1

TR CS-93-16

A Guide to C++ for C Programmers

Thomas A. Anastasio

Computer Science Department
University of Maryland, Baltimore County
Baltimore, MD 21228

27 October 1993

A Guide to C++ for C Programmers

Thomas A. Anastasio

Computer Science Department
University of Maryland, Baltimore County
Baltimore, MD 21228

27 October 1993

Contents

1 Introduction

This guide is intended to help C programmers learn C++. It puts particular emphasis on the differences between C++ and ANSI C, and on the extensions which C++ makes to ANSI C. Most of the standard C++ texts assume that the reader doesn't know C. Those who do must therefore wade through a lot of basic information to get to the C++ parts. Those texts which assume a good knowledge of C tend to cover C++ in great depth and detail. While these latter texts are extremely valuable, they do not allow for a fast overview of the main C++ features.

While this guide assumes a good knowledge of C, it hits only the high points of C++. It has neither depth nor detail. Most topics are covered by example. For greater depth, detail, and generality, please consult one of the references in Section 11.

2 Basics

Here are some of the basic differences between C and C++:

Comments: In addition to the `/* */` comments, C++ ignores all text between `//` and the end of the line. Generally, you should use the `//` comment style. Note that the `cpp` preprocessor does not recognize this style. Therefore, you should use the `/* */` style if you use preprocessor macros.

Example

```
#define N 3 //  number of items
```

will result in macro substitution of `N` by `3 // number of items`. Anyway, you should not use preprocessor macros in C++. More on this below.

char: The size of a character constant is one byte in C++, but is the size of an `int` in C. The size of a variable of type `char` is one byte in both C++ and C.

Declarations: They can go anywhere (in scope) in C++, as long as they occur before first use.

Example:

```
for (int i=0; i<N; i++);
```

is legal C++ code. `i` is declared and defined as needed in the `for` loop. Its scope is the remainder of the block in which the `for` loop occurs.

String Assignment: Aggregate assignment of strings must leave room for the terminating `NULL` character. Example:

`char foo[3] = "abc";` is an error in C++, no room for '\0'
`char foo[4] = "abc";` is legal in C and in C++
`char foo[] = "abc";` is legal (and preferred) in C and C++

Type Conversion (Casting): Two forms are available in C++.

```
a = (int) b;
a = int(b);
```

The two forms are entirely equivalent. The second form is preferable because it corresponds to the form for operator overloading (more on this below).

Scope Resolution Operator: :: is the scope resolution operator. It has the highest precedence level (level 1) and associates left-to-right. When it appears as a unary operator, it causes global variable reference. Example:

```
int foo;

main()
{
  int foo;

  foo = 3;    // references local foo in main
  ::foo = 5;  // references global foo
}
```

The scope resolution operator also is used when referring to member functions of a class. For example, the member function `bar` of class `Foo` is referred to as `Foo::bar`. More on this in Section 3.2.

Preprocessor Macros: C++ encourages the use of `const` values and `inline` functions rather than macros. Unlike macros, `inline` functions allow for type and remain available by name in symbolic debuggers. Another reason is to avoid the well-known substitution problems which can occur with macros. Example:

Use `const int NUMB = 3;` rather than `#define NUMB 3`

Use `inline int cube(int x) {return x*x*x;};`

rather than `#define cube(x) x*x*x`.

In addition to `cube` being known by name in a symbolic debugger, the inline function approach avoids the famous macro-substitution problem:

```
cube(a+b) --> a+b*a+b*a+b
```

Function Prototyping: Function prototyping helps in detecting errors at compile time and allows for function name overloading (cf. Section 7.1). ANSI C encourages, but C++ *requires* full prototyping of functions. C++ also requires that every non-void function explicitly return an appropriate value.

