Programmer's Guide

SPARCompiler™ C++

Version 3.0

SunPro
A Sun Microsystems, Inc. Business

© 1992 by Sun Microsystems, Inc.—Printed in the United States of America.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

All rights reserved. This product and related documentation is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or related documentation may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® and Berkeley 4.3 BSD systems, licensed from UNIX Systems Laboratories, Inc. and the University of California, respectively. Third party font software in this product is protected by copyright and licensed from Sun's Font Suppliers.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

The product described in this manual may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS

Sun Microsystems, Sun Workstation, Solaris, and NeWS are registered trademarks of Sun Microsystems, Inc.Sun, Sun-4, SunOS, SunPro, the SunPro logo, SunView, XView, X11/NeWS, and OpenWindows are trademarks of Sun Microsystems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc.

All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCworks and SPARCompiler are licensed exclusively to Sun Microsystems, Inc. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK® and Sun™ Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUI's and otherwise comply with Sun's written license agreements.

X Window System is a trademark and product of the Massachusetts Institute of Technology.

Chapter 1 "**Introduction**"—Introduction to the C++ product.

Chapter 2 "**About This Version of C++**"—Summarizes the C++ language.

Chapter 3 "**Using C and C++**"—How to move from programming in C to programming in C++, and how to write C++ libraries for C programs.

Chapter 4 "**Using the C++ Compiler**"—How to use the C++ compiler, with special attention to command-line options.

Chapter 5 "**The Iostream Package**"—Use of and basic documentation for the iostream library.

Chapter 6 "**Co-routine Library**"—Use of and basic documentation for the C++ co-routine library.

Chapter 7 "**Complex Arithmetic Library**"—Use of and basic documentation for the C++ complex mathematics library.

Chapter 8 "**C++ Programming Conventions**"—Programming tips for C++ programmers.

Chapter 9 "**FORTRAN Interface**"—How to call C++ from FORTRAN and how to call FORTRAN from C++.

Chapter 10 "**Tools**"—Description of C++ tools.

Appendix A "**Sample Program**"—Code for a sample class, a simple string type, and a sample program that uses the class.

Appendix B "**Co-routine Examples**"—Code for a sample program that uses the co-routine library.

Appendix C "**Differences Between C++ 3.0.1 and Previous Releases**"—Differences between previous versions of the AT&T or USL C++ Translator, in release 1.2, 2.0, 2.1, and the version of the translator used for Sun C++, up to Release 3.0.1. It also discusses modifying existing C++ 1.2, 2.0, and 3.0 code to work with Sun C++, and `#include` file limitations in Sun C++.

Appendix D "**Functions with Variable Numbers of Arguments**"—Technical details of defining functions with variable numbers of arguments.

Appendix E "**Creating Generic Header Files**"—Method you can use to create header files that can be used for Sun C, ANSI C, and C++.

- ***C++ 3.0.1 Language System Product Reference Manual***

  The *Product Reference Manual* provides a complete definition of the C++ language supported by Release 3.0.1.

- ***C++ 3.0.1 Language System Library Manual***

  The *Library Manual* contains information derived from papers that document the libraries included with C++.

- ***C++ 3.0.1 Language System Selected Readings***

  The *Selected Readings* manual contains papers that were presented at forums such as a C++ or object-oriented programming conferences. Those papers are included in the *Selected Readings* manual because they provide different perspectives on the C++ language.

- ***C++ 3.0.1 Language System Release Notes***

  The *Release Notes* manual describes enhancements to Release 3.0.1, and differences between this release and previous releases.

On-line Documentation

C++ on-line documentation UNIX manual pages and the AnswerBook™ documentation system. You do a separate install for the AnswerBook system. You display the manual pages with the `man` command.

Hard Copy and AnswerBook Documents

The following documents are on-line (in the Answerbook system) and in hard copy, as shown.

Title	**Part Number**	**Hard Copy**	**On-line**
C++ 3.0.1 Language System Product Reference Manual	800-7025-11	x	x
C++ 3.0.1 Language System Library Manual	800-6987-11		x
C++ 3.0.1 Language System Selected Readings	800-7024-11		x
C++ 3.0.1 Language System Release Notes	800-6988-11	x	x
Numerical Computation Guide	800-7097-11		x
Installing SPARCworks and SPARCompiler Software on Solaris 2.0	800-7333-11	x	

- The `courier font` shows system prompts, system replies, and C++ statements and key words.
- The **`boldface courier font`** shows text that you enter during interactive sessions.

```
tutorial% echo hello
hello
tutorial%
```

- A common operating system prompt is the percent sign (`%`), but most programmers customize their workstations to have distinct host names in front of a prompt. For this reason, and so that you can easily recognize examples in this manual, `tutorial%` denotes a system prompt.
- *Italics* indicate one of four things in this guide:
  - General arguments or parameters that you should replace with the appropriate input, for example:

    dc::dc (*dc ctor parameters*): [*class_name*] (`bc ct`*or parameters*)
  - Emphasis:

    Do not change anything here.
  - New terms:

    A *friend* is a class or a function that is not a member of a class, but is given permission to access private and protected members of that class.
  - Book titles:

    Product Reference Manual
- The names of operating system programs listed in text, such as `cfront`, `dbx`, are printed in `courier font`.

- `gprof++`
- `nm++`
- `ctags++`
- `rpcgen`

For C++ 3.0.1, these tools are either part of the package or bundled with operating system, release 5.0 .

C++ 3.0.1 is now based on ANSI C (not K&R C). See Appendix C, "This Release of C++," for a discussion of the major differences between C++ release 2.1 and Release 3.0.1 arising from this change.

1.2 *Compiler and Driver*

Note – In this guide, if the *assembler* is discussed, it refers to `fbe` for operating system 5.0 and `as` for operating system 4.1.x .

C++ is a complete compilation system. When you type `CC` the following occurs:

1. `acpp` **performs preprocessing.**
2. `cfront` **converts C++ code to C code.**
3. `ptcomp` **processes templates (only if the user program contains templates).**
4. `acomp` **compiles C code into assembly code.**
5. `iropt` **and** `cg` **optimize for execution time and generate assembly code (optional).**
6. `fbe` **(far back end) or** `as` **(in the 4.1.x operating system) converts assembly code into object files.**
7. `ptlink` **processes templates (only if the user program contains templates).**
8. `ld` **performs link editing.**

Chapter 4, "Using the C++ Compiler," discusses basic use of the driver, including how to compile a program.

This last feature, particularly, allows good design of modular, extensible interfaces among program modules.

This chapter provides a very brief overview of C++ from a conceptual point of view, with particular emphasis on the areas of difference and similarity with C. Chapter 2, "About This Version of C++," gives a full overview of the language. Chapter 3, "Using C and C++," summarizes issues important to C programmers moving to C++.

Compatibility with C

C++ is almost entirely compatible with C. The language was purposely designed this way; for one thing, experienced C programmers can learn C++ at their own pace and incorporate features of the new language when it seems appropriate. What is new about C++ supplements what is good and useful about C; most importantly, C++ retains C's efficient interface to the hardware of the computer, including types and operators that correspond directly to components of computing equipment.

C++ does have some important differences with C that you should be aware of. An ordinary C program probably won't be accepted by the C++ compiler without some modifications. Chapter 3, "Using C and C++," discusses what you must know to move from programming in C to programming in C++.

Even though the differences between C and C++ are most evident in the way you can design interfaces between program modules, C++ retains all of C's facilities for designing such interfaces. You can, for example, link C++ modules to C modules. This allows you to use C libraries with C++ programs.

Type Checking

A compiler or interpreter performs *type checking* when it ensures that operations are applied to data of the correct type. C++ has stronger type checking than C, though not as strong as that provided by Pascal. The approach to type checking is different from the approach in languages like Pascal: where Pascal always protests attempts to use data of the wrong type, the C++ translator protests in some cases and in other cases converts data to the correct type.

Rather than allowing the translator to do these automatic conversions, you can explicitly convert between types, as you can in C.

Object-Oriented Features

A program is *object-oriented* when the program is designed with classes organized so that common features are embodied in *base* classes. (Base classes are also sometimes called *parent* classes.) The feature that makes this possible is *inheritance*. A class in C++ can inherit features from one base class or from several. A class that has a base class is said to be *derived* from the base class.

The greatest use of this idea is in extending existing programs or libraries; you can define a new descendant that differs from its parent in some way that was not imagined when the parent class was designed. For example, a class defines a kind of window with scroll bars; you later want to implement windows with a different kind of scroll bars. You could create a descendant of the original window class and simply change the implementation of the scroll bar functions, without reimplementing or even examining the implementation of other parts of the program.

Other Differences from C

C++ differs from C in a number of other details. They are simply listed here:

- Defined constants in C++ allow you to avoid using the preprocessor to use named constants in your program.
- Default types for function parameters are not used in C++. You generally must specify function parameter types.
- Free store operators `new` and `delete` create dynamic variables in C++.
- You can use references as function parameters. References are alternate "handles" on the same object. A reference is an automatically dereferenced pointer, and acts something like an alternate name for a variable.
- There is a functional syntax for type coercions.
- C++ allows programmer-defined automatic type conversion.
- Variable declarations are allowed anywhere, not just at the beginning of the block.
- A new comment delimiter begins a comment that continues to the end of the line.

If you reset your system locale to, say, France and rerun the program, you'll still get the same output. The period won't be replaced with a comma, the French decimal unit.

Locale

You can change your application from one native language to another by setting the locale. For information on this and other native language support features, see the operating system documentation.

Use with OpenWindows

OpenWindows™ 3.0:1 and later releases provide C++ compatible header files for the XView (not NeWS) libraries (SunOS *5.0 only*).

OpenWindows™ 2.0 and later releases provide C++ compatible header files for the XView (not NeWS) libraries (SunOS *4.1.x only*).

- Overloading resolution

  The C++ overloading mechanism was revised to allow resolution of types that used to be too similar and to gain independence of declaration order.

- Type-safe linkage

  The `overload` declaration and keyword were abolished. Type specification of function arguments now eliminates ambiguity.

- Multiple inheritance

  It is possible to derive a class from more than one base class.

- Base and member initialization

  It is possible to specify the order in which base and member classes are initialized.

- Abstract classes

  A class with one or more pure virtual functions is an abstract class. A "pure virtual" function is a virtual function that does not have a definition. An abstract class can only be used as a base for another class.

- `static` member functions

  A `static` member function is a member whose name is in the class scope and the usual access control rules apply. A `static` member function is not associated with any particular object and need not be called using the special member function syntax.

- `const` member functions

  The `const` member function is a member function that can be called for all objects including `const` objects. (A non-`const` member function can be called only for a non-`const` object.)

- Initialization of `static` members

  A `static` data member of a class must be defined somewhere. The `static` declaration in the class declaration is only a declaration and does not set aside storage or provide an initializer. This is a change from the original C++ definition of `static` members, which relied on implicit definition of `static` members and on implicit initialization of such members to 0.

2.2 Classes and Members

You use a class - a user-defined data type - in the same way that you use a predefined data type. That is, a class not only contains data, like a C `struct`, but also defines operations that apply to objects of that class, as the translator or compiler does for objects of the built-in types. Here is an example of a class definition:

```
class string {
private:
     char* data;// private data fields
     int size;

public:
     string() { size= 0; data= NULL; } // inline constructor
     string(char*);// constructor function

void insert(char*) // public function

operator char*() { return data; } // conversion operator

string operator+(string);// operator functions
string operator=(string&);

friend ostream& operator<<(ostream&, string); //friends
friend istream& operator>>(istream&, string&);
};
```

Note – Most examples in this chapter, including this one, are part of the string module and programming example fully reproduced in Appendix A, "Sample Program".

A class is divided into two parts: one part preceding the label `public` and one part after. The part preceding `public` is known as the *private* part. (In this case the optional keyword `private` explicitly states that part is private.) Functions, operators, and data fields (collectively called *members* of the class) defined in the private part can only be accessed by other members or by *friends* of the class. (The `operator>>` function is an example of a friend function.) Members declared in the public part of the class can be accessed by any function within the scope of an object of this class. This allows you to make the

Member Data Fields

Member data fields work like fields of C structures, except that some data fields (those that are private or protected) can be hidden from functions that aren't members or friends of the class.

Member Functions

Functions that are part of the definition of a class are known as *member functions* of that class. When you call a member function, you always call it for a specific object of a class. For example, given the definition for class `string` used in an earlier example (and reproduced in full in Appendix A, "Sample Program"), you must create a variable of type `string` to call the member function `insert`:

```
string aString;
 aString.insert("Hello.");
```

Every member function is automatically and implicitly called with a parameter that is a pointer to the object used to call the function. Within the member function you can access the implicit parameter using the keyword `this`. For example, when you call `insert` as just shown, `insert` can use `this` to refer to a `string`.

A class can have any number of member functions with the same name, as long as the compiler can distinguish them based on their parameter types. These are called *overloaded* member functions. For example, the sample class `string` shown earlier in this chapter has overloaded constructors:

```
string();
string(char*);
```

The parameters of overloaded functions must differ enough so that the translator can distinguish between them; differences that are erased using

Referring to Members

You refer to members in five ways, whether the member is a function or a data field. One way only applies to `static` members. One way applies to a member you refer to from another member of the same class. The other ways apply to all members.

Referring to a Member from Another Member

When you want to refer to a member from another member of the same class, you simply give the member's name, as if it were an ordinary C variable or function. The meaning is as if `this->` preceded every member name. For example, given the definition of class `string` used at the beginning of this chapter (and in Appendix A, "Sample Program"), you can refer to the field `data` from within function `string::operator+` like this:

strcpy(holder, data);

The variable `holder` is a local variable.

Using the Operators -> and . to Refer to Member

These two operators have similar meanings and are used for some of the same purposes they are used for in C. As in C, you use -> with pointers and . with direct variables.

Note – If a class member function must refer to members of another (non-`this`) variable of the same class, it will use this syntax. See the implementation of the example `string::operator+` in Appendix A, "Sample Program" for an illustration.

Again as in C, you use these operators to refer to member data fields. In C++, they also can refer to member functions. In either case, you usually only need to use them from functions that are not members of the same class as the member you want to refer to. To use `.` (period), give a class-type variable name followed by the `.` followed by the name of the data field; to use `->` (a minus sign and a greater-than sign), give a pointer to a class followed by `->` followed by the name of the data field. For example, assuming that class

You can declare and initialize a pointer to the member function `insert` like this:

```
int (string::*pinsert)(char*) = &string::insert;
```

You can use the pointer where the last line calls function `insert` through the pointer:

```
string a;
(a.*pinsert)("hello");
```

You can also call a member function given a pointer to an object.

```
string* p;
(p->*pinsert)("hello");
```

You can also declare and use pointers to member data fields. For example,

```
struct S {
     int a;
};

int S::* psm = &S::a;

void f(S* ps)
{
     ps->*psm = 2;
}

void g()
{
     S a;
     f(&a);
}
```

This is equivalent to simply assigning `2` to `a.a`.

Operator functions are implemented by defining new functions that are called when you use the operators.

Every operator has a name formed by the word `operator` followed by the symbol for the operator. For example, `operator+` is the name of the + operator. You use the operator name to declare and define the operator function. The declaration of class `string` given at the beginning of this chapter shows a declaration of the + operator in the case of objects of type `string`; here is the implementation of the operator function:

```
string string::operator+(string second)
{
     char* holder = new char[ size + second.size +1 ];

     strcpy( holder, data );
     strcat( holder, second.data );
     string temp( holder );

     delete holder;
     return temp;
}
```

An important point about operator functions is that they do not imply anything about other operators that seem to be related. For example, for `int`, the += operator is related to the + and = operators. In the previous example, if you want the += operator to work with type `string`, you have to define an operator function for it. That operator function defines any relationship that might exist between += and any other operator. (In other words, nothing stops you from defining += to mean subtraction for your new type. That would be extremely bad programming, though.)

You cannot redefine operators for built-in types, and you cannot define brand new operators; you can only overload existing operators. The exception is conversion operators (see Section 2.8, "User-Defined Type Conversion," on page 47).

The following operators cannot be overloaded: `.`, `.*`, `::`, `?:`. The preprocessing symbols # and ## also cannot be overloaded (see the *C++ 3.0.1 Language System Product Reference Manual*). The `.*` operator is a binary operator which binds its second operand to its first operand. The second operand must be of type "pointer to member of class T" and the first operand must be of class T or a class publicly derived from class T. The result is an object or a function

If you want, you can also explicitly call operator functions using the normal function syntax. For example:

```
string first("Hello");
string second("there.");
string third;
operator=(third, operator+(first, second));
```

Use of this syntax is discouraged.

Overloading the Operator ->

The C++ Programming Language states that you cannot overload operator -> This is no longer true.

By overloading `operator ->`, you can create classes of objects that can act as "smart pointers." This may be important to programs where indirection is a key concept that can clearly be represented by `operator ->`. You can also use `operator ->` to provide C++ with a limited, but still very useful, form of delegation.

When overloading, `operator ->` is considered a unary operator of its left-hand operand and -> is reapplied to the result of executing `operator ->`. Therefore, the return type of an `operator ->` function must be a pointer to a class or an object of a class for which `operator ->` is defined. For example:

```
struct X {
Y* p;
     . . .
     Y* operator->() {
          if (p == 0) {
          // initialize p
          }
          else {
          // check p
          }
          return p;
     }
     . . .
};
```

```
void* operator new(size_t sz);
```

The type `size_t` is defined in `<stddef.h>`. It is defined as an `unsigned int` in Release 3.0.1 (see the C++ *3.0.1 Language System Product Reference Manual* for further details).

Once you make this definition, `X::operator new()` is used instead of the default `operator new()` for objects of `class X`. This does not affect other uses of operator `new` within the scope of `X`; it only affects the use of `new` on objects of class `X`.

The usual rules of inheritance apply. If you derive a class `Y` from `X`, `Y` objects are also allocated using `X::operator new()`. It is because of inheritance that `X::operator new()` needs an argument specifying how much space should be allocated; the size of a `Y` object may be different from the size of an `X` object. A class that is never used as a base class does not need the size argument. You always should use the size argument, though, unless you are absolutely sure the class will never be used as a base class.

