring. An assignment such as:

```
s(0,2) = "1";            // result is 1\0cdef
```

would cause a null byte to be inserted in the destination substring.

The new `String` class replaces the target substring with the source string, adjusting the length of the target string if necessary. Thus

```
String s = "abcdef";
s(0,2) = "123";          // result is 123cdef
```

and:

```
s(0,2) = "1";            // result is 1cdef
```

Changes to Class `Process`

An interface to `select`(2) has been added:

```
void Process::select(FDSet& rdmask, FDSet& wrmask, FDSet& exmask);
```

PORTING THE PROCESS CLASSES

This section describes the steps to follow if you want to be a pioneer and port the `Process` classes to a new machine/operating system.

If your target system provides the C library routines `setjmp()`, `longjmp()`, and `alloca()`, and if the implementation of `setjmp()`/`longjmp()` operates by saving/restoring all of the machine's volatile registers (as they do on the Sun-3, Sun-4, and IBM RT/AOS), then the port should be very easy; otherwise, you'll need to write versions of these routines in assembly language that behave as expected.

To find out how your `setjmp()`/`longjmp()` works, either look at the source code for these routines (if you're fortunate enough to have it) or use the debugger to disassemble them. A data structure of type `jmp_buf`, defined in `setjmp.h`, is passed as an argument to these routines. If your `setjmp()` just saves all the volatile registers in it to be restored later by `longjmp()`, then you're probably in luck---all you need to figure out are the offsets in the `jmp_buf` structure where the PC (Program Counter), SP (Stack Pointer), and FP (Frame Pointer) registers are saved.

Next, look at `nihclconfig.h` and locate the place where it defines the machine-specific inline functions `SETJMP()`, `LONGJMP()`, `_SETJMP()`, `_LONGJMP()`, `ENV_PC()`, `ENV_SP()`, and `ENV_FP()`. These define the interface to the `Process` classes. For example, here are the definitions for SunOS 4.0 on the Sun-3:

```
#ifdef SUNOS4

#ifdef mc68000
typedef jmp_buf JMP_BUF;
inline int SETJMP(JMP_BUF env)              { return setjmp(env); }
inline void LONGJMP(JMP_BUF env, int val)   { longjmp(env,val); }
inline int _SETJMP(JMP_BUF env)             { return _setjmp(env); }
inline void _LONGJMP(JMP_BUF env, int val)  { _longjmp(env,val); }
inline unsigned& ENV_PC(JMP_BUF env)
                    { return (unsigned&)env[3]; }
inline unsigned& ENV_SP(JMP_BUF env)
                    { return (unsigned&)env[2]; }
inline unsigned& ENV_FP(JMP_BUF env)
                    { return (unsigned&)env[15]; }
#endif

// ...
#endif
```

Add an `#if` ... `#endif` section for your machine and define the `ENV_PC()`, `ENV_SP()`, and `ENV_FP()` functions to return a reference to the appropriate word in the `JMP_BUF` array.

If your machine doesn't use both an SP and FP, then you'll also need to add some machine dependent C++ code to `HeapProc.c` to relocate only the one actually used. See the code for the `ibm032` in `HeapProc.c` as an example.

If your system has both `setjmp()`/`longjmp()` and `_setjmp()`/`_longjmp()`, define `SETJMP()`, `LONGJMP()`, `_SETJMP()`, and `_LONGJMP()` to call the corresponding routine. If your system doesn't have the "_" versions, check your documentation to see if your `setjmp()`/`longjmp()` saves and restores the signal mask; if so, define `_SETJMP()` and `_LONGJMP()` to call `setjmp()` and `longjmp()`, respectively. See the code for `SUNOS3` as an example.

If your `setjmp()` and `longjmp()` do *not* save and restore the signal mask, you'll need to provide versions that do. Define `JMP_BUF` to be a struct that consists of a `jmp_buf` plus whatever other members you need to save the signal mask. Then define `_SETJMP()` and `_LONGJMP()` to call `setjmp()` and `longjmp()` using the `jmp_buf` part of a

JMP_BUF, and define SETJMP() and LONGJMP() to do the same, but in addition to save/restore the signal mask using the other members of a JMP_BUF. See the code for the mc300 as an example.