Example:

```
int foo(float, int);   // a fully-prototyped C++ function declaration

main()
{
  foo(3.0,4);
}

int foo(float x, int y)   // definition of foo
{
  float a = x + float(y);
  return int(a);
}
```

The following legal C code is *not legal* in C++ (and is pretty shoddy in C):

```
foo();          /* return type not given, arguments not prototyped */

main()
{
  foo(3.0,4);
}

foo(float x, int y)       /* definition of foo with parameter list */
{                         /* different from declared parameter list */
  float a = x + (float) y;
  return (int)a;
}
```

Because C does not require function prototyping, programmers sometimes use library functions haphazardly. For example, some C programmers use `printf` without declaring it (either explicitly or by means of the header file `stdio.h`). It "works" in C because `printf` returns an `int` and because it is not necessary to declare the argument types. Because C++ requires function prototyping, it is necessary to correctly declare all functions. If you use `printf`, for example, you must either declare it explicitly or include `stdio.h`.

structs: Structure names and identifiers occur in a single name space in C++. Therefore, in C++ the keyword `struct` is not used when declaring structure variables.

Example:

```
struct Foo
{
  int a;
  char b;
};

main()
{
 Foo x;   // in C this would have to be:    struct Foo x
}
```

An implication of this is that a local structure name hides that name in an outer scope.

Example:

```
#include<stdio.h>

int Foo[20];

main()
{
  struct Foo
    {
      int a;
    };
  printf("size of Foo is %d\n",sizeof(Foo));

}
```

In C, the output for `sizeof(Foo)` is `80` (refers to the array `Foo`), while in C++, the output for `sizeof(Foo)` is `4` (refers to `struct Foo`).

An example of a structure definition in C++, and its use in a program is:

```
struct Foo
 {
   int size;
   float data[10];
 };
```

```
main()
{
  Foo x;       //  in C you would have to say   struct Foo x;  to
               // declare x as a structure variable
  x.size = 3;  //  select the size member of x
               // and change its value to 3
}
```

Except for the name space of structs, C and C++ treat structures in precisely the same manner. Access to the elements of a structure variable is by means of the `.` selector operator. There is no attempt to deny access to any of the members of a structure variable.

3 Classes

The C++ `class` is basically a `struct` with control over access to the elements and with member functions. The `class` allows easy implementation of the *abstract data type*, which is a model of the relationships among data elements along with definitions of the functions which may be performed on the data elements.

By analogy with structure variables, a variable of class type should be called a "class variable." However, most C++ programmers don't use this terminology, they call such a variable a "class instance" or an "instance of a class."

3.1 Access Control

Access to the members of a class instance can be *public*, *private*, or *protected.*

A `public` member of a class instance can be accessed freely. C++ and C structs give public access to their members. For example, the struct and class definitions below are equivalent.

```
struct Foo
 {
   int size;
   float data[10];
 };
```

```
class Foo
 {
  public:
   int size;
   float data[10];
 };
```

A `Private` member of a class instance can be accessed only through member or friend functions. (A *friend* of a class is any non-member which is declared by the class to be a friend. Friends are given access to the private members of the class.) The following program shows an illegal attempt to access private members of a class instance.

```
class Foo
 {
  private:
   int size;
   float data[10];
 };
```

```
main()   // a C++ program
{
  Foo x;       // declaration of x, a class instance
  x.size = 3;  // NOT LEGAL, the members of Foo instances are private
}
```

The declaration of `private` access in the above example is unnecessary, since the default access to members of a class instance is `private`. The default access to members of a struct variable is `public`.

A `Protected` member of a class instance can be accessed only by member and friend functions and through derived classes (more on this in Section 8).