Like the global `operator new()`, `X::operator new()` returns a `void*`. This indicates that it returns uninitialized memory. The translator must make sure that the memory returned by `operator new()` is converted to the proper type and, if necessary, initialized using the constructor of the class. The same thing happens for `X::operator new()`. The pointer in `X::operator new()` is uninitialized.

A constructor often has parameters that receive information needed to initialize data fields. For example, the following is the definition of a constructor of type `string`.

```
string::string(char* aStr);
{
     if( aStr == NULL ) size= 0;
     else       size = strlen( aStr );

     if( size == 0 ) {
          data = NULL;
     } else {
          data = new char[ size+1 ];
          strcpy( data, aStr );
     }
}
```

You implicitely call a constructor when you create a new object. (That is, the translator automatically calls the function; you only explicitly call a constructor function to convert a value of one type into a value of another type.) If the constructor has formal parameters, you must give actual parameters when you declare the object.

You can overload constructors. As with other overloaded function names, the parameters must differ enough that the translator can tell which to call. Class `string` has two constructor functions. The definition of one was just given. The implementation of the second constructor is given with the declaration of the function within the class declaration (making it an inline function) because it is very simple. Here it is:

```
string() { size= 0; data= NULL; }
```

The translator calls the first when you create a `string` using a `char*` string. When you create a `string` without giving a parameter, the other constructor is invoked. For example, the first line following invokes the constructor just shown. The second line invokes the other constructor.

```
string firstString("some initial data");
string secondString;
```

The constructors come last for members of the main class. A member constructor is written with its name. The base class is always constructed first, followed by the members, and, lastly, the class itself. If there is more than one member constructor, the translator calls them in the order you give them. Similarly, if there is more than one base class at a given level (that is, you've used multiple inheritance), the translator calls the constructors for those base classes in the order given.

You supply parameters using the same format. For example:

`dc anObj` *(dc ctor params)* : *[class_name]* *(c1 params)* , `d1`(*d1 params*);

To clarify the situation when you use multiple inheritance, consider the following definitions:

```
class X {public X(int, int);...};
class Y public {Y();...};
class Z : public X, public Y {public Z();...};
```

In the definition of the constructor `Z()` you specify the order of initialization by giving a statement like this:

`Z::Z():Y(), X(5, 10)` *{body of constructor}*

If there are other base classes you don't mention, the translator calls their constructors after the specified ones, in the default order, i.e. the order the classes appear in the program.

Constructors of virtual base classes are a special case. If the virtual base has a constructor, the translator calls that constructor before any constructor of its derived classes. See the *C++ 3.0.1 Language System Product Reference Manual* and "Virtual Base Classes" on page 39 in this chapter for definitions of virtual base classes.

Note – Virtual base classes can have constructors too. See the *C++ 3.0.1 Language System Product Reference Manual* for further details.

Destructors

Use destructor functions to do any cleanup when the program is done with an object. You must name a destructor function by concatenating a tilde (~) with the name of the class, in that order.

The assignment rule implies that for a class `X`, the constructor `X(const X&)` and the assignment operator `const X& X::operator=(const X&)` are supplied by the translator where necessary. Unless you supply it, a constructor `X(const X&)` is created for a class `X` where `X` has one of the following.

- A member or base of class `Z` for which `Z::operator=` or `Z::Z(Z&)` is defined
- A virtual function or a virtual base class

Access controls are correctly applied to both implicit and explicit copy operations so you have a way of prohibiting assignment of objects of a given class. For example:

```
class X {
     void operator=(X&);
     X(X&);
     . . .
public
     X(int);
     . . .
};
```

Because `operator=` is defined as a private member, only other members (or friends) of class `X` can use the operator. For example:

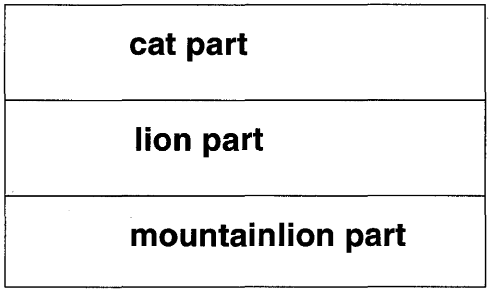
```
void f() {
     X a(1);
     X b= a;// error: X::(X&) private
     b = a;// error: X::operator=(X&) private
}
```

Taking another approach, a derived class may provide a specialized interface to a base class.

You can have multiple levels of inheritance. For example, given the preceding example,

```
class mountainLion: public lion
{ . . . };
```

An object of a class `mountainLion` is laid out like this



The different parts of the object are called *subobjects.*

The derived classes do not have access to the private members of their base classes. They do have access to public and protected members.

Public and Private Derived Classes

Notice the keyword `public` used in the sample derived class declarations in the previous section. Base classes can be declared `public` or `private`. When a base class is `public`, its members are inherited in their original form by the derived class. That is, public members of the base class become public members of the derived class. When a base class is declared `private`, all members inherited from the base class are private, even if they are declared `public` in the base class. Base classes are private by default, which is a possible source of confusion. The following two declarations are equivalent:

```
class window : private frame {...};
class window : frame {...};
```

You can also make a member of a public base class private. For example, given the definition of class `list` given in the last example, here is a similar definition of class `linkedlist`:

```
class linkedlist : public list {
     ...
private:
     list::print// print is now private but
                // add and remove are public
};
```

This is not exactly equivalent to the preceding example because it has the side effect of implying that you can no longer treat a `linkedlist` as if it were a `list` (that is, `linkedlist` is no longer a *subtype*[1] of `list`). There is no implicit coercion of a pointer to `linkedlist` to a pointer to `list`, and not all public members of `list` are automatically public members of `linkedlist`.

Virtual Functions

An important feature of object-oriented programming allows you to defer determination of what function is called to runtime. This ability is provided in C++ through the use of *virtual functions.*

For example, suppose you have a base class `shape` that has a `draw` member function.

```
class shape {

virtual void draw();
};
```

Suppose, further, that you have a number of classes derived from `shape`:

```
class rectangle {...};
class oval {...};
class polygon {...};
```

---

[1] If you take any `X` and treat it as a `Y`, then `X` is a *subtype* of `Y`.

Virtual Function Tables

Please reread the section on virtual function tables, or *virtual tables,* to understand the `+e` translator option (see the C++ *3.0.1 Language System Release Notes* manual).

When you create an object of some class, the translator allocates a contiguous region of memory for the object. In the simplest case, the object takes up just the space needed for the object data. For example, suppose you have a class with the definition:

```
class samp {
     int first;
     int second;
     void samp(int, int);
     void change(int, int);
};
```

The translator translates calls to the member functions `samp` and `change` into direct, ordinary function calls, so the program doesn't need information included with the objects about where to find member functions.

This is still true with derived objects. For example, a class derived from `samp` might have this definition:

```
class derived: samp {
     int third;
     void derived (int): int, int;
     void morechange(int, int, int);
}
```

Again, the translator changes calls of member functions `derived` and `morechanged` into direct, normal functions calls.

When you have virtual functions, such a simple scenario is not possible because the translator does not know what function will actually be called at execution time. When a class has virtual functions, the translator creates a virtual function table or `vtbl` for it. Then, when you create an object of that class, the translator inserts a pointer to the `vtbl` in the object. This pointer is sometimes called a `vptr`.

To illustrate this, consider the following example:

```
struct york {work();...};
struct america {work();...}
struct newyork : york, america {...}

main () {
     .
     .
     .
     newyork* dospassos;
     dospassos->work(); //error: ambiguous
     .
     .
     .
}
```

You could resolve this ambiguity by adding a function in the derived class with the same name. That function could call the function from the base class that you want to be called.

```
struct newyork : york, america {...
     work()
     {
          america::work();
     }
     ...
}
```

A class can appear as base class more than once in the ancestry of a derived class. For example, the following is legal:

```
class A : public X {...};
class B : public X {...};
class C : public A, public B {...};
```

Normally, this means that two (or more) copies or *instances* of the class appear more than once. If you want to have only one instance of that class, you should declare it as a *virtual base class*. See "Virtual Base Classes" on page 39 for details of virtual base classes.

The quality of being a virtual base class only applies to the use of a class as a base class. The class itself is not declared virtual.

Virtual classes exist primarily as a way of expressing dependencies among objects.

You can cast from a derived class to a virtual base class, but not from a virtual base class to a derived class. Casting from a derived class to a virtual base class involves following the virtual base pointer, which can be done. The opposite operation involves more information than is available at runtime.

2.4 Objects

An *object* is an *instance* of a class. In other words, an object is a part of memory allocated in a manner defined by the class definition: a specific instance of the general case defined by the class. Each object has its own data fields, except for static data fields, which are shared by all objects of a given class. The type of an object is not only its own class but also generally any base class of its class. For example, given the following class definitions, an object of type `thirdclass` is also of types `firstclass` and `secondclass`:

```
class firstclass {...};
class secondclass : firstclass {...};
class thirdclass : secondclass {...};
```

Once you've defined a class, you can create an object of the given type simply by declaring a variable of that type. For example,

```
firstclass anObject;
thirdclass anotherObject(5);
```

The first declaration creates an object of type `firstclass` that can be referred to using the name `anObject`. The second declaration creates an object of type `thirdclass` that can be referred to using the name `anotherObject`. The parameter 5 gets passed to the constructor of type `thirdclass`. Each new object remains in memory until its scope exits. (See "Allocation and Deallocation: Operators new and delete" on page 44 for information on how you can create objects that last even when the current scope exits.)

Note – The word *static* is used for two different purposes in C++. The declaration of static data members has nothing to do with the declaration of static objects.

The translator allocates space for all file-scope static objects and calls their constructors, if any, when the program begins. The translator calls the destructors for static objects and destroys the objects when the program ends.

Dynamic Objects

You create dynamic objects using the `new` free store operator. The declaration of a dynamic object looks like this:

```
someClass* aPointer = new someClass
```

(Notice that the `new` operator returns a pointer.)

The translator allocates space for a dynamic object and calls the constructor, if any, when it encounters the statement that declares the object. Unlike the situation with static and automatic objects, the translator does not destroy a dynamic object until you call the `delete` operator. When you call `delete`, it first calls the class destructor, if any, and then deallocates the space. (If you do not call `delete` before the program exits, the translator destroys the object at that time. The destructor for the object will not be called when the program exits, unless the program exits by calling the `return` function from within the `main` function.)

References to Objects

You can create additional names, or *references*, for objects that already exist.[1] If, for example, you've declared an `int` object (that is, an ordinary integer variable) like this

```
int anInt = 1;
```

you can declare a reference to the same `int` object like this:

```
int& theInt = anInt;
```

---

[1] A C programmer may think of a reference as a pointer that is automatically dereferenced except when passed as a reference parameter.

Allocation and Deallocation: Operators `new` *and* `delete`

C++ provides the `new` and `delete` operators to replace the standard UNIX system routines `malloc` and `free`. You use `new` to create a *dynamic object;* that is, an object that exists after the program exits the scope of the function that created it. You don't have to use `new` for objects that you don't want to use after the current function exits; the translator creates those *automatic* objects when they are declared, as with declarations of predefined types. The new dynamic object stays in existence until you use the `delete` operator to destroy it (or until the program completes).

There is no "garbage collection" built into C++; objects created with `new` that have no references to them are not destroyed until the program exits, even though there is no way to use them.

2.5 *Archiving Global Object Arrays in a C++ Library*

You may have problems if you archive a global object array into a C++ library you've built yourself. It is not a bug but intended cfront behavior.

When a global object array is defined and initialized, the actual initialization does not take place until the constructors for the array objects are called, which occurs at runtime.

If it is a single object instance rather than an array, cfront will initialize it to zero first. However, if it is an array of your own defined class objects, no initialization will occur. This is done intentionally to avoid increasing the size of the resulting object file.

For example,

```
class X {
public:
     int a;
     X(int b) : a(b) { }
};
X x[] = {1,2}, y(3);
```

2.6 In-line Functions

C programs sometimes use macros to replace small functions because frequently calling small functions can decrease the efficiency of a program. Macros, though, do not act exactly like functions. C++ provides in-line functions, thus eliminating the need to use macros for this purpose.

You can create an in-line function both explicitely and implicitely. To explicitly create an inline function, simply precede it with the keyword `inline`. For example:

```
inline int cube(int number) {
return number*number*number;
};
```

To create an implicitly in-line member function, simply declare and define the function (that is, give the body of the function) within a class definition. It will automatically be in-line. For example, this declaration appears in the definition of class `string`:

```
string() { size= 0; data= NULL; }
```

This constructor function is implicitly in-line. This form only works for class members.

In either case, when you use the function, the translator replaces it with equivalent code. For example, the expression

```
answer = cube(4)
```

is replaced with code equivalent to:

```
answer = 4*4*4
```

In-line functions are efficient only for very small functions, and they should be used only when necessary. The `inline` keyword is only a suggestion to the translator, and it may be ignored.

prints the string:

```
Here is some data.
```

Conversion Operators

Using constructors for conversion has limitations, in that you can't convert from a new type into a pre-existing type. The alternative is to define a *conversion operator.* A conversion operator is a function that is a member of the *source* type (unlike a constructor, which is a member of the *destination* type). You name a conversion operator by giving the keyword `operator` followed by the destination type name. For example, if you want an operator to convert a value of type `string` (see Appendix A, "Sample Program") to `char*`, you might include this definition in the definition of type `string`:

```
operator char*() {return data};
```

The operator `char*()` takes a value of type `string` as its input parameter and returns a value of type `char*` (simply by returning the `data` field, which is a `char*` field).

Once this operator is defined, if you use a value of type `string` where you need a value of type `char*`, the translator automatically uses the operator you've defined to convert the value. For example:

```
string aString("a");
char* x = aString;
```

You can also call it explicitly using a format like this:

```
char* x = (char*(aString));
```

Such conversion operators can render overloaded functions ambiguous and therefore illegal because the translator may no longer be able to tell two functions apart based on parameter types.

Type names containing `[]` and `()` as well as multiword types (such as `unsigned long`) cannot be defined this way. To define conversion operators for these types, give them names using `typedef`. For example,

```
typedef unsigned long u_long;
operator u_long(){ ... };
```

2.11 Overloaded Function Names

More than one function in a C++ program can share the same name. It makes sense to do this for functions that perform similar operations on values of different types. For example, you might define a function to store integer data in a file and want another function to store real number data. For similar reasons, you might want to create a function that takes the same kind of action as a standard function, but acts on values of a new type. You can give such functions different names, but it makes logical sense to give them the same name. In C++, you can declare them like this:

```
void store(int);
void store(float);
```

The translator decides which function to invoke based on the types of the parameters used when you call the function. The parameters of overloaded functions must differ enough so the translator can distinguish between them; differences that are erased using standard conversions or user-defined conversions are not enough. This means, for example, that the following is illegal:

```
int wontwork(int);
int wontwork(char);// error
```

This is illegal because one of the standard conversions would promote an argument of type `char` to match a formal argument of type `int`, leading to an ambiguous situation.

The overloading mechanism can even distinguish between signed and unsigned values. For example:

```
void f(int);
void f(unsigned);
void g1(int i, unsigned u)
{
     f(i);//invoke f(int)
     f(u);//invoke f(unsigned)
}
```

long	**new**	**operator**	**overload**	**private**
protected	**public**	register	return	short
signed	sizeof	sparc	static	struct
sun	switch	**template**	**this**	**throw**
try	typedef	union	unix	unsigned
virtual	void	volatile	while	

`_ _STDC_ _` is predefined, but has the value 0. For example, the following program:

```
#include <stdio.h>
main()
{
    #ifdef _ _STDC_ _
        printf("yes\n");
    #else
        printf("no\n");
    #endif

    #ifdef _ _STDC_ _ ==0
        printf("yes\n");
    #else
        printf("no\n");
    #endif
}
```

produces the following output:

```
yes
yes
```

Note – Treating `overload` as a keyword is an anachronism; future releases of C++ may not use this keyword. The names `catch`, `throw`, and `try` are not currently used for anything, but are reserved for use in future versions of the language.

Function Return Value Declarations

In C, when you don't declare the type of a function's return value, the compiler assumes the return value is an `int`. Although this is still true in C++, you should declare all function return values or declare the function `void`; otherwise, the translator is unable to check types. (This also makes your program more meaningful to those who use it.)

3.3 Structures

C++ reacts to structure definitions in slightly different ways from C, which may cause problems in C programs.

Structure Tags in Declarations

Structure tag names in C++ are also type names. You can use the tag name of a structure you've defined in a declaration without the keyword `struct`, although you also can give the keyword, if you want. For example:

```
struct anything {
/*contents of structure*/
};
void afunc(anything);
```

The last line can also be given as follows.

```
void afunc(struct anything);
```

Both lines have the same effect.

Structure Tags and Functions with the Same Names

C puts variables and structure tag names in different name spaces. C++, because of its abstract data typing and classes, uses one name space for variables and types. However, to maintain conformance with C and ANSI C, C++ permits:

```
struct growth { };
int growth(int *, struct growth*);
```

Operating System 5.0

For operating system, release 5.0, static constructors are executed from the `.init` section and `_main` should not be called. All static destructors are called from the `.fini` section.

3.5 Writing C++ Libraries for C Programs

Note – This section applies to operating system 4.1.x only.

This section discusses C++ implementation and component-dependency issues you may encounter if you are writing C++ libraries to link with C programs. These issues are particularly important if the C++ libraries are to be used by C programmers who do not have access to the Sun C++ translator.

The following examples describe two different scenarios:

- Writing C++ libraries *without* static initialization or destruction
- *Writing C++ libraries with* static initialization or destruction

The first example is not affected by implementation changes; it is the recommended way to write C++ libraries for C programs. The second example is affected by implementation changes. It is based on C++ 3.0.1 implementation only and may change in future releases.

This section will only discuss implementation-specific issues. For language specific issues, see Appendix F, "C Wrappers for C++ Functions". C++ runtime library licensing issues are not addressed here either.