If your setjmp()/longjmp() do not work by saving/restoring all volatile registers (as on the VAX), you'll need to write versions with different names that do, and call these instead from the interface functions.

If you succeed in porting the Process classes to a new machine/operating system, we'd appreciate a copy of the code for inclusion in future releases.

AT&T C++ TRANSLATOR RELEASE 2.00/2.1 BUGS

Releases 2.00 and R2.1 of the AT&T C++ Translator have a few bugs that we had to insert work-arounds for in the NIH Class Library. These are conditionally compiled based on the definition of preprocessor symbols beginning with BUG_. If you are using the NIH Class Library to test a Release 2.00 or 2.1 -compatible C++ compiler, we suggest that you edit the master Makefile to define these symbols:

```
# Disable AT&T R2.0/R2.1 bug work-around code
#BUGDEFS =
BUGDEFS = -DBUG_bC2728 -DBUG_38 -DBUG_39 -DBUG_OPTYPECONST
```

YACC STACK OVERFLOWS

The preprocessor symbol BUG_TOOBIG controls compilation of code we had to insert to avoid "yacc stack overflow" errors in the SunOS 3.5 C compiler. Release 2.0 produces very complicated expressions for the inline copy constructors it generates for deeply-derived classes. Explicitly defining non-inline copy constructors solves the problem. If you are a C compiler vendor, please make your tables big enough to handle the C code generated by the AT&T C++ Translator!

COMPILING UNDER AT&T C++ TRANSLATOR RELEASE 2.1

R3.0 of the NIH Class Library has been tested with Release 2.1 of the AT&T C++ Translator under SunOS 4.0. The following subsections summarize the changes required when compiling with R2.1.

Inconsistent declarations of alloca() in header files

The header files supplied with R2.1 declare alloca() with a return type of void* in alloca.h and char* in malloc.h, so you get an error message from the compiler when both of these files are included in the same compilation unit. We solved this problem by changing the return type of the declaration of alloca() in malloc.h to void*.

Warning and error messages due to #pragmas in SunOS 4.0 header files

The C compiler occasionally issues warning messages such as the following when compiling the output of the AT&T C++ Translator (both R2.0 and R2.1) under SunOS 4.0:

"/usr/include/CC/sys/signal.h", line 38: warning: function name expected

This is because C++ doesn't understand the `#pragma` directives it encounters in some system header files, so it just passes them through to the C compiler. The warning message results because C++ eliminates or moves the C function declarations that the `pragma` references. Another problem is that the C++ header files use `#define` to temporarily rename system functions when they include the vendor's C header files as a way to hide the effects of the vendor's C declarations for these functions. Unfortunately, this garbles the function names in the `#pragma` directives also.

Under R2.0, the `#pragma` problem just results in warning messages, but under R2.1, the C compilation occasionally fails with an error message. We manually made a new, self-contained version of `/usr/include/CC/setjmp.h` with the following `pragma` at the end:

```
/*
 * Routines that call setjmp have strange control flow graphs,
 * since a call to a routine that calls resume/longjmp will
eventually
 * return at the setjmp site, not the original call site.  This
 * utterly wrecks control flow analysis.
 */
#pragma unknown_control_flow(sigsetjmp, setjmp, _setjmp)
```

This eliminates the error messages, but not the warning messages. It would probably be no worse to simply remove the `#pragma`s altogether, since they don't seem to be having the intended effect, and that would eliminate the warning messages also.

C++ Translator +p option broken

The +p option produces spurious errors under R2.1. Edit the master `Makefile` to not use the +p option when compiling with R2.1:

```
# C++ flags
# NOTE: Disable +p option when compiling with AT&T R2.1
#CCFLAGS = +p
CCFLAGS =
```

Optional support for nested types

You can optionally define the preprocessor symbol `NESTED_TYPES` to cause the NIH

Class Library to use nested types under R2.1. Edit the master `Makefile` as follows:

```
# Compile with nested types (works with AT&T R2.1 and GNU C++)
#NESTED_TYPES =
NESTED_TYPES = -DNESTED_TYPES
```