3.2 Member Functions

A class allows a member to be a function. Such a function is called a *member function* or a *method*. For example, in

```
class Foo
 {
   int size;
   float data[10];
  public:
   void set_size(const int s) {size = s;};
   int get_size(void) {return size;};
 };
```

`set_size` and `get_size` are member functions, and access to them is `public`. Access to the other members of `Foo` is `private` by default. The following program show how one can use `set_size` and `get_size` to access the `size` member of a `Foo` instance.

```
class Foo
 {
   int size;
   float data[10];
  public:
   void set_size(const int s) {size = s;};
   int get_size(void) {return size;};
 };
main()
{
  Foo x;           // declaration of x, a class instance
  int y;

  x.set_size(3);    // use a member function to change the value of the
                    // private member size of x to 3
  y = x.get_size(); // use a member function to retrieve it
}
```

Note that `set_size` and `get_size` are accessed as members of the `Foo` instance `x` by using `x.set_size` and `x.get_size`.

In the above example, the member functions were declared and defined within the declaration of the class. It is not necessary to define member functions within the class declaration. Programs are usually much easier to read if you declare the member functions within the class declaration, but define them elsewhere. The scope resolution operator `::` is used when reference is made to a member function.

Example:

```
class Foo
{
  int size;
  float data[10];
 public:
  void set_size(const int);   // member function
  int get_size(void);         // declarations
};
main()
{
  Foo x;           // declaration of x, a class instance
  int y;

  x.set_size(3);    // use a member function to change the value of the
                    // private member size of x to 3
  y = x.get_size(); // use a member function to retrieve it
}
void Foo::set_size(const int s)  // definition of member function
{
  size = s;
}
int Foo::get_size(void)          // definition of member function
{
  return size;
}
```

3.2.1 this

Every C++ member function automatically gets a first argument named `this` in addition to the arguments you declare. The argument allows the function to refer to the particular class instance invoked. In the example above, for instance, the call to

```
x.get_size();
```

was "really" made as:

```
x.get_size(&x);
```

The address of the class instance **x** is passed as the first argument to every member function of the class. This, of course, means that C++ modified your function definition to include a first argument in addition to your arguments. The first argument is a pointer to a class instance and is named **this**.

In the example above, the programmer's definition of `Foo::set_size`

```
void Foo::set_size(const int s)  // definition of member function
{
  size = s;
}
```

is modified by the compiler to be:

```
void Foo::set_size(Foo * const this, const int s)  // actual definition
{
  size = s;
}
```

(i.e., **this** is a constant pointer to a (mutable) Foo instance).

Therefore, every member function gives you automatic access to the class instance to which it applies. You may use the argument **this** just as if you had declared it yourself (except that you may not change it to point to something else).

3.2.2 Constructors and Destructors

Constructors and destructors are member functions which C++ uses to initialize and delete class instances. The name of a constructor is required to be the same as the class name. The name of the destructor is required to be the same as the class name, preceded by a tilde.

Note that constructors and destructors are used in C, but the programmer doesn't get to deal with them. For example, in the block:

```
{
  struct
   {
     int i;
     float f;
   } s;
}
```

an implicit constructor allocates memory for the struct variable `s`. Upon leaving the block, an implicit destructor frees the memory. Constructors and destructors are also provided implicitly in C++, but the programmer may also explicitly defined them for classes.

The following example shows the class Foo with explicit constructor and destructor functions:

```
class Foo
{
  int size;
  float data[10];
 public:
  Foo(void);
  Foo(int);
  Foo(int,float);
  Foo(float);
  ~Foo(void);
  void set_size(const int);
  int get_size(void);
};
```

There are four constructor functions declared. They are all named Foo, the same name as the class, but they have different arguments. This is an example of overloading (cf. Section 7.1). The actual arguments used in the call determine which constructor is invoked.