Writing Libraries Without Static Initialization or Destruction

Writing a C++ library without static initialization or destruction applies when the following occurs:

- No static initialization or destruction exists, and therefore no class objects are declared in `FILE` scope — either internally linked (`static`) or externally linked (`global`).

  The object refers only to classes that have constructors or destructors defined in either the current class, or in a class from which it directly or indirectly derives. This also includes classes that contain virtual functions

To correctly call the static initialization and destruction mechanism in C++ 3.0.1, do the following:

If the Sun C++ translator is available to you — the `main()` module of the program must be compiled by the C++ translator driver, then all C and C++ object modules must be linked together by `CC`.

If the Sun C++ translator is not available — two minimal Sun C++ components are still required for Sun C++ 3.0.1:

- The `patch` C++ post-linker
- The `libC`

With the above components, modify the C program's `main() module`. For example:

```
main() {
  _main();
/* your code here */
}
/* this is to link in __head from libC.so for patch version
of cfront */
extern struct __linkl *__head;
struct __linkl **__LinkInHead = (struct __linkl **)(& __head );
```

All symbols with two preceding '_' (underscores) are, by convention, reserved for C++ implementation, and their use should be avoided. `__head` is not referenced anywhere else in the program; it is used by the `patch` postlinker to position the beginning of the chain of static initializer and destructor functions needed by `_main()`.

`Next`, link the program as follows:

```
tutorial% CC other modules and flags -lyour C++ library -lC
```

Make sure that `libC` is searched before `libc` (C library) because `libC` includes a different version of `exit()` that invokes the static destruction mechanism before exiting.

Last, run the postlinker `patch` on the final executable to chain the static initializer and destructor structures together.

```
tutorial% patch a.out
```

3.8 Linking to C Functions

The translator encodes C++ function names to allow overloading. To call a C function or a C++ function "masquerading" [1] as a C function, you must prevent this encoding. Do so by using the `extern "C"` declaration. For example:

```
extern "C" {
double sqrt(double); //sqrt(double) has C linkage
    }
```

This linkage specification does not affect the semantics of the program using `sqrt()` but simply tells the translator to use the C naming conventions for `sqrt()`.

Only one instance of an overloaded C++ function can have linkage. You can use C linkage for C++ functions that you intend to call from a C program, but you might not want to do that since you would only be able to use one instance of that function.

You cannot specify C linkage inside a function definition. It can only be specified globally.

---

[1] Although this section concentrates on using the `extern` "C" declaration to call C functions from C++ programs, you can also use it to create a C++ function that can be called from C programs. A C++ function that is called by a C program is masquerading as a C function. C++ functions masquerading as C functions cannot use many of the capabilities of C++; in particular, such functions cannot be overloaded and cannot be member functions. For example, A C++ function that is called from a C program cannot be an overloaded function or a member function.You may not want to do this because you would only be able to use one instance of the function.

CASE I - The object is exported and will be referenced by user applications.

If an object is exported and will be referenced by user applications, you can put its definition into a `.sa` file.The `.sa` file will be statically linked into the final executable only if user applications reference some data item defined in it.

For example, if a `.sa` file contains the following definition, object `obj_1` and `obj_2` will be properly initialized and destroyed if at least one of them is referenced by the user application:

```
Foo obj_1(3, "string");
Goo obj_2(5.432);
```

If you put all of the exported global objects into one `.sa` file, the whole file will be linked into the final executable — even if only a few of them are referenced. This not only makes the executable larger than it should be, but also degrades the performance of the application due to all the unnecessary constructor and destructor calls for the unused library objects. It will be even worse if those constructor or destructor calls result in side effects.

Note – Avoid using global objects in a library that may result in side effects during construction and destruction. Always put each exported library object into a single `.sa` file unless some of them are closely related and will always be used together; then it should work to group them into a single `.sa` file.

```
makefile:

        test: main.cc libfoo.so.0.1 libfoo.sa.0.1
                CC -o test main.cc -L. -lfoo
        lsrc.o: lsrc.cc
                CC -pic -c lsrc.cc

        sa.o: sa.cc
                CC -c sa.cc
        libfoo.so.0.1: lsrc.o
                ld -o $@ -assert pure-text $?

       libfoo.sa.0.1: sa.o
               ar rv $@ $?
               ranlib $@
```

After you make and run these files, they produce:

```
% ldd test
        -lfoo.0 => ./libfoo.so.0.1
        -lC.0 => /usr/lang/SC1.0/libC.so.0.2
        -lc.1 => /usr/lib/libc.so.1.5
% test
A::A(5)
main()
A::~A()
%
```

CASE 2 - The object is not exported; or, it is exported but its constructor and/or destructor should be invoked no matter whether user applications reference it or not.

If the object is only used internally in the library, you should not define it in a `.sa` file. The reason is that unless the user application happens to reference something else in the same `.sa` file, the `.sa` file won't be linked into the final executable. Thus, the constructor and destructor of the object won't be invoked. Worse yet, internal use of the global object in `.so` files without explicit referencing of the object in the user program will result in a "`Symbol not found`" dynamic linker error during runtime, if the object is defined in a `.sa` file.

```
A a_lib_obj;

lsrc2.cc:

        #include <stdio.h>
        #include <new.h>
        #include "libfoo.h"

        int B::a = 0;
        extern A a_lib_obj;

        B::B() {
        printf("B::B(%d)\n", a);
        ++a;
        if (a > 1) return;
        new (&a_lib_obj) A(5);
}

B::~B() {
        printf("B::~B(%d)\n", a);
        --a;
        if (a > 0) return;
        a_lib_obj.A::~A();
}

main.cc:

       #include <stdio.h>
       #include "libfoo.h"

main() {
       printf("main()\n");
         // ...
}

dummy.cc:

        #include "libfoo.h"

makefile:

       test: main.cc dummy.cc libfoo.so.0.1
             CC -o test main.cc dummy.cc -L. -lfoo
```

This will be equivalent to typing the following commands:

```
CC -c -pic lsrc1.cc lsrc2.cc
ld -dy -G -z text -o libfoo.so.1 lsrc1.o lsrc2.o
```

If you want to assign a name to your shared library for versioning purposes, type:

```
CC -G -o libfoo.so.1 lsrc1.cc lsrc2.cc -h libfoo.so.1
```

The resulting executable file is called, in this case, `myProg` because this command line uses the `-o name` argument. Without that argument, the executable file gets the default name `a.out`.

The file name extension can be `.c .c, .C, .cc, or .cxx`.

Compiling a Program That Uses a Standard Library

Under normal circumstances, you don't need to do anything special to compile a program that calls routines in a standard library. However, the standard library header file must be included at the beginning of your program using a format like:

```
#include <stdlib.h>
```

If the header files you want to use are in a different place, you can specify the location on the `CC` command line. For example, if the header files are in `/usr/libraries/include`:

```
tutorial% CC -I/usr/libraries/include myProg.cc
```

Compiling a Program with a Module

The sample program `testr` (see Appendix A, "Sample Program"), consists of two modules: the main program module `testr.cc` and the string class module, `str.cc` and `str.h`.

When you have a second module like the string class module, both the implementation part of the second module and the main program module must include the header file for the second module. For example, `testr.cc` and `str.cc` include the header file `str.h` with a line like this one:

```
#include "str.h"
```

If there is not an object file for the second module, you can compile the second module and link it with the program with a command line like this one:

```
tutorial% CC testr.cc str.cc -o testr
```

The order of the files is not significant.

Alternately, you can create an object file for the second module with a command line like this one:

```
tutorial% CC -c str.cc
```

2. **To include .C in your default , add .C .C~ to the end of the SUFFIXES macro.**

3. **Next, copy these lines from** `default.mk` **.**

```
.cc:
          $(LINK.cc) -o $@ $< $(LDLIBS)
     .cc.o:
          $(COMPILE.cc) $(OUTPUT_OPTION) $<
     .cc.a:
          $(COMPILE.cc) -o $% $<
          $(AR) $(ARFLAGS) $@ $%
          $(RM) $%
```

4. **Add them to your makefile, replacing** `.cc` **with** `.C` **(or whatever file extension you wish to use).**

5. **If you are editing** `default.mk`**, add these lines to the end of the file.**

```
.C:
          $(LINK.cc) -o $@ $< $(LDLIBS)
     .C.o:
          $(COMPILE.cc) $(OUTPUT_OPTION) $<
     .C.a:
          $(COMPILE.cc) -o $% $<
          $(AR) $(ARFLAGS) $@ $%
          $(RM) $%
```

Since `.c` is supported as a C-language suffix, it is the one suffix that cannot be added to the `SUFFIXES` macro to support C++. Write explicit rules in your own makefile to handle C++ files with a `.c` suffix.

This message may be explained by noting that `make` examines the exit status of each program that it invokes where the program's exit status is the value returned by `main()` or passed to `exit()`. If `main()` does not call `exit()`, or `return` explicitly, the exit status is undefined and may cause `make` to fail.

4.4 Using the Complex and Task Libraries

Give `CC` an extra option when you use the complex math or the task (co-routine) library. You must give this extra option because these libraries call functions in `libm`, the standard math library. For example, `cos(complex)` in the complex library calls `cos(double)` if you use the complex library:

```
tutorial% CC yourFile -lcomplex
```

or, if you use the task library:

```
tutorial% CC yourFile -ltask
```

4.5 Predefined Macro

You can use the `__cplusplus` macro to mix C and C++ code. For example,

```
#ifdef __cplusplus
int printf(char*...); // C++ function declaration
#else
int printf();/* C function declaration */
#endif
```

Note – There are two underline characters at the beginning of `__cplusplus`

See Section 3.7, "The __cplusplus Directive," on page 61 and Appendix E, "Creating Generic Header Files" for more information.

4.6 Static Linking of `libC`

The `CC` driver links in several libraries by default by passing `-l` options to `ld`. On 4.1.x, the driver passes `-lm -lansi -lC -lc` to ld. On 5.0, the driver passes `-lm -lC -lc` to `ld`. This occasionally causes problems because the shared version of `libC` gets linked by default. Since the shared library `libC.so` is not bundled with the operating

- Linker `ld` using `-qoption` or `-qpath` (see the `CC`.1 manual page)

Before you use the `CC` command, insert `/opt/SUNWspro/bin` (or the name of the directory in which you have chosen to install the C++ translator) at the beginning of your search path. This is usually done in the `.cshrc` file, in a line with `set path =` at the start; or in the `.profile` file, in a line with `PATH=` at the start. (Applies to SunOS 5.0 only).

Before you use the `man` command, insert `/opt/SUNWspro/man` (or the name of the directory in which you have chosen to install the C++ translator) at the beginning of your search path. This is usually done in the `.cshrc` file, in a line with `setenv MANPATH=` at the start; or in the `.profile` file, in a line with `export MANPATH=` at the start. (Applies to SunOS 5.0 only).

The options for those programs do not conflict. All compiler options are position independent except `-Bstatic` and `-Bdynamic` .

-a

Prepares object code for coverage analysis using `tcov` .

`-bsdmalloc` (*SunOS 5.0 only*)

Directs the compiler to link in calls to `malloc` from the library `libbsdmalloc.a`. When invoked, causes flags `-u_malloc` and `/lib/libbsdmalloc.a` to be passed to the linker.

`-B`*binding*

Specifies whether bindings of libraries for linking are static or dynamic, indicating whether libraries are nonshared or shared. The possible values for `binding` are static and dynamic. The default is dynamic.

`-c`

Directs `CC` to suppress linking with `ld` and produce a `.o` file for each source file. You can explicitly name a single object file with the `-o` option.

`-cg[87,89,92]`

Code generator. Generates code that runs on both the older and the newer Sun-4 systems or on only the newer Sun-4 systems. There is one option under SunOS 5.0 and three under 4.1.x.

Use `fpversion (1)` to tell you which floating-point hardware you have. It may take about a minute to display its report.

If you are building a shared library with `-cgx` and `-pic`, then there is no load-time check for any modules miscombining with other `-cgx` options. You must do the check yourself.

`+d`

Prevents the compiler from expanding in-line functions. Use this option if you want to debug in-line functions. For maximum flexibility, this option is not automatically invoked when you specify the debugging (`-g`) option. This option now operates as it did in C++ 2.1. In this respect, and for this option, the behavior of C++ 3.0.1 is identical to C++ 2.1.

`-dalign`

Generates `double load` and `store` instructions whenever possible for improved performance. It assumes that all double-typed data are double aligned, and should not be used when correct alignment is not assured.

`-dryrun`

Directs `CC` to show but not execute the commands constructed by the compilation driver.

`-D`*name* [=*def*]

Defines a symbol *name* to the preprocessor `acpp`. This is equivalent to a `#define` directive at the beginning of the source. If you don't use `=def`, `name` is defined as '1'. You may give multiple `-D` options.

`-E`

Tells `CC` to run only `acpp` and to send the result to the standard output.

`+e`*number*

Since release 2.0, the `+e`*number* option is honored only if the new virtual table optimizations cannot be automatically employed by the compiler. The option lets you optimize your program manually to use less space. It ensures that only one virtual table is generated per class.

The C++ compiler in almost all cases generates one virtual table per class per executable regardless of how often it sees any particular class definition; using this option is rarely necessary (see the C++ *3.0.1 Language System Release Notes* manual).

Use the `+enumber` option on classes where virtual functions are present and all the virtual functions are either defined as `inline` or `pure.` `number` can be `0` or `1`.

This is a convenience option that chooses the fastest code generation option available on the compile-time hardware, the optimization level `-O2`, a set of inline expansion templates, the `-fnonstd` floating-point option, and on a SPARCstation, the `-dalign` option.

If you combine `-fast` with other options, the last specification applies. The code generation option, the optimization level, and using in-line template files can be overridden by subsequent switches. For example, although the optimization part of `-fast` is `-O2`, the optimization part of `-fast -O3` is `-O3`.

Do not use this option for programs that depend on IEEE standard exception handling; you can get different numerical results, premature program termination, or unexpected `SIGFPE` signals.

`-g`

Produces additional symbol table information for the debugger.This also causes the C++ compiler to produce C code for every declaration in the compilation rather than only for those declarations that are needed or used. This additional information enables easier debugging, but also increases the size of the object file because the symbol table is larger. The `+d` option is no longer turned on automatically when you select `-g`. This provides you with more control when you debug your code. When you debug in-line functions, you must also select the `+d` option.

`-G` (`SunOS` *5.0 only*)

Builds a shared library (see the `ld(1)` manual page). All source files specified in the command line are compiled with `-pic`. Also, `-dy`, `-G`, `-z text` options are passed to `ld` if `-c` is not specified.

`-H`

Prints, one per line, the path name of each file included during the current compilation on the standard error output. This option is processed by `acpp`.

`-h` *name* (`SunOS` *5.0 only*)

Names a shared dynamic library. Provides a way to have versions of a shared dynamic library. In general, the name after `-h` should be exactly what you have after the `-o`. The space between the `-h` and *name* is optional. This is a loader option.

Do not use the `-L` *directory* option to specify `/usr/lib` or `/usr/ccs/lib`, since they are searched by default and including them here prevents using the unbundled `libm`. Do not use `LD_LIBRARY_PATH` to do this either, for the same reasons.

Problem: Library not Found

You may get the following error message while executing any program.

```
ld.so: library not found
```

This happens *during* the running of `a.out`, not during compilation or linking.

Solution

Set `LD_LIBRARY_PATH` to include the directory where the missing library resides. It is usually better to add the directory to the list of paths, rather than replacing the whole list of paths with the one directory.

As an example of the problem, if you are using OpenWindows and you define the `LD_LIBRARY_PATH` environment variable to link in the Xview libraries, and if you get the above error message while executing your program, then you can fix the problem by setting the variable: `LD_LIBRARY_PATH`.

Do *not* include `/usr/lib` or `/usr/ccs/lib` here, since they are searched by default, and including them here prevents using the unbundled `libm`.

Example: Set `LD_LIBRARY_PATH`.

In `sh` under SunOS 5.0:

```
demo$ LD_LIBRARY_PATH=/opt/SUNWspro/SC2.0.1 :"$LD_LIBRARY_PATH"
demo$ export LD_LIBRARY_PATH
```

In `csh` under SunOS 5.0:

```
demo% setenv LD_LIBRARY_PATH /opt/SUNWspro/SC2.0.1:"$LD_LIBRARY_PATH":
```

In `csh` under SunOS 4.1.x:

```
demo% setenv LD_LIBRARY_PATH /usr/lang/SC2.0.1:"$LD_LIBRARY_PATH":
```

Background

Do only the minimum amount of optimization (peephole). This is postpass assembly-level optimization. Do not use `-O1` unless `-O2` and `-O3` result in excessive compilation time, or running out of swap space.

`-O2`

Do basic local and global optimization. This is induction-variable elimination, local and global common subexpression elimination, algebraic simplification, copy propagation, constant propagation, loop-invariant optimization, register allocation, basic block merging, tail recursion elimination, dead code elimination, tail call elimination and complex expression expansion.

The `-O2` level does not optimize references or definitions for external or indirect variables. Do not use `-O2` unless `-O3` results in excessive compilation time, or running out of swap space. In general, the `-O2` level results in minimum code size.

`-O3`

In addition to optimizations performed at the `-O2` level, this also optimizes references and definitions for external variables. The `-O3` level does not trace the effects of pointer assignments. Do not use `-O3` when compiling either device drivers, or programs that modify external variables from within signal handlers. In general, the `-O3` level results in increased code size.

`-O4`

In addition to optimizations performed at the `-O3` level, this also does automatic in-lining of functions contained in the same file; this usually improves execution speed, but sometimes makes it worse. In general, the `-O4` level results in increased code size.

For most programs:

`-O4` is faster than `-O3`

`-O3` is faster than `-O2`

`-O2` is faster than `-O1`

In a few cases `-O2` may perform better than the others, and `-O3` may outperform `-O4`. Try compiling with each level to see if you have one of these rare cases.

`-P`

Runs the source file through `acpp`, the preprocessor, only. It then puts the output in a file with a `.i` suffix. Does not include `acpp`-type line number information in the output.

`+p`

Disallows all anachronistic constructs. See the C++ *3.0.1 Language System Product Reference Manual* for all disallowed anachronisms under this option.

`-pg`

Prepares the object code to collect data for profiling with `gprof`. It invokes a runtime recording mechanism that produces a `gmon.out` file at normal termination.

`-pic`

Produces position-independent code. Each reference to a global datum is generated as a dereference of a pointer in the global offset table. Each function call is generated in pc-relative addressing mode through a procedure linkage table. The size of the global offset table is limited to 8Kbytes on SPARC stations.

`-PIC`

This option is similar to `-pic`, but lets the global offset table span the range of 32-bit addresses in those rare cases where there are too many global data objects for `-pic`.

`-pipe`

Directs `CC` to use pipes, rather than intermediate files, between compilation stages (very CPU-intensive).

Templates

The template instantiation system adds several options to `CC`. These are specified on the `CC` line or by setting the environment variable PTOPTS. For example, to permanently enable verbose mode, you would say:

```
demo: export PTOPTS=-ptv                    {in the .profile file}
demo: setenv PTOPTS -ptv                    {in the .cshrc file}
```

-pta

Prepares object code to collect data for profiling with `lprof` (see the `lprof(1) man` page).

`-qp`

Prepares the object code to collect data for profiling with `prof` (see the `prof(1) man` page). Invokes a runtime recording mechanism that produces a `mon.out` file (at normal termination).

`-Qpath` or `-qpath` *pathname*

Inserts a directory path name into the search path used to locate compiler components. This path will also be searched first for certain relocatable object files that are implicitly referenced by the compiler driver, for example, `*crt*.o` and `bb_link.o` . This lets you choose whether or not to use default versions of programs invoked during compilation.

`-Qproduce` or `-qproduce` *sourcetype*
Causes `CC` to produce source code of the type sourcetype. Sourcetype can be one of the following:

`.cc`
C source (from `cfront`).

`.i`
Preprocessed C++ source from `acpp`.

`.o`
Object file from `fbe`, the assembler.

`.s`
Assembler source (from `acomp`, or `fbe`).

`-R` *path* (`SunOS` *5.0 only*)
A colon-separated list of directories used to specify library search directories to the run-time linker. If present and not null, it is recorded in the output object file and passed to the run-time linker. If both the `LD_RUN_PATH` and the `-R` option are specified, the `-R` option takes precedence.

`-S`
Directs `CC` to produce an assembly source file but not to assemble the program.

`-sb`
Directs `CC` to generate SourceBrowser database.

Allows the use of the `$` (dollar sign) character in identifier names. Unlike C, `$` cannot be the first character of an identifier in C++.

`-xs`

Places symbol table information in the executable. Without this option, the symbol table information is kept in `.o` files. This option increases the size of the executable.

5.1 Introduction

C++, like C, has no built-in input or output statements. The standard C++ I/O library is iostream.

As with much of object-oriented programming, discussions of iostreams may be somewhat circular and may be difficult to understand without knowing more about the topic. A terminology section at the end of this chapter defines many of the basic terms you need to know. You can to refer to that section as you progress through this chapter.

Using Iostreams with `stdio`

You can use `stdio` with C++ programs, but problems can occur when you mix `iostreams` and `stdio` within a program. To eliminate this problem, execute the following:

```
cin.sync_with_stdio()
```

This will connect the predefined iostreams with the corresponding stdio `FILE`s. Such connection is not the default because there is a significant performance penalty when the predefined files are made unbuffered as part of the connection.

5.2 Basic Structure of Iostream Interaction

The iostream package allows a program to use any number of input or output streams. Each stream has some source or sink, which might be standard input, standard output, or a file. A stream can be restricted to input or output. The iostream package implements these streams using two processing layers, or a single stream can allow both input and output.

The lower layer implements *sequences*, which are simply streams of characters. These sequences are implemented by the `streambuf` class.

The upper layer performs formatting operations on sequences. These formatting operations are implemented by the `iostream` class, which has as one of its members an object of type `streambuf`.

Standard input and output are handled by objects of class `iostream`.

Output Using Iostreams

Output using iostream usually relies on the overloaded leftshift operator <<, which, in the context of iostream, is called the *insertion operator.* To output a value to standard output, you insert the value in the predefined iostream `cout`. For example, given a value `someValue`, you send it to standard output with a statement like

```
cout << someValue;
```

The insertion operator is overloaded for most (but not all) built-in types, and the value represented by `someValue` is converted to its proper output representation. If, for example, `someValue` is a `float` value, the << operator converts the value to the proper sequence of digits with a decimal point. Where it inserts `float` values on the output stream, << is called the *float inserter.* In general, given a type `X`, << is called the *X inserter.*

The format of output and how you can control it is discussed later in this chapter in the section "Format Control."

The iostream package does not, of course, know about user-defined types. If you define types that you want to output in your own way, you must define an inserter (that is, overload the << operator) to handle them correctly.

The operator << can be applied repetitively; to insert two values on `cout`, you can use a statement like this one:

```
cout << someValue << anotherValue;
```

This will have no space between the two values, though, so you might want to do this:

```
cout << someValue << " " << anotherValue;
```

The << operator has the precedence of the left shift operator (its built-in meaning). As with other operators, you can always use parentheses to guarantee the order of action. It may be a good idea to always use parentheses to avoid problems of precedence. Of the following four statements, the first two are equivalent, but the last two are not.

```
cout << a+b;    //+ has higher precedence than <<
cout << (a+b);
cout << (a&y); //but << has precedence higher than &
cout << a&y;
```

`error`, which takes a string and aborts the program. `error` is not a predefined function. (See Section , "Handling Input Errors," on page 102 for an example of an `error` function.) You can examine the state of an iostream with the operator `!`, which will return a nonzero value if the iostream is in an error state. For example:

```
if (!cout) error("aborted due to output error");
```

There is another way to test for errors. The `iostream` class defines `operator void *()` so it returns a NULL pointer when there is an error. This allows you to use a statement like:

```
if (cout << x) return ;
```

You can also use the function `good`, a member of `iostream`:

```
if ( cout.good() ) return ;
```

The error bits are declared in the enum:

```
enum io_state { goodbit=0, eofbit=1, failbit=2,
     badbit=4, hardfail=0200} ;
```

For details of this as well as the error functions see the `man` pages.

Flushing

As with most I/O packages, iostream often accumulates output and sends it on in larger and generally more efficient chunks. If you want to flush the buffer, you simply insert the special value `flush`. For example,

```
cout << "This needs to get out immediately." << flush ;
```

Note – If you want to use any manipulators, you must include the header file `.iomanip.h`

`flush` is an example of a kind of object known as a *manipulator,* which is a value that may be inserted into an iostream to have some effect other than causing output of its value. It is really a function that takes an `ostream&` or `istream&` argument and returns its argument after performing some actions on it (see Section 5.8, "Manipulators," on page 109).

Class `string` (defined in Section , "Output Using Iostreams," on page 96 and more completely in Appendix A, "Sample Program") defines its extraction operator like this:

```
istream& operator>> (istream& ios, string& input)
     {char holder[256];
     ios.get(holder, 256 , "\en");
     string got( holder );
     input = got;
     return ios;
     }
```

By convention, an extractor converts characters from its first argument (in this case, `istream& ios`), stores them in its second argument (always a reference), and returns its first argument. The second argument must be a reference because an extractor is meant to store the input value in its second argument.

The `char*` *Extractor*

This predefined extractor is mentioned here because it can cause problems. You use it like this:

```
char x[50];
cin >> x;
```

This extractor skips leading white space and extracts characters and copies them to `x` until it reaches another white space character. It then completes the string with a terminating null (0) character. Be careful because input can overflow the given array.

Handling Input Errors

By convention an extractor whose first argument has a nonzero error state should not extract anything from the input stream and should not clear any error bits. However, an extractor that fails can and should set at least one error bit. The `string` extractor shown previously does not explicitly follow these conventions. Nevertheless, because it only modifies the iostream using other extractors that do follow the conventions (as all the predefined extractors do), the conventions are implicitly followed. You can also follow that strategy.

As with output errors, you should check the error state periodically and take some action (such as aborting) when you find a nonzero state. The `!` prefix operator returns the error state of an `iostream`. For example, the following code produces an input error if you type alphabetic characters for input:

```
#include <stream.h>
void error (char* message)
{    cout << message << "\n" ;
     exit(1);
}
main()
{    cout << "Put in some characters: ";
     int bad;
     cin >> bad;
     if (!cin) error("aborted due to input error");
     cout << "If you see this, not an error." << "\n";
}
```

Class `iostream` has member functions that you can use for error handling. See the `iostream` `man` pages for details.

5.4 Predefined Iostreams

There are four predefined iostreams: the two mentioned earlier, `cin` and `cout`, and two others, `cerr` and `clog`.

Both `cerr` and `clog` are connected to standard error, but `clog` is buffered while `cerr` is not.

Open Mode

The mode is constructed from the `open_mode` `enum`, which has the definition:

```
enum open_mode {in=1, out=2, ate=4, trunc=20, app=010,
     nocreate=040, noreplace=0100};
```

For compatibility reasons, the following constants (used for `open modes`) are defined:

- `static const int input = (ios::in);`
- `static const int output = (ios::out);`
- `static const int append = (ios::app);`
- `static const int atend = (ios::ate);`

You can open a file for input and output simultaneously. For example:

```
fstream inoutFile("someName", input|output);
```

Declaring an `fstream` *without a File*

You can declare an `fstream` without specifying a file and open the file later. For example,

```
fstream toFile;
toFile.open(argv[1], output);
```

Opening and Closing Files

You can close the `fstream` and then open it with another file. For example:

```
fstream infile;
for (char** f = &argv[1]; *f; ++f) {
infile.open(*f, input);
...;
infile.close();
}
```

`seekg(seekp)` can take one or two parameters. When it has two parameters, the first is a position relative to the position indicated by the `seek_dir` value given as the second parameter. For example, the following code moves to 10 bytes from the end:

```
aFile.seekp(-10, ios::end);
```

While this second example moves to 10 bytes forward from the current position:

```
aFile.seekp(10, ios::cur);
```

5.6 Assigning Iostreams

Earlier versions of C++ allowed assignment of one stream to another. This is no longer allowed. The C++ *3.0.1 Language System Library Manual* briefly discusses this:

> Assignment of streams is not possible in general but the predefined streams have special types which allow it.

This problem is discussed also in the C++ *3.0.1 Language System Library Manual*:

> The old stream library allowed assignment of one stream to another. Such assignments should be changed to user pointers or references to streams in iostreams.

The problem with copying a stream object is that there are two versions of the state information (such as a pointer to the current write point within an output file), which may be changed independently. This could cause havoc.

Although Stroustrup's book, *The* C++ *Programming Language*, indicates that it is possible to copy streams, he implies that this is usually used for initialization. This is borne out by the available examples, such as:

```
cout = cerr
```

There is no need (beyond initialization) to copy stream objects. Most streams (such as `fstream`) need no object copy at all.

If, however, we replace `void print(fstream b)` with `void print(fstream &b)`, then it compiles without error as follows:

```
tutorial% CC t.c
cc -Wl,-L/c++/cfront/2.00 t.c -lC
```

If `fstream` were an ordinary class, passing it would create a new instance of `fstream` and initialize it by doing a member by member copy from instance `a`. It turns out, however, that `iostream` has carefully defined `operator=` and `ios(ios&)` as private to prevent this default behavior.

Consider another case. The following test code example:

```
#include <stream.h>
void foo(ostream s) {
     s << "Hello\n";
}
main() {
     foo(cout);
}
```

will cause a compilation error:

```
tutorial% CC report2.c
"report2.c", line 9: error: ostream::ostream() cannot
     access ios::ios(): private member
1 error
```

If you tried hacking `iostream.h` to make `ios::ios()` public, you would get a linker error for no definition for `ios::ios()`. If you looked closely at source code, you would realize that you could change the functions expecting `ostream` to take `ostream &`.

5.7 Format Control

Format control is discussed in detail in the *C++ 3.0.1 Language System Library Manual* and in the `IOS` man page.

To use predefined manipulators, you must include the file `iomanip.h` in your program.

You can easily define your own manipulators and parameterized manipulators. There are three basic types of manipulators:

- *Macro-type manipulators,* which use `#define` statements.
- *Plain manipulators,* which take an `istream&` or `ostream&` argument, operate on the iostream, and then return `istream&` or `ostream`. You use a plain manipulator by inserting it into or extracting it from an iostream.
- *Parameterized manipulators*
  which are functions that return other manipulators.

The following subsections give examples of each type.

Manipulators Using Macros

Here is an example of a macro manipulator that simply inserts a newline.

```
#define eol "\n" << flush
cout << "y = " << y << eol
```

Parameterized Manipulators

One of the manipulators that is not included in iostream sets the fill character controlled by the format state variable `fill`.

Here is a definition for the parameterized manipulator `setfill`:

```
ostream& ios_setfill(ostream& ios, int f) {
     ios.fill(f);
     return ios;
     }
ioap setfill = ios_setfill;
```

An `ioap` is a class that looks like a function. The type of the result of applying an `ioap` to an int is an `iomanip`. An `iomanip` is a data structure that contains both the functional value and the `int` parameter. The inserter (and extractor) for `iomanip` applies the functional value in the obvious way. For example, when an insertion or extraction operator is invoked, as in the following code:

this purpose. For example, to declare types analogous to `ioap` and `iomanip` for use with a manipulator with an extra `iostream*` argument, use a statement like

```
IOMANIP(iosptr_manip, iosptr_ap, ostream*);
```

This declares two classes, `iosptr_manip` and `iosptr_ap`, for use with an extra `ostream*` argument.

Here is an example of a field manipulator for floating point-numbers:

```
IOMANIP(dbl_manip, dbl_ap, double);
ostream& do_dbl(ostream& ios, double x)
{
     int oldp = ios.precision();
     ios.precision(12);
     ios << x;
     ios.precision(oldp);
     return ios;
     }
dbl_ap dfield = do_dbl;
cout << dfield(3.14);
```

To create a manipulator with two extra arguments, use the macro `IOMANIP2`. (There are no macros for defining manipulators with more than two extra arguments.) For example:

```
IOMANIP2(icp_manip, icp_ap, int, char*);
ostream& repeat_str(ostream&, ios, int n, char* s) {
     while (ios && --n >= 0 ) ios << s;
     return ios;
     }
icp_manip tentimes(repeat_str, 10);
cout << tentimes("a");
```

This produces the following output:

```
aaaaaaaaaa
```

5.11 Streambufs

Iostreams are actually the formatting part of a two-part input/output system. The other part of the system is made up of *streambufs*, which deal in input or output of unformatted streams of characters.

You usually use streambufs through iostreams, so you don't have to worry about the details of streambufs. However, you can use streambufs directly if you choose to, for example, if you need to improve efficiency or to get around the error handling or formatting built in to `iostreams`.

How Streambufs Work

A streambuf consists of a stream or *sequence* of characters and one or two pointers into that sequence. One of the two possible pointers is a *put* pointer, while the other is a *get* pointer. A streambuf can have one or both of these pointers.

Position of Pointers

Each pointer points between two characters; the get pointer points just before the next character that will be fetched; the put pointer points just before the position of the next character delivered. (You can also think of the position of the put pointer as just after the last character delivered, but the pointer may have been moved since the last character was actually delivered or may be moved before the next character is delivered.)

The positions of the pointers and the contents of the sequences can be manipulated in various ways. Whether or not both pointers move when one pointer moves depends on the kind of streambuf used. Generally, with *queue-like* streambufs, the get and put pointers move independently; with *file-like* streambufs the get and put pointers always move together.

Using Streambufs

See the *C++ 3.0.1 Language System Library Manual* for information on using streambufs.

Describes the interface needed by programmers who are coding a class derived from class *streambuf*. You may also want to see `sbuf.pub(1)`, because some public functions are not discussed in this `man` page.

`sbuf.pub`

Details the public interface of class `streambuf`. In particular, this `man` page describes the public member functions of `streambuf`.

This `man` page contains the information you need if you want to use a `streambuf`-type object directly, or if you want to find out about functions that classes derived from `streambuf` inherit from it. If you want to derive a class from `streambuf`, see `sbuf.prot`.

`stdiobuf`

Contains minimal description of class `stdiobuf`, which is derived from `streambuf` and specialized for dealing with `stdio` FILE. See the `sbuf.pub(1)` and `sbuf.prot(1)` `man` pages for details of features inherited from class `streambuf`.

`strstream`

Details the specialized member functions of strstreams, which are implemented by a set of classes derived from the iostream classes and specialized for dealing with arrays.

`ssbuf`

Details the specialized public interface of class `strstreambuf`, which is derived from `streambuf` and specialized for dealing with arrays. See the `sbuf.pub(3)` and sbuf.prot(3) man pages for details of features inherited from class `streambuf`.

5.13 Iostream Terminology

The iostream package has similar or identical terms that are used differently. This section defines those terms as they are used in discussing the iostream package.

Buffer

A word with two meanings, one specific to the iostream package and one more generally applied to input and output.

Class `iostream` has an object of class `streambuf` as a member. A streambuf presents a simple sequence of characters that may have pointers for input and output associated with it. An iostream uses the `streambuf` and adds formatting so you can deal with output or input of specific data types (including class types). In addition, output iostreams can take formatting commands to change the way printed information appears.

Iostream package
The package implemented by the include files `iostream.h`, `fstream.h`, `strstream.h`, `iomanip.h`, and `stdiostream.h`. In the nature of object-oriented packages, this is intended to be extended by programmers who use it; some of what you can do with this package is not actually implemented in it.

Pipestream
An iostream specialized as a circular queue.

Stream
An iostream, fstream, strstream, pipestream, or user-defined stream in general.

`streambuf`
An object of class `streambuf` (printed in `courier` font).

Streambuf
A buffer that contains a sequence of characters with a put or get pointer, or both (printed in default font). Generally an object of class `streambuf` or a class derived from `streambuf`.