We now define the four constructors and the single destructor:

```
Foo::Foo(void)
   // this constructor takes no arguments
   // and initializes everything to zero
{
 size = 0;
 for (int i=0; i<10; i++) data[i] = 0.0;
}
Foo::Foo(int size_init)
  // this contructor takes an int argument and
  // initializes size to the argument, and  data[] to 0.0
{
  size = size_init;
 for (int i=0; i<10; i++) data[i] = 0.0;
}
Foo::Foo(int size_init; float data_init)
    // this constructor takes an int and a float and
    // initializes size and data[] accordingly
{
  size = size_init;
  for (int i=0; i<10; i++)  data[i] = data_init;
}
```

```
Foo::Foo(float data_init)
   // this constructor takes a float argument and
   // initializes size to 0 and data[] accordingly
{
  size = 0;
  for (int i=0; i<10; i++) data[i] = data_init;
}
Foo::~Foo()
  // this is the destructor.  It really doesn't do anything useful
{
 cout << "Foo destructor called" << endl;
}
```

Note that the constructor and destructor functions have no declared return type. There can be no more than one destructor function for a class, and it has no arguments.

The destructor is automatically called when the class instance goes out of scope. When a class contains a data member that points to dynamically allocated memory, this memory must be deallocated when the class instance goes out of scope. This is one of the main uses for destructors. The destructor function would be written to explicitly free the memory.

The following example shows definition of four `Foo` class instances, each one using a different constructor:

```
{
  Foo w;         // no arguments. size and data[] zeroed
  Foo x(3);      // one int argument.  size set to 3, data[] zeroed
  Foo y(3,2.5); // int and float args.  size set to 3, data[] to 2.5
  Foo z(2.5);    // one float arg. data[] set to 2.5, size zeroed
}
```

3.2.3 Constructor Initialization Lists

The primary purpose of a class constructor is to "construct" a class instance and initialize its members. C++ allows default initial values to be provided by means of the *constructor initialization list*. This is a list of initial values for the members of the class.

A constructor initialization list is a comma-separated list of member names and their initial values. It is separated from the function declaration by a colon.

Example:

```
class Foo
{
  int a;
  int b;
  int c;
 public:
  Foo();
};
```

```
// the constructor initializes a to 1, b to 2, c to 3
Foo::Foo() : a(1), b(2), c(3)
{ }
```

Here is an extended example showing initialization of various kinds:

```
#include <iostream.h>
const int NUMB = 5;

class Foo
{
  int data[NUMB];
  int size;
 public:
  Foo(int=99);          // an initial value for the argument initval
  void print(void);
};
main()
{
  Foo x;
  Foo y(3);
  cout << "class instance 'x'" << endl;
  x.print();
  cout << "class instance 'y'" << endl;
  y.print();
}
void Foo::print(void)
{
 cout << "size = " <<  size << endl;
 cout << "data = "  ;
 for (int i = 0; i<NUMB; i++) cout << data[i] << " " ;
 cout << endl;
}
Foo::Foo(int initval) : size(5)  // constructor initialization list
{
  for (int i=0; i<NUMB; i++)
    data[i] = initval;
}
```

The above program produces the output

```
class instance 'x'
size = 5
data = 99 99 99 99 99
class instance 'y'
size = 5
data = 3 3 3 3 3
```

4 Streams and I/O

C++ handles input and output through streams. A stream can be thought of as simply a sequence of data. The big reason to use stream operators rather than `printf` and `scanf` is that you can overload the stream operators to work with your own data types in addition to the built in data types.

The definitions of the stream objects `cin`, `cout`, and `cerr` are in the include file `iostream.h`. `cin` defaults to the keyboard and is equivalent to `stdin`. `cout` and `cerr` default to the screen and are equivalent to `stdout` and `stderr`, respectively.

There are two stream operators `<<` (the *insert* operator) and `>>` (the *extract* operator). Think of these operators as taking two arguments. The left-hand argument is a stream and the right-hand argument is an object of some data type. As defined in `iostream.h`, `<<` and `>>` take arguments restricted to the built-in data types (e.g. `int`, `float`, `char`, etc.) and to streams. Overloading the operators to also take non-built-in data types will be discussed below. Expressions involving the stream operators evaluate to their stream argument.