Strstream
An iostream specialized for use with arrays.

6.1 Structure of the Co-Routine Classes

The co-routine library provides six basic kinds of objects.

Table 6-1 Six Basic Objects in a Co-routine Library

Object	Action
Tasks	Co-routines. When you want to create a task, you derive a class from the predefined class task. You put the action or program of the task in the constructor of the new class.
Schedulers	Control the basic operation of a program, specifically choosing which task runs next. There is one scheduler per program.
Queues	Data structures that allow you to make ordered collections of objects.
Timers	Classes that allow you to implement timeouts and other time-dependent functions.
Histograms	Data structures provided to help gather data.
Interrupt handlers	Classes that represent external events.

In addition, two important base classes are defined.

Table 6-2 Two Base Classes in a Co-routine Library

Object	Action
Class object	Provides a basic definition of an object.
Class sched	Provides a basic definition for an object that knows about time. Used as a base class for the classes `timer` and `task`, as well as being the class for schedulers.

6.2 Objects

The co-routine library defines class object as a base class for every other class in the library. You can derive from object yourself; in particular, messages passed between tasks are usually instances of classes derived from class `object`. (Queues, which often store messages, can only store `object`-type objects.)

Class `task`

A task is an object of a class derived from class `.task` The action of a task is contained in the constructor of the task's class; if a task is like a process then the constructor is like the program running in the process. Because of the nature of co-routine programming, the constructor of a task never completes until the program as a whole completes.

A task is always in one of three states:

- RUNNING — Executing instructions or on the scheduler's ready-to-run list.
- IDLE — Suspended; that is, waiting for something to happen before returning to RUNNING.
- TERMINATED — Completely done running. It cannot return to a RUNNING or IDLE state. However, it is not completely dead because another task can access its result.

Parts of a Task

This table discusses each line of the public part of class `task`.

`task(char* =0, int =0, int =0`	Constructor for class `task`. Every derived class has its own constructor that contains the "program" of the task. When you create an object of your derived class, you can optionally pass parameters to `task()`.
`~task()`	Destructor for class `task`. This takes care of default destruction. `task* t_next` inserts the class in `.task_chain` `task_chain` is a chain of tasks created by `task()`. A new task is placed at the start of `task_chain`. It is used by `task()` and `~task()`.
`char* t_name`	String naming the task provided for use by debugging aids and error reporting functions. Value of the optional first parameter of `.task()`. A task does not have to have a name.
`waitvec to sleep`	Functions dealing with suspending this task. Discussed in Section , "Waiting States for Tasks" on page 126.
`void resultis(int)`	Function that returns the result of the task and puts the task in a TERMINATED state. Takes the place of the usual function return mechanism and, in fact, you *cannot* use `return`.
`void cancel(int)`	Puts the task into the terminated state and sets the return value just like `resultis` does. However, `cancel` does not invoke the scheduler, so a task can call `cancel` on another task and still retain control.
`void swap_stack(-int*, int*, int*,int*,int*);`	Function that the scheduler calls when it "wakes up" the task. Restores the stack frame and other features of the task environment.

The implementation of the constructor `getString()` is very simple:

```
getString::getString () {
     char aString[256];
     cout << "Enter String: ";
     cin >> aString;
     resultis ( (int) aString);
}
```

The declaration for `countDollar` is also simple:

```
countDollar::countDollar (getString *theGetter) {
          char *s;
          register int i = 0;
          register char c;
     s = (char*) (theGetter->result ());
          while ( c = *s++ ) {
                  if (c == ’$’) i++;
     }
     resultis (i);
}
```

The main program looks like this:

```
void main() {
     getString getter;
     countDollar counter ( &getter);
     cout << "Result is: "<<counter.result()<<"\n";
     thistask->resultis(0); //the main routine is also a
                // task and should be terminated by resultis()
}
```

Waiting States for Tasks

When a task needs to wait, generally for some other task to take some action or produce some information, it needs to change its state to IDLE. Later, when the condition that led to its suspension no longer exists, the task needs to change its state back to RUNNING. The definition of class `task` provides a number of means to achieve that behavior.

You can put a task to sleep by calling the following void function:

```
void sleep(object* t = 0)
```

A task calls `sleep()` on itself. The calling task goes to sleep until the object pointed to by the parameter is no longer pending. If the task is not pending when you execute this call, the calling task goes to sleep indefinitely.

If you don't give a pointer as follows, your task goes to sleep indefinitely:

```
sleep();
```

Waiting for an Object

A task can wait for another task to take some unspecified action. You do so with the `wait()` task member function.

You can make a task wait by calling:

```
void wait(object* ob);
```

A task calls `wait()` on itself. The calling task waits until the object pointed to by the parameter is no longer pending. If the task is not pending when you execute this call, the calling task continues execution immediately.

If you give a null pointer as follows, your task waits indefinitely:

```
wait(0);
```

Waiting for a List of Tasks

Tasks have two member functions that let them wait for one of a list of pending objects to become no longer pending. The two functions are:

```
int waitlist(object* ...);
int waitvec(object**);
```

You give `waitvec` a list of objects to wait for. They can be queues or tasks. For example:

```
qhead* firstQ;
qhead* secondQ;
taskType* aTask;
. . .
int which = waitlist(firstQ, secondQ, aTask, 0);
```

A more concurrent way to write these tasks is to give them a different way of passing information and let each routine loop indefinitely. For example, you could write `countDollars()` like this:

```
countDollars::countDollars()
{
while (1)
{
      //get a string somehow
      //process the string
      //pass the total on
}
```

Appendix B, "Co-Routine Examples," gives the full text of a program written this way.

The mechanism provided by the co-routine package for such intertask communication is embodied in *queues*. A queue is a data structure made up of a series of linked objects. Two kinds of queues are: circular queues and first-in-first-out (FIFO) queues with a head and a tail.

Both kinds of queues can hold only descendants of type `object`.

FIFO Queues

A FIFO queue is made of two objects: a `qhead` and a `.qtail.` You create a queue by creating a `qhead` object for it. You then create a tail by calling the member function of `qhead` :

```
qtail* tail();
```

You can place objects on the queue with the member function of `qtail` (the return value is `1` if the action is successful) :

```
int put(object*)
```

and take objects from the queue with the member function of `qhead`:

```
object* get()
```

You can also put an object back at the head of the queue with the `qhead` member function:

```
int putback(object*)
```

You can use this to treat a queue head like a stack.

The implementation for `countDollars` is:

```
countDollars::countDollars(qhead *stringQ,qtail * countQ)
{
     register char c;
     stringHolder *inmessage;
     while (1) {
          inmessage = (stringHolder *) stringQ->get();
          char *s = inmessage->theString;
          register int i = 0;
          while (c = *s++)
               if (c == ’$’) i++;
          numDollars *num = new numDollars(i);
     countQ->put(num);
}
};
```

Since `countDollar` is first created in the main program (which is different from the original version, where it couldn’t be created first), when `countDollar()` tries to get a message from the queue, `countDollar` suspends because there is no message. This is because it is the default waiting-type queue. At this point, the main program creates the string getter. Here is the implementation of `getString()`:

```
getString::getString(qhead *countQ,qtail* stringQ)
{
     numDollars * cmessage;
     while (1) {
          cout << "Enter a string. Use Control-C to end
          session. ";
          char aString[256];
          cin >> aString;
          stringQ->put(new stringHolder(aString));
          cmessage = (numDollars *) countQ->get();
          printf("The number of dollar signs was %d\n"
               ,cmessage->dollars);
     }
};
```

You can find out how many objects are in a queue with the `qhead` member function int rdcount().

You can find out how many more objects can be inserted in a queue with the `qtail` member function int rdspace().

6.5 *The Scheduler*

Although you don't deal directly with the task scheduler, it oversees the life of tasks; you may need to know some of the principles under which it operates.

- The main activity of the scheduler is maintaining the *run chain.* The run chain is the list of tasks that have state RUNNING and therefore are ready to run.
- The scheduler runs "in between" tasks. In other words, it does what it has to do after a task has given up execution and before it starts up the next task on the run chain.
- When a task changes its state from IDLE to RUNNING, the scheduler adds it to the end of the run chain.
- When a task gives up execution but does not change its state (still has the state RUNNING), the scheduler puts it on the end of the run chain.
- The scheduler cannot preempt a task (also, a task cannot preempt another task). The currently running task stops execution only when it wants to or when it asks for information that is not yet available.
- If the run chain is empty and there are no interrupt handlers, the scheduler exits because no task can become RUNNING.

6.6 *Task Library Limitations*

The task library is "flat" in the sense that a class derived from `task` may not have derived classes. That is, only one "level" of derivation is allowed. This is not a bug; this is the way the library was designed, and reflects the way the tasks are manipulated on the stack. The enhancement of allowing multiple levels would require a rewrite of the current implementation.

If you need to have the certain sets of tasks share information, a multiple inheritance scheme needs to be adopted.

```
#include <task.h>

const int NO_OF_TASKS = 2;
const int MAX_ITERATIONS = 5;

class task_info_to_share {
    static int task_count;
    int sharedinfo;
protected:
    task_info_to_share (){
        if (task_count)
            task_count++;
        else
            task_count= NO_OF_TASKS;
        // main is created with the 1st task
        sharedinfo = 0; }
public:
    static int get_task_count(){
        return task_count; }
    int get_sharedinfo(){
        return sharedinfo; }
    int set_sharedinfo(int i){
        int info = sharedinfo;
        sharedinfo= i
        return info; }
};

//Caveat: members of class task_info_to_share will not
//be accessible via the thistask pointer, since that is
//only a pointer to a task.

//Use of multiple inheritance here is used to share
//information from class task_info_to_share. Note that
//this is one flat level of inheritance.

struct pc : public task, public task_info_to_share {
        pc(char*, qtail*, qhead*);
};
```

The output of the previous is as follows:

```
main
new pc(a)
task_count = (2)
main()'s loop
new pc(b)
task_count = (3)
main()'s loop
new pc(first pc)
task_count = (4)
main: task_count = 4
main: task_chain is:
===================================================
task
task first pc (IDLE) this = d350:
===================================================
task
task b (IDLE) this = d2a8:
===================================================
task
task a (IDLE) this = d1b0:
====================================================
task
task main (is thistask, RUNNING): this = b450:
====================================================
task
task Interrupt_alerter (IDLE) this = ab38:

main: here we go
main: exit
task b
task a
task first pc
task b
task a
task first pc
task b
task a
task first pc
task b
task a
task first pc
```

7.1 *Type Complex*

The complex arithmetic library defines one class: type `complex`. An object of type `complex` can hold a single complex number. The complex number is constructed of two parts: the real part and the imaginary part. The numerical values of each part are held in `double` fields. Here is the relevant part of the definition of type `complex`:

```
class complex {
     double    re, im;
```

The value of an object of type `complex` is a pair of `double` values. The first value represents the real part; the second value represents the imaginary part.

Constructors of Type `complex`

There are two constructors for type `complex`. Their definitions are:

```
complex()       { re=0.0; im=0.0; }
complex(double r, double i = 0.0)  { re=r; im=i; }
```

If you declare a complex variable without parameters, the first constructor is used and the variable is initialized so that both parts are `0`. For example, complex aComp; creates a complex variable whose real and imaginary parts are both `0`.

If you give parameters, you can give one or two parameters. In either case, the second constructor is used. When you give only one parameter, it is taken as the magnitude for the real part and the imaginary part is set to `0`. For example, complex aComp(4.533); creates a complex variable with the value 4.533 + 0i.

If you give two values, the first is taken as the magnitude of the real part and the second as the magnitude of the imaginary part. For example, complex aComp(8.999, 2.333); creates a complex variable with the value 8.999 + 2.333i.

You can also create a complex number using the `polar` function. The `polar` function creates a complex value given a pair of polar coordinates (magnitude and angle).

There is no special destructor for type `complex`.

Exceptions for `cosh`:

C_COSH_RE
The real part was too large. A value with the correct angle and a huge magnitude was returned.

C_COSH_IM
The imaginary part was too large. A value with real and imaginary part of `0` returned.

Exceptions for `exp`:

C_EXP_RE_POS
The imaginary part was too small. A value with the correct angle and a huge magnitude was returned.

C_EXP_RE_NEG
The real part was too small. A value with real and imaginary part of `0` returned.

Exceptions for `log`:

C_LOG_0
The real and imaginary parts were both `0`. The same value returned.

Exceptions for `sinh:`

C_SINH_RE
The real part was too large. A value with the correct angle and a huge magnitude was returned.

C_SINH_IM
The imaginary part was too large. A value with real and imaginary part of `0` returned.

7.3 Mathematical Functions

The complex library provides 15 mathematical functions. Five are peculiar to complex numbers; the rest are complex number versions of functions in the standard C mathematical library.

`polar`
Takes a pair of polar coordinates that represent the magnitude and angle of a complex number and returns a complex number with the given magnitude and angle.

`pow`
This function takes two arguments. In the following example, it raises `a` to the power of `b`:

```
pow(a, b)
```

For example, to calculate (1-i)**4, enter:

```
pow(complex(1,-1),4)
```

This will produce the value (-4,0).

`real`
Returns the real part of a complex number.

`sin`
Returns the sine of its argument.

`sinh`
Returns the hyperbolic sine of its argument.

`sqrt`
Returns the square root of its argument.

7.4 Input and Output

The complex library provides extractors and inserters for complex numbers. (See *The C++ Programming Language* for basic information on extractors and inserters.)

For input, the complex extractor >> extracts a pair of numbers surrounded by parentheses and separated by a comma from the input stream, and reads them into a complex object. The first number is taken as the magnitude of the real part; the second as the magnitude of the imaginary part.

8.2 C++ Constructs

Naming Conventions

In general, variable and function names should consist of lowercase letters; exceptions are listed in the following subsections. Multiword names should use underscore(_) to separate the words. For example:

`draw_circle(), get_value(), maximum_rectangle_width`

Manifest Constants

Manifest constant names defined with `#define` should be all uppercase letters. For example:

```
#define MAX_HEIGHT 30
```

The use of `#define` constants is discouraged; use `const` or `enum` instead.

User-Defined Types

In C++, the `class`, `struct`, `union`, and `enum` tags are type names, and can be used in the same way as `typedef` names in C. Therefore, they should follow the naming conventions for all user-defined types. The first letter of `class`, `struct`, `union`, and `enum` names should be capitalized; all other letters should be lowercase or underscore. You might also capitalize the first letter or append `_t` (underscore `t`) to any types you create through `typedef`.

```
class rodent { };        // not recommended
typedef int rat;         // not recommended

class Rodent { };        // better
typedef int Rat;         // better
typedef int Rat_t;       // better
```

Use of `const`

A `const` object is one that is not allowed to change. A variable that is never known to be modified after initialization should be declared a `const`. Not only does this help other programmers, but it also allows the compiler to perform some optimizations that might not otherwise be possible. A `const` object should always be used in preference to a `#define` manifest constant or literal.

A `const` formal parameter of a function means that the function does not change the parameter. Formal parameters should be declared `const` if they are pointers or references and they are not changed in the call. Failure to do this will prevent other functions from calling that function with a `const` parameter. All class member functions that do not change the class object (`*this`) should also be declared `const`. For example:

```
class Foo {
public:
     get_val() const;
};
```

Only `const` member functions may be called for `const` class objects.

Use of `enum`

When there are a set of related constants, they should be defined as an enumeration, rather than as separate constants. For example:

```
// not recommended:
const int color_red = 0;
const int color_green = 1;
const int color_blue = 2;

// better:
enum Color { red, green, blue };
```

The `enums` are full-fledged types in C++ and their use allows the compiler to do stronger type checking.

The exceptions to these rules are constructors and destructors. Constructors cannot be virtual. Destructors should be virtual if there is a virtual function member, but cannot be pure virtual. See "Constructors and Destructors" on page 161.

Structures versus Classes

The keywords `struct` and `class` are interchangeable except for different default member access types (public for `struct` and private for `class`). To avoid confusion, `struct` should be used only when the structure would qualify as a valid C structure; that is, when there are no private or protected members and no member functions (including constructors and destructors). The `class` keyword should be used in all other cases.

Class Declarations

A C++ class declaration has several elements: public members, protected members, private members, and friend declarations. The complete declaration of a major class can be very large and complex. A consistent layout style for the class elements increases readability and allows you to quickly find any particular element. A consistent style also allows construction of a simple class pretty-printer that outputs class declarations (with only the public members visible) for other programmers who might use the class.

There are two ways you might declare the elements of a class. This is the first way:

public members

protected members

private members

friend declarations

Type declarations

Data members

Constructors (default constructor first, followed by other constructors)

Destructors

Overloaded operators

Other member functions

The colon separating a class name from its derived class name should have a space on both sides of it. The labels public, private, and protected within the class body should start in the same column as the class keyword so as to stand out from the member declarations. An alternate style is to indent them four spaces.

Defining Member Functions

As a general rule, member functions should be defined outside the class body. There are several reasons for this.

- It increases the separation between the class interface and its implementation by hiding the implementation of the class methods from users of the class.
- It makes it easier to change between in-line and non in-line versions of the functions since member functions defined outside the class body are not automatically in-lined.
- It makes the class definition less cluttered and easier to read.

Occasionally you might want to define very simple member functions inside a class body. This is acceptable under the following circumstances:

- If the member function is very simple, consisting of not more than a few lines of source code. (This is the same as one of the criteria used to determine whether or not to make a function an in-line function.)
- If the member function is intended to always be an in-line function.

Further, inlining increases code size, causing larger programs. This, in turn, causes the system to do more paging operations, which could easily overwhelm any saved call/return overhead. Hence, functions whose body contains more than a few lines of code should not be in-lined (and should also not be defined inside a class body).

Inlining also makes debugging more difficult because in-lined functions lose their identity. One way to resolve this is to use the cfront compiler flag `+d` during development. The `+d` flag keeps all functions from being in-lined, even those which have been explicitly declared as in-line. For the production compile, remove that flag. The `inline` keyword is only a recommendation to the compiler; the compiler may or may not in-line a function depending on the function's complexity. In C++ 3.0, the `-g` flag will automatically turn on the `+d` flag.