For example, the following program requests input from the keyboard by placing the string `"Enter an integer"` onto the output stream `cout`. The entered value is assigned to `i` from the input stream `cin`. Finally, the string `"You entered "`, the value of `i`, and a newline are placed on the output stream `cout`.

```
#include <iostream.h>

main()
{
  int i;

  cout << "Enter an integer: ";
  cin >> i;
  cout << "You entered " << i << '\n';
}
```

Note that there are no formatting requirements. The appropriate format is determined by C++ based on the data type of the arguments.

The stream operators `<<` and `>>` have precedence level 7 and associate left-to-right. Look at the line

```
cout << "You entered " << i << '\n';
```

Since `<<` associates left-to-right, the line is to be understood as:

```
(((cout << "You entered ") << i) << '\n')
```

Since an expression involving `<<` evaluates to a stream, each of the uses of the `<<` operator has a left-hand argument of `cout` and a right-hand argument of some built-in data type.

4.1 Formatting I/O

Each of the i/o stream class instances (`cin`, `cout`, and `cerr`) has a member variable which keeps track of formatting. This variable, known as the *format state*, can be manipulated by means of class *manipulators*. There are manipulators for changing the numeric base, for control characters, and for format control.

4.1.1 Numeric Base Manipulators

The number base for input and for output can be changed by means of the manipulators `dec`, `hex`, and `oct`.

Example:

```
#include <iostream.h>
main()
{
  int i = 35;    // nothing special about this value

  cout << i << '\n';        // prints the value of i in decimal format
  cout << hex;              // changes the number base to hexadecimal
  cout << i << '\n';        // prints the value of i in hexadecimal format
  cout << i << '\n';        // still in hex format
  cout << oct << i << '\n'; // prints the value of i in octal format
  cout << dec;              // back to decimal format
}
```

4.1.2 Character Control Manipulators

Three handy character control manipulators are `endl`, `ends`, and `flush`. `ends` simply inserts a null character into the stream. It is equivalent (and preferable) to using `'\0'`. The manipulator `endl` inserts a newline, and also flushes the buffer. Finally, `flush` simply flushes the buffer. You may wish to flush the buffer when you want output to appear immediately.

4.1.3 Format Control Manipulators

Numeric format can be controlled by means of the manipulators `setprecision(int)` and `setw(int)`. General format can be controlled by the manipulators `setiosflags(flag)`, and `resetiosflags(flag)`. Note that these manipulators look like functions and take arguments of various types.

You must include `iomanip.h` if you use format control manipulators. `setprecision(int)` specifies output floating point precision. The default precision is 6. `setprecision(0)` resets the precision to the default value;

`setw(int)` specifies the width of the data field to be printed. The default width is whatever is necessary to correctly display the data. This is the only manipulator which resets automatically.

Example:

```
#include <iostream.h>
#include <iomanip.h>

main()
{
  int i = 123456;
  float f = 1.2345678;

  cout << i << endl;
  cout << setw(10) << i << endl;
  cout << setw(3) << i << endl;  // actually will use minimum necessary
  cout << i << endl;
  cout << f << endl;
  cout << setw(15) << f << endl;
  cout << setprecision(3) << f << endl;
  cout << setw(15) << f << endl;
  cout << setprecision(0) << f << endl;  // reset precision to default
}
```

prints:

```
123456
    123456
123456
123456
1.23457
        1.23457
1.23
           1.23
1.23457
```

`setiosflag(flag)` and `resetiosflag(flag)` take *format flag* arguments. These flags are bits from the format status of `cin`, `cout`, and `cerr`. C++ provides the following flags:

```
ios::left        Left justify output
ios::right       Right justify output
ios::scientific  Use scientific notation for numbers
ios::fixed       Use regular format for numbers
ios::uppercase   All hex characters and scientific notation in uppercase
ios::showbase    Hex output preceded by 0x, octal preceded by 0
ios::showpos     Use a leading + sign for positive numbers
```

Use the flags with `setiosflags` to set them and with `resetiosflags` to reset them.

Example:

```
#include <iostream.h>
#include <iomanip.h>

main()
{
  int i = 123456;
  float f = 1.2345678;

  cout << i << endl;
  cout << setiosflags(ios::left) << setw(10) << i << endl;
  cout << setiosflags(ios::right) << setw(10) << i << endl;
  cout << f << endl;
  cout << setiosflags(ios::scientific) << f << endl;
  cout << setiosflags(ios::uppercase) << f << endl;
  cout << resetiosflags(ios::scientific) << f << endl;
  cout << setiosflags(ios::showbase ) << hex << i << endl;
}

prints:

123456
123456
    123456
1.23457
1.234568e+00
1.234568E+00
1.23457
0X1E240
```

5 Pointers and References

C++ provides for `reference` variables. The primary use for references is as the arguments to overloaded operators. Although references also provide a cleaner mechanism for "call by reference" arguments than can be obtained with pointers, the possible loss of program clarity may make this use inadvisable. It is not possible to tell, from the form of the function call, that the argument may be changed by the call.

A reference type is sometimes called an *alias* because a reference variable serves as an alternative name for another data object. A reference type must be initialized upon declaration (i.e., the declaration is a definition).

Although the primary use for references is as arguments and return values of functions, here is an example which does not involve functions:

```
int val = 10;        //  declaration and initialization of
                     // an ordinary int variable val
int &ref1 = val;     //  declaration and initialization of
                     // an int reference variable ref1
cout << val << endl;  // prints 10
ref1 = 20;
cout << val << endl;  // prints 20
```

`ref1` is an alias for `val` and any operation on ref1 is also performed on val (and vice versa).

References are closely related to pointers. The following example compares pointers and references.

```
#include <iostream.h>

main()
{
 int foo, bar;
 int &fooref = foo;      // fooref is an alias for foo
 int *fooptr = &foo;     // fooptr is a pointer to foo

 foo = 3;
 bar = 4;
           // the following prints     3 3 3 4
 cout << fooref  << *fooptr << foo << bar << endl;

 fooptr = &bar;    // now fooptr is a pointer to bar
           // the following prints     3 4 3 4
 cout << fooref  << *fooptr << foo << bar << endl;

 fooref = bar;     // fooref is still an alias for foo.  we have
                   // assigned the value of bar to foo

           // the following prints     4 4 4 4
 cout << fooref  << *fooptr << foo << bar << endl;

 bar = 5;   // remember, fooptr points to bar, fooref is an alias
           // for foo

           // the following prints     4 5 4 5
 cout << fooref  << *fooptr << foo << bar << endl;
}
```

5.1 Constant References and Pointers

There are four interesting combinations of constants, references, and pointers.

Pointers to Constants: A pointer to a constant can be changed to point to some other constant, but the item it points to cannot itself be changed.

Example:

```
const float * val;   //  val is a pointer to a float constant
const float foo = 3.0;
const float bar = 4.0;

val = &foo;          // val now points to the float constant foo
val = &bar;          // val now points to the float constant bar
```

Constant Pointers: A constant pointer cannot be changed to point to something else, but the item it points to may itself be changed.

Example:

```
float foo;
float * const val = &foo;   // val is a constant pointer to foo
                            // and cannot be changed to point to anything
                            // else
```

Constant Pointers to Constants: A constant pointer can not be changed to point to something else, and the item it points to cannot itself be changed.