Using Overloaded Functions

Be careful if you use overloaded functions with similar arguments that might be converted, particularly if they are used in several files. Put the declarations in one header file. For example,

```
sts *alloc_and_init (int size);
sts *alloc_and_init (enum foobar info);
```

One of your files could have a bug that the compiler cannot report if one of the declarations is missing.

```
sts *alloc_and_init (int size);

enum foobar a;
c_structure *foo;

foo = alloc_and_init (a); // BUG: 'a' converted to int
```

Using Operator Functions

The use of operator functions can potentially make code difficult to understand. Their infix use may disguise the fact that an expression is actually a function call on class-type operands.

Constructor Initialization Lists

Initialization is distinct from assignment, both conceptually and semantically. Class constructors are called in the context of initialization; so the members should be initialized in the initialization list for the constructor rather than assigned in the body of the constructor.

```
class Complex {
     double r, i;
     Complex() { r = 0.0; i = 0.0; }// inferior
     Complex(double x, double y) { r = x; i = y; }// inferior
};
class Complex {
     double r, i;
     Complex(): r(0.0), i(0.0) { }// preferred
     Complex(double x, double y): r(x), i(y) { }// preferred
};
```

Declarations inside a `for` *Initializing Statement*

C++ allows variables to be declared inside the initializing statement of a `for` loop. The scope of such variables is not the `for` loop, as you might expect from its lexical location, but extends to the end of the block enclosing the `for` loop. This can cause considerable confusion. For example, the following code fragment does not terminate.

```
for (int i=0; i<10; ) {
     for (int i=0; i<10; i++) {
          // ...
     }
     i++;      // CAUTION: refers to inner i!
}
```

Using Virtual Destructors

As a rule, if a class has virtual functions, give it a virtual destructor. Look at the following example. The destructor for the derived class will not be called. Memory will probably be left undeleted.

```
class Base {
     // ...
     Base();
     ~Base();
};
class Derived : public Base {
     // ...
     Obj();
     ~Obj();
};
void f()
{
     Base * b = new Derived; // create a Derived object
     delete *b; // delete a Base object, not a Derived object
}
```

Using `bzero()`

It is a common practice in C to zero out C structures by using `bzero()`. This should not be done if those structures are converted to C++ objects, since you might destroy the virtual pointer within the object. The correct method is to zero out each field of the object explicitly.

The guard name is created by turning the file name to all uppercase characters, preceding it with an underscore, and replacing all nonalphanumeric characters with an underscore. The last line in the file is the `#endif` as follows:

```
#endif /* _FOO_H */
```

`#ident` *String*

After the guard, there should an `#ident` line that contains a standard SCCS ID string. The SCCS ID should contain the magic string `@(#)`, the module name, the SCCS version number, the date of last modification, and a company designator. SCCS will automatically insert this information if the following string is used:

```
#ident  "%Z%c++style.mif  2.190/11/05 XYZ"
```

A tab character appears between `#ident` and the double quote.

Block Comment

Next is a block comment describing the purpose of the header file and the objects contained within. The description should be concise.

`#include`

Any include files come next. All additional header files that are required by the `#include` file should be included, so that you or another programmer do not have to do this. Double quotes around the included file name allow for greater flexibility at compilation time. For example:

```
#include "dir/foo.h"
```

Absolute path names should be avoided in include statements.

`#define`

Any define statements that are global to the module but not specific to a member of a class or structure come next.

Try to avoid define statements. For single-value macros, `enum` or `const` variables should be used instead. For parameterized macros, in-line functions should be used instead. Your program will be easier to debug and understand.

Here are some ways to work around the problem:

- Write smaller header files.

  Usually, one class per file is a good rule to follow. If several closely related small classes are often used together, you may want to put the class declarations into the same header file. This will reduce the amount of time acpp needs to open/close multiple header files.

- Write organized `.cc` files.

  Do not put unrelated functions into the same file. Group logically related functions together to reduce the number of unrelated header files that need to be included into the same `.cc` file.

- Enclose header files with `#ifndef`, `#define` or `#endifs` directives.

  Always enclose header files with `#ifndef`, `#define`, or `#endif` acpp directives. In large C++ applications, it will prevent you from the accidental multiple inclusion of header files.

- Include (or nest) header files in other header files only when absolutely necessary.

  Including unnecessary header files in other header files instead of `.cc` files is usually the main cause of large include problems. Many of the header files are actually more appropriate in the `.cc` file than the header file.

  The procedure for reducing the number of header file includes statements is simple. Look at the include statements in each header file. For each `#include`, see if the types defined in that file are referenced in the header file.

In the following examples the `#include` statement should be in the header file.

- `Base.h` defines a base class of one of the classes in a header file. Therefore, it should be included in that header file. For example:

```
#include "Base.h"

class Obj : public Base {

          ...
};
```

8.5 Source File Structure

The format of a source file is similar to that of a header file.

```
// Copyright (c) 1990 by XYZ, Inc.

#ident "@(#)foo.h 1.1 90/04/23 XYZ"

// Functions to manipulate object foo

{#includes}
{#defines}
{typedefs}
{class declarations}
{global variables definitions}
{local function declarations}
{local function definitions}
{class function definitions}
```

Copyright notice

The first line of the file should be a comment containing a copyright notice.

```
// Copyright (c) 1990 by XYZ, Inc.
```

The `#ident` String

As with header files, source files contain a `#ident` string containing a standard SCCS identification string.

Block Comment

Next, a block comment describes the purpose of the source file and the functions contained within. The description should be concise.

The `#include` File

Any include files come next. Only those header files needed directly by the source file should be included. It is assumed that those header files include any additional header files needed indirectly.

8.6 Additional Miscellaneous Guidelines

Avoid declaring anything global. Globals are problematic, especially in libraries. The problems with global variables include:

- They pollute the global name space, increasing the potential for name conflicts.
- They are directly referenced by functions, causing those functions to have side effects and making the program harder to understand.
- A global variable contains state for the program, that is, the operations depending on that global cannot be parameterized.
- They cause trouble for multithreaded applications, since each thread does not have its own copy of the data.

The number of global variables or functions in a program can be reduced by a more object-oriented programming approach. With this approach, the program state is contained within the objects rather than being global to all objects. If global variables or functions still remain, consider defining them as static class members. Although static class members also have the same problems as the last two items listed above for globals, their access is more controlled and their names are scoped within the class.

Avoid using unions, because union types defeat strong type checking and can be difficult to inspect in a debugger.

9.2 How to Use this Chapter

1. **Study the above sample and Section 9.3, "Getting It Right," on page 173.**
2. **Turn to Section 9.5, "FORTRAN Calls C++," on page 180 or Section 9.6, "C++ Calls FORTRAN," on page 202.**
3. **Within any of the two sections mentioned in step 2, choose one of these subsections:**
   - Arguments Passed by Reference
   - Arguments Passed by Value
   - Function Return Values
   - Labeled Common
   - Sharing I/O
   - Alternate Returns
4. **Within any of the subsections mentioned in step 3, choose one of these examples:**

   For the arguments, there is an example for each of these:

   - Simple types (character*1, logical, integer, real, double precision)
   - Complex types (complex, double complex)
   - Character strings (character*n)
   - One-dimensional arrays (integer a(9))
   - Two-dimensional arrays (integer a(4,4))
   - Structured records (structure and record)
   - Pointers

   For the function return values, there is an example for each of these:

   - Integer (int)
   - Real (float)
   - Pointer to real (pointer to float)
   - Double precision (double)
   - Complex
   - Character string

   For each of labeled common, sharing I/O, and alternate returns, there is one set of examples.

If any one of these cannot be done, a statement says so.

Data Type Compatibility

Default data type sizes and alignments (that is, without `-f`, `-i2`, `-misalign`, `-r4`, or `-r8`) are shown in Table 9-1.

The `REAL*16` and the `COMPLEX*32` can be passed between FORTRAN and C++, but not between FORTRAN and some previous versions of C++.

Note: C++ does not support long double

Table 9-1 Argument Sizes and Alignments — Pass by Reference (No Options)

FORTRAN Type	**C++ Type**	**Size (bytes)**	**Alignment (bytes)**
byte x	`char x`	1	1
character x	`char x`	1	1
character**n* x	`char x[n]`	*n*	1
complex x	`struct {float r,i;} x;`	8	4
complex*8 x	`struct {float r,i;} x;`	8	4
double complex x	`struct {double dr,di;}x;`	16	4
complex*16 x	`struct {double dr,di;}x;`	16	4
double precision x	`double x`	8	4
real x	`float x`	4	4
real*4 x	`float x`	4	4
real*8 x	`double x`	8	4
integer x	`int x`	4	4
integer*2 x	`short x`	2	2
integer*4 x	`int x`	4	4
logical x	`int x`	4	4
logical*4 x	`int x`	4	4
logical*2 x	`short x`	2	2
logical*1 x	`char x`	1	1

Remarks

- Alignments are for FORTRAN types.
- Arrays pass by reference, if the elements are compatible.

Underscore in Names of Routines

The FORTRAN compiler normally appends an underscore (_) to the names of subprograms, for both a subprogram and a call to a subprogram. This distinguishes it from C++ procedures or external variables with the same user-assigned name. If the name has exactly 32 characters, the underscore is not appended. All FORTRAN library procedure names have double leading underscores to reduce clashes with user-assigned subroutine names.

Two common solutions to the underscore problem are:

- In the C++ function, change the name of the function by appending an underscore to that name.
- Use the `C()` pragma to tell the FORTRAN compiler to omit those trailing underscores.

Use one or the other, but not both.

Most of the examples in this chapter use the FORTRAN `C()` compiler pragma and do not use the underscores.

The `C()` pragma directive takes the names of external functions as arguments. It specifies that these functions are written in the C (or C++) language, so the FORTRAN compiler does not append an underscore to such names, as it ordinarily does with external names. The `C()` directive for a particular function must appear before the first reference to that function. It must appear in each subprogram that contains such a reference. The conventional usage is this:

```
      EXTERNAL ABC, XYZ!$PRAGMA C( ABC, XYZ )
```

If you use this pragma, then in the C++ function you must not append an underscore to those names.

9.4 C++ Name Encoding

To implement function overloading and type-safe linkage, the C++ compiler normally appends `type` information to the names of functions. To prevent the C++ compiler from appending `type` information to the names of functions,

Array Order

FORTRAN arrays are stored in column-major order, C++ arrays in row-major order. For one-dimensional arrays, this is no problem. This is only a minor problem for two-dimensional arrays as long as the array is square. Sometimes it is enough to just switch subscripts.

For two-dimensional arrays that are not square, it is not enough to just switch subscripts. For arrays of more than two dimensions, this is usually considered too much of a problem.

Libraries and Linking with the `f77` Command

To get the proper FORTRAN libraries linked, use the `f77` command to pass the `.o` files on to the linker. This usually shows up as a problem only if a C++ main calls FORTRAN. Dynamic linking is encouraged and made easy.

Example 1: Use `f77` to link.

```
demo%  f77 -c -silent RetCmplx.f
demo% CC -c RetCmplxmain.cc
demo% f77 RetCmplx.o RetCmplxmain.o ← This does the linking.
demo% a.out
 4.0 4.5
 8.0 9.0
demo% ■
```

Example 2: Use `CC` to link. This fails. The libraries are not linked.

```
demo% f77 -c RetCmplx.f
RetCmplx.f:
     retcmplx:
demo% CC RetCmplx.o RetCmplxmain.cc   ←  wrong link command
ld: Undefined symbol                     ←  missing routine
 __Fc_mult
demo% ■
```

File Descriptors and `stdio`

FORTRAN I/O channels are in terms of unit numbers. The I/O system does not deal with unit numbers, but with *file descriptors*. The FORTRAN runtime system translates from one to the other, so most FORTRAN programs don't have to know about file descriptors. Many C++ programs use a set of

This occurs transparently and should be of concern only if you try to perform a `READ`, `WRITE`, or `ENDFILE` but you don't have permission. Magnetic tape operations are an exception to this general freedom, since you could have write permissions on a file but not have a write ring on the tape.

9.5 FORTRAN Calls C++

The following sections discuss the FORTRAN calls within C++

Arguments Passed by Reference (f77 Calls C++)

Simple Types Passed by Reference (f77 Calls C++)

For simple types, define each C++ argument as a reference.

`SimRef.cc`

```
extern "C" void simref (
     char& t,
     char& f,
     char& c,
     int& i,
     float& r,
     double& d,
     short& si )
{
     t = 1;
     f = 0;
     c = 'z';
     i = 9;
     r = 9.9;
     d = 9.9;
     si = 9;
}
```

Compile and execute, with output.

```
demo% CC -c CmplxRef.cc
demo% f77 -silent CmplxRef.o CmplxRefmain.f
demo% a.out
  ( 6.00000,    7.00000)
  ( 8.0000000000000,    9.0000000000000)
demo% ■
```

A C++ reference to a `float` matches a `REAL` passed by reference.

Character Strings Passed by Reference (`f77` Calls C++)

Passing strings between C++ and FORTRAN is not encouraged.

For every FORTRAN argument of character type, FORTRAN associates an extra argument, giving the length of the string. The string lengths are equivalent to C++ `long int` quantities passed by value. This differs from standard C++ use where all C++ strings are passed by reference. The order of arguments is as follows:

1. Address for each argument (datum or function)
2. The length of each character argument, as a `long int`.

The whole list of string lengths comes after the whole list of other arguments.

The FORTRAN call in:

```
      CHARACTER*7 S
      INTEGER B(3)
(arguments)
      CALL SAM( B(2), S )
```

is equivalent to the C++ call in:

```
      char s[7];
      long int b[3];
(arguments)
      sam_( &b[1], s, 7L );
```

Using the Extra Arguments

You can use the extra arguments. In the following example, all this C++ function does with the lengths is print them; what you really do with them is up to you.

`StrRef2.cc`

```
#include <string.h>
#include <stdio.h>

extern "C" void strref ( char (&s10)[], char (&s26)[], int L10,
int L26 ) {
     static char ax[11] = "abcdefghij";
     static char sx[27] = "abcdefghijklmnopqrstuvwxyz";
     printf( "%d %d\n", L10, L26 );
     strncpy( s10, ax, 10 );
     strncpy( s26, sx, 26 );
}
```

If you compile `StrRef2.c` and `StrRefmain.f`, then `you` `get` this output.

```
10 26
s10='abcdefghij'
s26='abcdefghijklmnopqrstuvwxyz'
```

One-Dimensional Arrays Passed by Reference (`f77` *Calls C++)*

A C++ array, indexed from 0 to 8:

`FixVec.cc`

```
extern "C" void fixvec ( int V[9], int& Sum )
{
     Sum= 0;
     for( int i= 0; i < 9; ++i ) {
         Sum += V[i];
     }
}
```

A FORTRAN array, implicitly indexed from 1 to 9:

`FixVecmain.f`

```
      integer i, Sum
      integer a(9) / 1,2,3,4,5,6,7,8,9 /
      external FixVec !$pragma C( FixVec )
      call FixVec( a, Sum )
      write( *, '(9I2, " ->" I3)') (a(i),i=1,9), Sum
      end
```

A 2 by 2 FORTRAN array, explicitly indexed from 0 to 1, and 0 to 1:

`FixMatmain.f`

```
      integer c, m(0:1,0:1) / 00, 10, 01, 11 /, r
      external FixMat !$pragma C( FixMat )
      do r= 0, 1
         do c= 0, 1
           write( *, '("m(",I1,",",I1,")=",I2.2)')  r, c, m(r,c)
         end do
      end do

      call FixMat( m )
      write( *, * )

      do r= 0, 1
         do c= 0, 1
           write( *, '("m(",I1,",",I1,")=",I2.2)')  r, c, m(r,c)
         end do
      end do

      end
```

Compile and execute. Show `m` before and after the C call.

Compare `a[0][1]` with `m(1,0)`: C++ changed `a[0][1]`, which is FORTRAN `m(1,0)`.

```
demo% CC -c FixMat.cc
demo% f77 -silent FixMat.o FixMatmain.f
demo% a.out
m(0,0) = 00
m(0,1) = 01
m(1,0) = 10
m(1,1) = 11

m(0,0) = 00
m(0,1) = 01
m(1,0) = 99
m(1,1) = 11
demo% ■
```

Pointers Passed by Reference (`f77` *Calls C++)*

C++ gets it as a reference to a pointer.

`PassPtr.cc`

```
extern "C" void passptr ( int* & i, double* & d )
{
     *i = 9;
     *d = 9.9;
}
```

FORTRAN passes by reference, and it is passing a pointer.

`PassPtrmain.f`

```
       program PassPtrmain
       integer i
       double precision d
       pointer (iPtr, i), (dPtr, d)
       external PassPtr !$pragma C ( PassPtr )
       iPtr = malloc( 4 )
       dPtr = malloc( 8 )
       i = 0
       d = 0.0
       call PassPtr( iPtr, dPtr )
       write( *, "(i2, f4.1)" ) i, d
       end
```

Compile and execute, with output:

```
demo% CC -c PassPtr.cc
demo% f77 -silent PassPtr.o PassPtrmain.f
demo% a.out
 9 9.9
demo% ■
```

Arguments Passed by Value (`f77` *Calls C++)*

In the call, enclose an argument in the nonstandard function `%VAL()`.

Compile and execute, with output.

```
demo% CC -c SimVal.cc
demo% f77 -silent SimVal.o SimValmain.f
demo% a.out
args=111111(If nth digit=1, arg n OK)
demo% ■
```

Real Variables Passed by Value (`f77` *Calls C++*)

In some previous versions of C++, if C++ passed an argument of type `float` by value, C++ promoted it to a `double`. To avoid this, the macros `FLOATP...` and `FLOATP... VALUE` were used. Using `FLOATP...` and `FLOATP... VALUE` is no longer necessary. Compare this example with the first one in this chapter, `Samp.cc` and `Sampmain.f.` In `Sampmain.f`, FORTRAN passes an integer and a real by reference to C++; then C++ uses them as references to an integer and to a real.

`FloatVal.cc`

```
#include <math.h>

extern "C" void floatval ( FLOATPARAMETER f, double& d ) {
    float x;
    x = FLOATPARAMETERVALUE( f );
    d = double(x) + 1.0 ;
}
```

`FloatValmain.f`

```
      double precision d
      real r / 8.0 /
      external FloatVal !$pragma C( FloatVal )
      call FloatVal( %VAL(r), d )
      write( *, * ) r, d
      end
```

Compile and execute, with output.

```
demo% CC -c FloatVal.cc
demo% f77 -silent FloatVal.o FloatValmain.f
demo% a.out
    8.00000 9.0000000000000
demo% ■
```

Pointers Passed by Value (`f77` Calls C++)

C++ gets it as a pointer.

`PassPtrVal.cc`

```
extern "C" void passptrval ( int* i, double* d )
{
     *i = 9;
     *d = 9.9;
}
```

FORTRAN passes a pointer by value:

`PassPtrValmain.f`

```
      program PassPtrValmain
      integer i
      double precision d
      pointer (iPtr, i), (dPtr, d)
      external PassPtrVal !$pragma C ( PassPtrVal )
      iPtr = malloc( 4 )
      dPtr = malloc( 8 )
      i = 0
      d = 0.0
      call PassPtrVal( %VAL(iPtr), %VAL(dPtr) ) ! Nonstandard?
      write( *, "(i2, f4.1)" ) i, d
      end
```

Compile and execute, with output:

```
demo% CC -c PassPtrVal.cc
demo% f77 -silent PassPtrVal.o PassPtrValmain.f
demo% a.out
 9 9.9
demo% ■
```

Function Return Values (`f77` Calls C++)

For function return values, a FORTRAN function of type `BYTE`, `INTEGER`, REAL, `LOGICAL`, or `DOUBLE PRECISION` is equivalent to a C++ function that returns the corresponding type. There are two extra arguments for the return values of character functions and one extra argument for the return values of complex functions.

Return a `float` *(*`f77` *Calls C++)*

RetFloat.cc

```
extern "C" float retfloat ( float& pf )
{
    float f;
    f = pf;
    ++f;
    return f;
}
```

RetFloatmain.f

```
      real RetFloat, r, s
      external RetFloat !$pragma C( RetFloat )
      r = 8.0
      s = RetFloat( r )
      print *, r, s
      end
```

```
demo% CC -c RetFloat.cc
demo% f77 -silent RetFloat.o RetFloatmain.f
demo% a.out
    8.00000 9.00000
demo% ■
```

In earlier versions of C++, if C++ returned a function value that was a `float`, C++ promoted it to a `double`, and various tricks were needed to get around that.

Return a Pointer to a `float` *(*`f77` *Calls C++)*

This example shows how to return a function value that is a pointer to a `float`. Compare with previous example.

RetPtrF.cc

```
static float f;

extern "C" float* retptrf ( float& a )
{
    f = a;
    ++f;
    return &f;
}
```

Compile and execute, with output.

```
demo% CC -c RetDbl.cc
demo% f77 -silent RetDbl.o RetDblmain.f
demo% a.out
   8.0 9.0
demo% ■
```

Return a Complex (`f77` Calls C++)

A COMPLEX or DOUBLE COMPLEX function is equivalent to a C++ routine having an additional initial argument that points to the return value storage location. A general pattern for such a FORTRAN function is

```
      COMPLEX FUNCTION F ( arguments )
```

The pattern for a corresponding C++ function is

```
struct complex { float r, i; };
f_ ( complex temp, arguments );
```

Example — C++ returns a type COMPLEX function value to FORTRAN:

RetCmplx.cc

```
struct complex { float r, i; };

extern "C" void retcmplx ( complex& RetVal, complex& w ) {
    RetVal.r = w.r + 1.0 ;
    RetVal.i = w.i + 1.0 ;
    return;
}
```

RetCmplxmain.f

```
      complex u, v, RetCmplx
      external RetCmplx !$pragma C( RetCmplx )
      u = ( 7.0, 8.0 )
      v = RetCmplx( u )
      write( *, * ) u
      write( *, * ) v
      end
```

- The returned string is passed by the extra arguments `retval_ptr` and `retval_len`, a pointer to the start of the string and the string's length.
- The character-string argument is passed with `ch_ptr` and `ch_len`.
- The `ch_len` is at the end of the argument list.
- The repeat factor is passed as `n_ptr`.

In FORTRAN, use the above C++ function as follows:

`RetStrmain.f`

```
      character String*100, RetStr*50
      String = RetStr( '*', 10 )
      print *, "'", String(1:10), "'"
      end
```

```
demo% CC -c RetStr.cc
demo% f77 -silent RetStr.o RetStrmain.f
'**********'
demo% ■
```

Labeled Common

C++ and FORTRAN can share values in labeled common. The method is the same no matter which language calls which.

UseCom.f

```
      subroutine UseCom ( n )
      integer n
      real u, v, w
      common / ilk / u, v, w
      n = 3
      u = 7.0
      v = 8.0
      w = 9.0
      return
      end
```

If a FORTRAN main program calls C++, then before the FORTRAN program starts, the FORTRAN I/O library is initialized to connect units 0, 5, and 6 to `stderr`, `stdin`, and `stdout`, respectively. The C++ function must take the FORTRAN I/O environment into consideration to perform I/O on open file descriptors.

Mixing with `stdout` *(*`f77` *Calls C++)*

A C++ function that writes to `stderr` and to `stdout`:

MixIO.cc

```
#include <stdio.h>

extern "C" void mixio ( int& n ) {
    if( n <= 0 ) {
     fprintf( stderr, "Error:  negative line number (%d)\n", n );
     n= 1;
    }
    printf( "In   C++:    line # = %2d\n", n );
}
```

In FORTRAN, use the above C++ function as follows:

`MixIOmain.f`

```
      integer n/ -9 /
      external MixIO !$pragma C( MixIO )
      do i= 1, 6
         n = n +1
         if ( abs(mod(n,2)) .eq. 1 ) then
            call MixIO( n )
         else
            write( *, '("In Fortran:  line # = ", i2)' ) n
         end if
      end do
      end
```

Alternate Returns (`f77` *Calls C++) - N/A*

C++ does not have an alternate return. The work-around is to pass an argument and branch on that.

9.6 C++ Calls FORTRAN

Arguments Passed by Reference (C++ Calls f77)

Simple Types Passed by Reference (C++ Calls f77)

Here, FORTRAN expects all these arguments to be passed by reference (default).

`SimRef.f`

```
      subroutine SimRef ( t, f, c, i, d, si, sr )
      logical*1 t, f
      character c
      integer i
      double precision d
      integer*2 si
      real sr
      t = .true.
      f = .false.
      c = 'z'
      i = 9
      d = 9.9
      si = 9
      sr = 9.9
      return
      end
```

Complex Types Passed by Reference (C++ Calls `f77`*)*

The complex types require a simple structure.

CmplxRef.f

```
      subroutine CmplxRef ( w, z )
      complex w
      double complex z
      w = ( 6, 7 )
      z = ( 8, 9 )
      return
      end
```

In the previous example, w and z are passed by reference (default).

CmplxRefmain.cc

```
#include <stdlib.h>

struct complex { float r, i; };
struct dcomplex { double r, i; };

extern "C" void cmplxref_ ( complex& w, dcomplex& z );

main ( ) {
    complex d1;
    dcomplex d2;

    cmplxref_( d1, d2 );
   printf( "%3.1f %3.1f\n%3.1f %3.1f\n", d1.r, d1.i, d2.r, d2.i
);
    exit(0);
}
```

In the previous example, w and z are references.

Compile and execute, with output.

```
demo% f77 -c -silent CmplxRef.f
demo% CC -c CmplxRefmain.cc
demo% f77 CmplxRef.o CmplxRefmain.o
demo% a.out
6.0 7.0
8.0 9.0
demo% █
```

Arguments Passed by Value (C++ Calls `f77`) - N/A

FORTRAN can call C++, and pass an argument by value. But FORTRAN cannot handle an argument passed by value if C++ calls FORTRAN. The workaround is to pass all arguments by reference.

Function Return Values (C++ Calls `f77`)

For function return values, a FORTRAN function of type `BYTE`, `INTEGER`, `LOGICAL`, or `DOUBLE PRECISION` is equivalent to a C++ function that returns the corresponding type. There are two extra arguments for the return values of character functions and one extra argument for the return values of complex functions.

Return an `int` (C++ Calls `f77`)

Example: FORTRAN returns an `INTEGER` function value to C++.

`RetInt.f`

```
      integer function RetInt ( k )
      integer k
      RetInt = k + 1
      return
      end
```

`RetIntmain.cc`

```
#include <stdlib.h>

extern "C" int retint_ ( int& );

main ( ) {
      int k = 8;
      int m = retint_( k );
      printf( "%d %d\n", k, m );
      exit(0);
}
```

Return a `double` *(C++ Calls* `f77`*)*

Example: FORTRAN returns a `DOUBLE PRECISION` function value to C++.

`RetDbl.f`

```
      double precision function RetDbl ( x )
      double precision x
      RetDbl = x + 1.0
      return
      end
```

`RetDblmain.cc`

```
#include <stdlib.h>

extern "C" double retdbl_ ( double& );

main ( ) {
      double x = 8.0;
      double y = retdbl_( x );
      printf( "%8.6f %8.6f\n", x, y );
      exit(0);
}
```

Compile and execute, with output.

```
demo% f77 -c -silent RetDbl.f
demo% CC -c RetDblmain.cc
demo% f77 RetDbl.o RetDblmain.o
demo% a.out
8.000000 9.000000
demo% ■
```

Return a `COMPLEX` *(C++ Calls* `f77`*)*

A `COMPLEX` or `DOUBLE COMPLEX` function is equivalent to a C++ routine having an additional initial argument that points to the return value storage location. A general pattern for such a FORTRAN function is

```
     COMPLEX FUNCTION F ( arguments )
```

The pattern for a corresponding C++ function is

```
struct complex { float r, i; };
void f_ ( complex &, other arguments )
```

A FORTRAN string function has two extra initial arguments — data address and length. If you have a FORTRAN function of the following form, with no `C++()` pragma,

```
      CHARACTER*15 FUNCTION G ( arguments )
```

and a C++ function of this form

```
g_ ( char * result, long int length, other arguments )
```

they are equivalent, and can be invoked in C++ with

```
char chars[15];
g_ ( chars, 15L, arguments );
```

The lengths are passed by value. You must provide the null terminator.

`RetChr.f`

```
      function RetChr( c, n )
      character RetChr*(*), c
      RetChr = ''
      do i = 1, n
         RetChr(i:i) = c
      end do

      RetChr(n+1:n+1) = char(0) ! Put in the null terminator.
      return
      end
```

The method is the same no matter which language calls which.

UseCom.f

```
      subroutine UseCom ( n )
      integer n
      real u, v, w
      common /ilk/ u, v, w
      n = 3
      u = 7.0
      v = 8.0
      w = 9.0
      return
      end
```

UseCommain.cc

```
#include <stdio.h>

struct ilk_type {
    float p;
    float q;
    float r;
};

extern ilk_type ilk_ ;
extern "C" void usecom_ ( int& );

main ( ) {
    char *string = "abc0" ;
    int count = 3;
    ilk_.p = 1.0;
    ilk_.q = 2.0;
    ilk_.r = 3.0;
    usecom_( count );
    printf( " ilk_.p=%4.1f,  ilk_.q=%4.1f,  ilk_.r=%4.1f\n",
        ilk_.p, ilk_.q, ilk_.r );
    exit(0);
}
```

Example: Sharing I/O using a C++ main and a FORTRAN subroutine.

MixIO.f

```
      subroutine MixIO ( n )
      integer n
      if ( n .le. 0 ) then
         write(0,*) "error: negative line #"
         n = 1
      end if

      write( *, '("In Fortran:  line # = ", i2 )' ) n
      end
```

MixIOmain.cc

```
#include <stdio.h>

extern "C" {
    void mixio_( int& );
    void f_init();
    void f_exit();
};

main ( ) {
    f_init();
    int m= -9;

    for( int i= 0; i < 5; ++i ) {
     ++m;
     if( m == 2  ||  m == 4 ) {
         printf( "In  C++  :  line # = %d\n", m );
     } else {
         mixio_( m );
     }
    }
    f_exit();
}
```

C++ invokes the subroutine as a function.

AltRetmain.cc

```
#include <stdlib.h>

extern "C" int altret_ ( int& );

main ( ) {
    int k = 0;
    int m = altret_( k );
    printf( "%d %d\n", k, m );
    exit(0);
}
```

Compile, link, and execute:

```
demo% f772.0 -c AltRet.f
AltRet.f:
        altret:
demo% aCC -c AltRetmain.cc
demo% f772.0 AltRet.o AltRetmain.o
demo% a.out
k:
20
9 2
demo%
```

In this example, the C++ main receives a 2 as the return value of the subroutine, because the user typed in a 20.

You can also use this utility to get an index. For example:

```
tutorial% ctags -v testr.cc str.cc | sort -f > index
tutorial% cat index
main testr.cc 1
operator<< str.cc 1
operator>> str.cc 1
string::insert str.cc 1
string::string str.cc 1
```

The number at the end of the line refers to the page number in the program listing. In this case all the numbers are 1 because the programs are short and all functions are defined on one page.

In the previous example, the output that `ctags` produces is suitable for input to the `vgrind` utility, which processes files for improved output appearance.

10.2 The `dem` *Utility*

When a C++ program has overloaded function names, the C++ translator alters the names of the functions to produce unique names. These altered names are called *mangled* names. The demangler utility, `dem`, takes C++ mangled names and produces the function prototypes that must have produced them. For example:

```
tutorial% dem _ _ct_ _6stringFRC6string
_ _ct_ _6stringFRC6string == string::string(const string&)
```

See the `dem` manual page for more information on the `dem` program.

`for static destructor.`

The demangled names for static constructors and destructors are printed in the following format:

For static constructors:

`static constructor function for <filename>`

For static destructors:

`static destructor function for <filename>`

For example, `___std___stream_in_c__` is demangled as:

`static destructor function for _ _stream_in_c .`

The file name is left in the mangled format because the `cfront` name demangling scheme does not preserve enough information to demangle it.

For C++ virtual table symbols, the mangled name takes the following format:

```
_ _vtbl_ _<class>
_ _vtbl_ _<rootclass>_ _<derived class>
```

In the `nm++` output, the demangled names for the virtual table symbols are printed as:

```
virtual table for <class>
virtual table for <derived class> derived from <rootclass>
```

For example, the demangled format of:

`_ _ _vtbl_ _7fstream`

is:

`virtual table for fstream`

and the demangled format of:

`_ _ _vtbl_ _3ios_ _18ostream_withassign`

is:

`virtual table for class ostream_withassign derived from ios`

Suppose that the final executable file is named `index`. Now you can run the `index` program as usual. When a program is profiled, the results appear in a file called `mon.out` at the end of the run. Every time you run the program, a new `mon.out` is created, overwriting the old version. You then use `prof++` to interpret the results of the profile as shown in the following example.

```
tutorial% index
tutorial% prof index
%time cumsecs #call ms/call name
  14.8    3.88  3918    0.99  write
                              [_write]
  11.5    6.90                count
                              [mcount]
   8.7    9.18   608    3.75  yyparse()
                              [_yyparse_ _Fv]
   5.6    10.66 24393   0.06  tlex()
                              [_tlex_ _Fv]
   4.9    11.94 22920   0.06  fputs
                              [_fputs]
   4.0    12.98 16454   0.06  table::look(char*,unsigned char)
                              [_look_ _5tableFPcUc]
   3.3    13.84 24393   0.04  deltok(int)
                              [_deltok_ _Fi]
   3.2    14.68 10770   0.08  expr::typ(table*)
                              [_typ_ _4exprFP5table]
   2.7    15.38 7939    0.09  type::check(type*,unsigned char)
                              [_check_ _4typeFP4typeUc]
   2.2    15.96 24392   0.02  lalex()
                              [_lalex_ _Fv]
   1.9    16.46 4382    0.11  _doprnt
                              [_ _doprnt]
   1.9    16.96 10147   0.05  lxget(int,int)
                              [_lxget_ _FiT1]
     ..............
```

In the output all C++ mangled names are decoded and their corresponding demangled names are also printed. See the `prof` manual page for an interpretation of the profiling results.

Note – Some of the following code sample was cut so it could fit on the page.

```
tutorial% index
tutorial% gprof++ index

                                 called/total      parents
index   %time   self  descendents called+self   name       index
                                 called/total      children
                                            <spontaneous>
[1]      97.3   0.00       204.63             tart [1]
                0.00       204.63    1/1          main [3]
                0.00         0.00    1/1          finitfp_ [204]
                0.00         0.00    1/1          on_exit [212]

-----------------------------------------------

               34.53       170.11    1/1          main [3]
[2]      97.3  34.53       170.11    1            Proc0() [2]
                                                 [_Proc0_ _Fv]
               25.22        31.77  500000/500000     Proc1(Record*) [4]
                                                    [_Proc1_ _FP6Record]
               12.58        30.37  500000/500000     Func2(char[31],char[31]) [5]
                                                    [_Func2_ _FA31_cT1]
               24.85         0.00  500000/500000     Proc8(int[51],int([51])
                                                    [_Proc8_ _FA51_iA51_
..............................
```

In the output, all C++ mangled names are decoded and their corresponding demangled names are also printed out. See the manual page for `gprof++` for an interpretation of the profiling results and the `call-graph`.

10.7 *The* `lex++` *Utility*

The `lex++` utility is based on operating system 4.1 `lex` with C++ enhancements so that its output can be compiled by both the C and C++ compilers.

```
{number}        { }
%%
```

Running `lex xyz.l` produces slightly different output than running SunOS 4.1.x `lex` on `xyz.l`.