Example:

```
const float foo = 3.0;
const float * const val = &foo;   // val is a constant pointer to foo
                                  // it cannot be pointed to anything
                                  // else, nor can foo be changed
```

References to Constants: Recall that a reference is always constant in the sense that it refers to one variable only and cannot be changed to refer to another variable. A reference can also be made to a constant. Example:

```
const float &ref = 1.2;    // ref is a reference to the constant foo.
```

A temporary is created to hold the value 1.2, and the reference is made to that temporary. It's as if the following were done:

```
float *ref;
float temp;
temp = 1.2;
ref = &temp;
```

5.2 References and Functions

References find their primary use in passing arguments to and returning values from functions. The following example shows how C and C++ pass an argument by reference, using pointers.

```
#include <iostream.h>

void get_data(int *);

main()
{
  int val;

  get_data(&val);
  cout << "val is " << val << " in main"  << endl;
}

void get_data(int *x)
{
  cout << "Enter an integer: ";
  cin >> *x;
}
```

The same effect can be obtained with references. Example:

```
#include <iostream.h>

void get_data(int &);

main()
{
  int val;

  get_data(val);
  cout << "val is " << val << " in main"  << endl;
}

void get_data(int &x)
{
  cout << "Enter an integer: ";
  cin >> x;
}
```

6 Memory Allocation

C and C++ both support `malloc()` and `free()`. Since function prototyping is required in C++, it is necessary to include either `alloc.h` or `stdlib.h`. C++ has a better way to allocate and free memory by means of the operators `new` and `delete`. Since `new` and `delete` are operators, they can be overloaded to allow programmer control of the allocation/deallocation process. For example, the programmer might implement a garbage-collection mechanism.

When allocation is done in the form `new T[size]`, deletion is done in the form `delete []`. You must include `new.h` when using `new`.

Examples:

```
#include <new.h>
main()
{
char *cvar = new char ;   // Allocates a new character
int *ivar = new int;      // Allocates a new integer

float *fvar = new float[50]; // Allocates 50 contiguous floats

float *foo  = new float[10] (0.0) // Allocates 10 contiguous floats, and
                                  // initializes them to 0.0

delete [] foo;  // De-allocates the memory used by foo
delete [] fvar; // De-allocates the memory used by fvar
delete cvar;    // De-allocates the single character in cvar
delete ivar;    // De-allocates the single int in ivar
}
```

C++ provides a mechanism for handling allocation errors (such as when `new` cannot allocate the required memory). The function `set_new_handler` is declared in `new.h` as

```
void (*set_new_handler(void(*)()))()
```

namely a void function which takes a function pointer (which points to a function returning void) as its argument. You write an allocation error handler function (which returns void), and pass its address to `set_new_handler`. When no allocation handler is provided, `new` simply returns the null pointer upon failure

The following example shows the use of an allocation error handler which prints the message `An allocation error has occurred` and exits.

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

void my_handler();    // this function will be invoked if an allocation
                      // error occurs.  The function takes no arguments.

main()
{
 set_new_handler(my_handler);
 float *foo = new float [1000000000];  // too much memory
}

void my_handler()
{
 cout << "An allocation error has occurred" << endl;
 exit(1);
}
```

7 Function and Operator Overloading

A function or operator is said to be *overloaded* when it can be used with a variety of data types. C already has overloaded operators. For example, the operator `+` can be used with a variety of data types such as int, float, double, char, etc. How does a C compiler "know" which `+` operator to use? First it promotes the arguments to the same type (if necessary), then uses the `+` appropriate for that type. C does not support overloaded functions or overloading of operators by the programmer.

7.1 Function Overloading

C++ supports overloaded functions by a process of matching actual and declared argument lists. The *signature* of a function is a list of the data types of the declared arguments of the function. The signature along with the name of a function must be unique. The function `foo()` is overloaded in the following example:

```
#include <iostream.h>

void foo(int x)
{
  cout << "integer " << x << endl;
}
void foo(float x)
{
  cout << "floating point " << x << endl;
}
```