10.8 The `yacc++` Utility (SunOS 4.1.x only)

The `yacc++` utility is based on operating system 4.1 `yacc`, with enhancements to permit successful compilation under C++. The `yacc++` utility is available with SunOS 4.1.x only. `yacc++` uses the parser prototype file `yaccpar` installed under the same directory where `yacc++` resides. If `yacc++` does not find `yaccpar` in that directory, it uses the copy of `yaccpar` in `/usr/lib`.

```
% yacc++ normal.y
% cat y.tab.c
#if defined (__cplusplus) || defined (c_plusplus)
#include <c_varieties.h>
#ifdef __EXTERN_C__
EXTERN_FUNCTION (extern int yylex, ());
#else
extern int yylex();
#endif
extern void yyerror(char*);entern int yyparse();
#endif
#include <malloc.h>
    .
    .
    .
```

The `yacc++` *Utility Limitations*

C++ 3.0.1 `yacc` and 2.1 `yacc++` (operating system 4.1-based) are different from 2.0 `yacc++` (operating system 4.0-based) in that they allow dynamic memory reallocation of `yacc`'s value and state stacks (expands in multiples of `YYMAXDEPTH`).

There is a side effect because of the dynamic memory reallocation: `yacc` does not support any class types that define their own assignment operator functions as `YYSTYPE.` This can be implemented, but it will involve considerable performance trade-offs in the resulting program and is not recommended.

Instead of using the actual class type itself as `YYSTYPE`, use a pointer to the class type as `YYSTYPE`. There is no problem with `yacc` 3.0 using this scheme, and the resulting memory allocation and reallocation is more efficient.

Since C++ 2.0, 2.1, and 3.0.1 do not allow any class types with constructors, destructors, or user-defined assignment functions to be member fields of a union, those classes cannot be member fields of `YYSTYPE`, if it is a union in `yacc++`. This can be circumvented by using pointers to class types instead.

You can also run `rpcgen` without the `-C` option and compile the resulting C files with `cc`. You can link the compiled object modules with the other C++ modules that you provide.

10.10 *The* `vgrind` *Utility (SunOS 4.1.x only)*

You can use `vgrind` with C++ programs. This utility is available on SunOS 4.1.x only. Use the `-l` *language* option to specify the language. For C++ the option is `-lc++`.

The `vgrind` utility uses `troff` to format the program sources named by the *filename* arguments. Comments are placed in *italics*, keywords in **boldface**; as each function is encountered, its name is listed on the page margin. See the manual page `vgrind(1)` for more information.

```
string operator+(string);
string operator=(string&);

friend ostream& operator<<(ostream&, string);
friend istream& operator>>(istream&, string&);
};

***************************** str.cc *****************************
// implementation for toy C++ strings package
// header file str.h

#include "str.h"

string::string(char *aStr)
{
  if (aStr == NULL)
    size = 0;
  else
    size = strlen(aStr);

  if (size == 0)
    data = NULL;
  else
    {
      data = new char[size+1];
      strcpy(data, aStr);
    }
}

void string::insert(char *ins)
{
  char *holder = new char [size + strlen(ins)+1];

strcpy(holder, data);
strcat(holder, ins);
  if (data)
     delete data;

  size = strlen(holder) - 1;
  data = holder;
};

string string::operator+(string second)
{
  char *holder = new char[size + second.size + 1];
  strcpy(holder, data);
```

```
  cout << "first: " << first << "\n";

  string sec("And this is an another.");
  cout << "Type in a string ..... ";

  cin >> sec;
  cout << "sec: " << sec << "\n";

  string third;
  third = sec+first;
  cout << "sec + first: " << third << "\n";

  third = sec+sec;
  cout << "sec + sec: " << third << "\n";

  third.insert(" plus");
  cout << "with insert:" << third << "\n";

  third = third + sec;
  cout << "added to itself:" << third << "\n";
};
```

```
    s = (char*) (theGetter->result ());
        while ( c = *s++ ) {
                if (c == '$') i++;
}
resultis (i);
}


void main() {
    getString getter;
        countDollar counter ( &getter);
        cout << "Result is: " << counter.result() << "\n";
    thistask->resultis(0); // the main routine is also a task
                  // and should be terminated by resultis()
}

// Program using queues
#include <task.h>
#include <stream.h>

class getString : public task {
    public:
        getString(qhead *,qtail*);
};

class countDollars : public task {
    public:
    countDollars(qhead *,qtail*);
};

class stringHolder :public object {
    public:
        stringHolder(char *aString) {theString = aString;};
        char *theString;
};

class numDollars:public object {
    public:
        numDollars(int count) {dollars = count; };
        int dollars;
};

getString::getString(qhead *countQ,qtail* stringQ)
{
    numDollars * cmessage;
    while (1) {
```

The enhanced `+w` option issues warning messages for in-line functions that are not in-lined. You may find this information useful for performance analysis.

C.2 New Enhancements: Release 2.1 to 3.0.1

Release 3.0.1 includes the following enhancements to Release 2.1:

- Template instantiation tools (`ptcomp` and `ptlink`)
- Template support
- True nested types
- Generates ANSI C code not K&R C code
- Options `-g` (debugging)and `-O` (optimization) can be used together

For other minor changes, see the *C++ 3.0.1 Language System Release Notes.*

C.3 C++ 2.1 K&R C and C++ 3.0.1 ANSI C Differences

This release of C++ 3.0.1 is based on ANSI C, not K&R C. This section describes the major differences between C++ 2.1 and 3.0.1 because of this change:

- `acpp` predefines the macro `__STDC__` (as `0`), `__TIME__`, and `__DATE__`, while `cpp` doesn't.
- `acpp` specifies that a new-line character immediately preceded by a `'\'` character is spliced together. `cpp` does not allow `'\'` in places other than `comment` and `string`. For example, the following constructs are legal in C++ 3.0.1, but not in C++ 2.1:

```
#def\
ine write printf

// this is a \
        comment
```

- `acpp` supports `##` as the preprocessor operator, which performs token passing. It also treats comments found within macro replacement as whitespace, hence delimiting adjacent tokens.

  `cpp` does not support the `##` operator. Instead, you may use a comment as illustrated:

```
#ifdef __STDC__
#define PASTE(A,B) A##B
#else
#define PASTE(A,B) A/**/B
#endif
```

- `acpp` will not replace a macro if the macro is found in the replacement list during the rescan. `cpp` will recursively substitute. For example:

```
#define F(X) X(arg)
F(F)
```

yields `"arg(arg)"` with `cpp` , and `acpp` issues the error:

```
fatal: macro recursion
```

- `acpp` supports the `#error` directive that causes the implementation to generate a diagnostic message with a user-specified token sequence. `cpp` does not support this directive.
- C++ 2.1 allows comments that start in an `include` file to be terminated in the file which includes the first file, C++ 3.0.1 does not.

- C++ 2.1 uses a bottom-up algorithm when parsing and processing partially elided initializers within braces. C++ 3.0.1 uses a top-down parsing algorithm. As an example, look at the following program:

```
#include <stdio.h>

struct A { int a[3], b; } w[2] = { {1 }, 2 };

main()
{
        for (int i = 0; i < 2; ++i)
        {
                print("w[%d].a = %d, %d, %d\n", i,
                        w[i].a[0], w[i].a[1], w[i].a[2]);
                print("w[%d].b = %d\n", i, w[i].b);
        }
}
```

When it is compiled and run under C++ 2.1, it gives the following output:

```
% CC test.cc
% a.out
w[0].a = 1, 0, 0
w[0].b = 2
w[1].a = 0, 0, 0
w[1].b = 0
```

When it is compiled and run under C++ 3.0.1, it gives the following output:

```
% CC test.cc
% a.out
w[0].a = 1, 0, 0
w[0].b = 0
w[1].a = 2, 0, 0
w[1].b = 0
```

C.5 New Enhancements: Release 1.2 to 2.0

Version 2.0 included significant enhancements to C++ Release 1.2:

- Multiple inheritance
- Default member-wise initialization and assignment
- Ability of each class to define its own `new` and `delete` operators
- Type-safe linkage

Type-Safe Linkage and Handling of Overloaded Function Names

Release 2.0 implemented type-safe linkage as described in Bjarne Stroustrup's paper, "Type-safe Linkage for C++." Now all function names are encoded, overloading of functions is now implicit (use of the `overload` keyword is now optional), and C functions must be explicitly declared as requiring C linkage (that is, you need to tell the translator those names should not be encoded). You do so by using the `extern "C"` declaration.

These changes might have caused some old code to break; however, the new linkage scheme fixes what proved to be a rather pernicious category of user bug. Overloading is now independent of the order the functions are declared. In addition, you gain some degree of type checking across files.

In past releases, the first instance of an overloaded function name was not encoded. This enabled the user to overload library function names, and still link with the existing library by defining its instance first. For example:

```
overload abs;
abs( int i );
double abs( double d );
complex abs( complex c );

main() {
     int i = abs( i );
}
```

will no longer link. `abs` will be encoded and `ld` will not find the `libc.a abs`. Instead, `abs` must be declared as requiring C linkage, as follows:

In the previous C++ 2.1 release there were also constraints to the number of `#include` files — both nested and non-nested — allowed. This is no longer true in C++ 3.0.1.

Don't forget that to conform with ANSI C, you should put a comma at the end of the first argument of the `printf()` declaration:

```
ifdef _ _STDC_ _
 void printf(char*, ...);
 int fread(char*, int,int,FJLEX);
 int getpid(void);
#else
 void printf();
 int fread();
 int getpid;
#endif
```

When user-defined "generic" header files are used for Sun C, the C++ include directory path has to be passed with the `-I` option to the `cc` driver to locate `c_varieties.h`.

This path is passed by the `CC` driver by default. One alternative (not necessarily recommended) is to include the file with the absolute path, as in `/.../include/CC/c_varieties.h`, or have a duplicate copy of the `c_varieties.h` in your source directory.

E.2 The `struct s { /* ... */ } s` *Tags*

In C++, `struct` tags are in the same name space as variables. This restriction is relaxed for ANSI C/C conformance. (Also see "C++: As Close As Possible to C — But No Closer" in the *C++ 3.0.1 Language System Selected Readings* manual for more information.).

functions and operators can be overloaded, their names are encoded by the C++ compiler. This makes it difficult to reference C++ functions by using the names shown in header files.

It is not realistic to expect that all existing C code will be rewritten in C++ or recompiled with a C++ compiler. First of all, much C code is used to provide libraries for users who may not have access to a C++ compiler. This leads to the same problem of encoded function names in a library that is to be linked with C programs.

Second, the performance of code generated by a C++ compiler may not be as good as that generated by a C compiler. As C++ compilers mature, this objection will go away. The problem is how to allow C programs to use code compiled with a C++ compiler. Any solution must be reasonably easy to use and not cost too much in execution efficiency.

F.2 Proposed Solution

To solve the problem, first separate it into three somewhat simpler problems. These are characterized by the type of C++ function which must be called by the C programs:

- Methods for C++ classes
- Overloaded operators (whether or not part of a class definition)
- Other C++ functions

Class Methods

Class methods are invoked by sending a message to an object instance of a class. This is really a function call with one extra hidden parameter; a pointer to the object being sent the message. The C language does not allow for objects. It does allow for effective use of pointers. As long as the C program has a pointer to the object in question, a simple scheme can be used to invoke the proper method for the object. The following example describes how this is done.

In C++, you have an access function, `Type()`, for objects of type `TComponentSet`. This returns the value of the type member variable (type `cpt_set_type_t`). If the variable `cSet` is a pointer to an object of type `TComponentSet` then you can invoke the method with the message:

It is invoked either statically by having global variables of type `TComponentSet` on entry to a function, or explicitly through the `new` operator. When attempting to create a new object from a C program, pointers are needed; so you will usually invoke the `new` operator. In the example, assume that there is a constructor for class `TComponentSet` that is declared as:

```
TComponentSet( cpt_set_type_t, cpt_name_t );
```

A corresponding C-callable function will be defined in the following way:

```
extern "C" {
     Handle WNewComponentSet( cpt_set_type t, cpt_name_t n )
     {
          TComponentSet *s;
          s = new TComponentSet( t, n );
          return (Handle)s;
     }
}
```

It is probably not necessary to use the temporary variable `s`; it may be more readable to do so. To use this function in a C program, you simply need code like the following:

```
Handle    h;
...
h = WNewComponentSet( myType, myName );
```

Now the handle `h` can be used in the invocation of functions like `WGetSetType()`.

Overloaded Operators

The technique used to invoke overloaded operators is similar to the previous technique. Realizing that an overloaded operator in C++ is just a function invocation, all that is necessary to invoke the function in a C program is to provide a wrapper that is not encoded. Two types of operator overloading can occur: overloading operators as part of a class definition, and overloading operators outside of a class definition. In the first case, a handle to the object

Other Functions

This case is the easiest of the three. The technique used has already been covered. It is similar to overloaded operators which are not part of a class definition. First of all, if there are no objects of any sort involved, and the function is not overloaded, then the function can just be wrapped with the `extern  { . . . }` syntax and it can be called directly from C programs.

If there are objects involved, all that is necessary is to build a wrapper where handles are used to identify the appropriate objects and then cast to the appropriate type in the C++ wrapper. For example, if you have the C++ function:

```
int Foo( Object1& o1, int i, Object2 *o2 );
```

You can build the wrapper for it as follows:

```
extern "C" {
     int WFoo( Handle o1, int i, Handle o2 )
     {
          return Foo( *(Object1 *)o1, i, (Object2 *)o2 );
     }
}
```

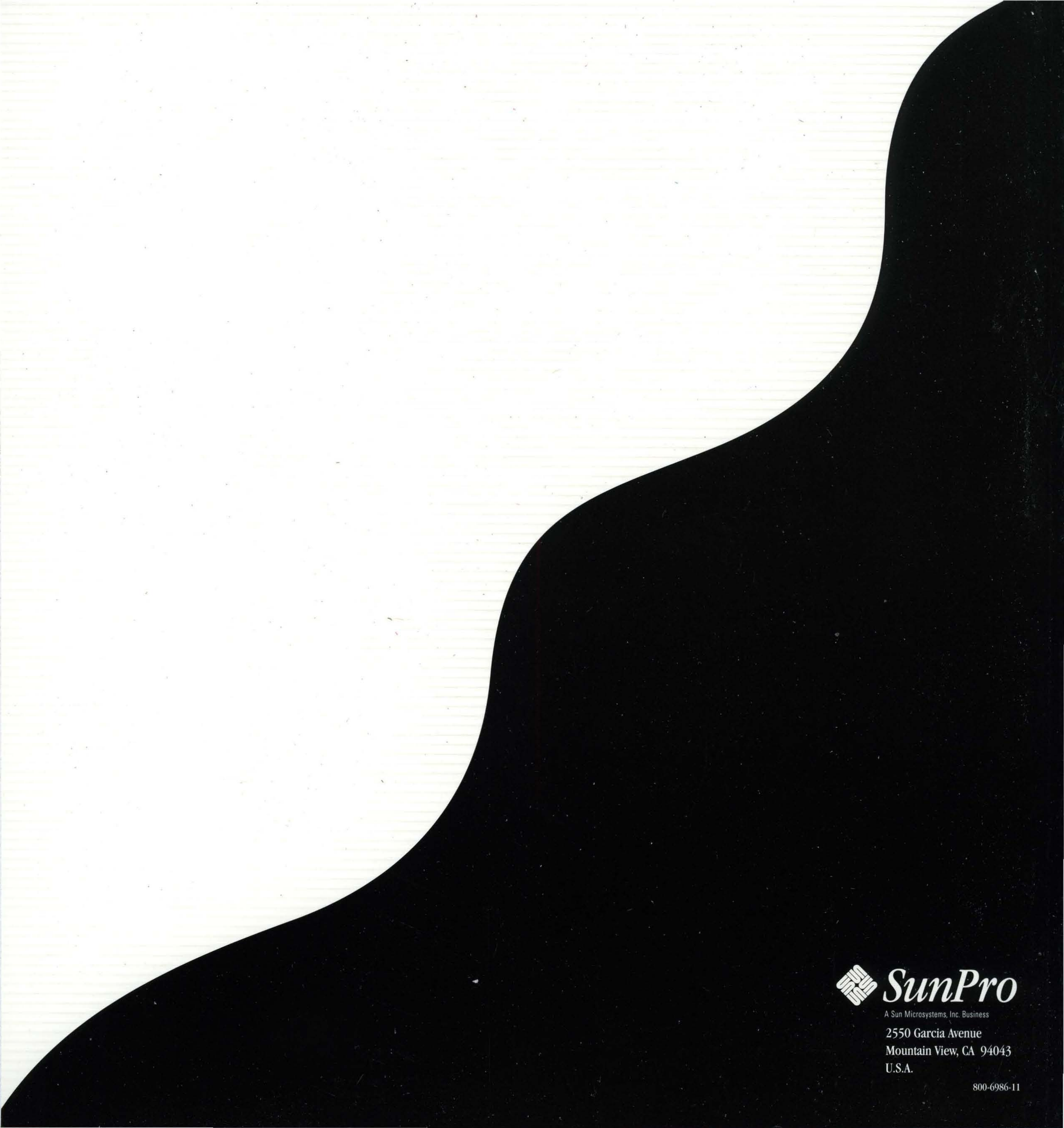
10.11 Comments

The proposed solution is far from ideal. It is more cumbersome than you would like to see when dealing with languages as close as C and C++. One view of a better solution would be to enable the C compiler to understand that a function is a C++ function and thereby call the appropriate encoded function (perhaps with `extern "C++" {...}` being recognized). This would also imply that there be a way to indicate something about classes and objects to the C compiler. This solution is rather difficult and complex for the language implementors.

It might be possible to simulate the above mechanism in a C program. This would involve knowing how functions are encoded and then calling the proper function, with the proper arguments (providing the address of an object as a "hidden" argument in the case of class methods). This would require additional header file information and should be rejected as being much too complex.

The Waite Group's C++ Programming , John Thomas Barry (Howard W. Sams & Co, 1988)

Using C++ , Bruce Eckel (Osborne/McGraw-Hill, July 1989)



SunPro
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043
U.S.A.