```
main()
{
  float f = 1.23;
  int   i = 4;

  foo(f);
  foo(i);
}
```

Even though `foo` has the same `void` return type in both instances, this has no effect on the overloading. Only the signature of a function is used to determine the overloading.

When a function of the same name is declared multiple times in a program, all declarations after the first are treated (by the compiler) as follows:

- If the declaration matches in both return type and signature, it is considered a redeclaration.
- If the declaration matches in signature, but not in return type, it is an error.
- If the signatures differ (in either number or type of argument), the function name is overloaded. The return types don't matter, only the signatures.

7.1.1 Resolving Overloaded Functions

The particular instance of an overloaded function is resolved by argument matching (comparing the actual arguments with the declared arguments). There are three possibilities:

- The particular instance is determined because the arguments match. The arguments are said to match when they pass any one of the following three tests (applied in order):
  1. The type of each argument in the signature is precisely the same as the type of the corresponding argument in the actual argument list.
  2. The type of each argument in the signature is the same as the type of the corresponding argument in the actual argument list after applying *standard* type conversions.
  3. The type of each argument in the signature is the same as the type of the corresponding argument in the actual argument list after applying a *programmer-defined* type conversion.
- The particular instance cannot be determined because the arguments do not match. A compiler error results.
- The particular instance cannot be determined because the arguments do not match unambiguously. There is more than one instance which matches. A compiler error results.

Because the matching process can involve type conversion(s), the ambiguous case can be treacherous. In the following example, the `float` argument can be converted to an `int` to match the `foo(int)` case, but the `int` can be converted to a `char` to match the `foo(char)` case. Therefore, it is not possible for the compiler to unambiguously determine which of the overloaded instances of `foo()` to call.

```
void foo(int x){}
void foo(char x){}

main()
{
  float f = 1.23;

  foo(f);
}
```

7.1.2 Function Name Mangling

Obviously, overloaded functions must refer to different functional entities. How does C++ produce C code which compiles overloaded functions to different addresses? It generates a name for each of the overloaded possibilities. This process is called *name mangling*. In fact, C++ mangles the name of *every* function, whether overloaded or not. Name mangling is normally invisible (and of little interest) to the programmer.

One place in which it becomes important is when you wish to link ordinary C functions into your C++ programs. You don't want C++ to mangle the names of your C functions. The way to avoid name mangling is by using the *C linkage directive*. For example, to avoid mangling the name of the C function `int foo_in_c(float)`, you would declare it as:

```
extern "C" int foo_in_c(float);
```

Similarly, to include the standard C math library, use:

```
extern "C" {
#include <math.h> }
```

7.2 Operator Overloading

C++ supports overloading of operators for programmer defined data types. For example, the insertion operator, `<<`, can be overloaded to allow customized printing of class instances.

Programmer defined operators are treated as functions. The operator function must take at least one class argument (this precludes overloading operators for the built-in data types). An operator function is defined in the same way as an ordinary member function, but its name must consist of the reserved word `operator` followed by one of the predefined C++ operators. The programmer may not "invent" a new operator. The precedence and associativity of the operator are unchanged.

The assignment `=` and address-of `&` operators are defined by the compiler upon declaration of any class type. The programmer can depend on these two operators being defined, but might want to overload the definitions for particular classes (see Section 7.2.7).

7.2.1 Friend vs Member Functions

Any non-member function can be declared to be a *friend* of one or more classes. A friend function has full access to all the members of a class instance passed to it as an argument. Bjarne Stroustrup recommends that operator overloading should be done by means of a member function when there is a choice.

Member functions *must* be used for overloading the assignment `=`, the function call `()`, the subscript `[]`, and the dereference `-> *` operators.

Friend functions *must* be used for operators which have a left-hand side which is not an instance of the class. Recall that every member function is supplied a first argument of a pointer to the calling class instance. The compiler does not supply friend functions with extra arguments.

7.2.2 Comparison Operators

The following extended example shows the overloading of the comparison operator `==` by means of a friend function. The other comparison operators are done in a similar fashion. The function takes two arguments, both of them references to `Foo` instances. The first argument refers to the left-hand side of the `==` expression, while the second argument refers to the right-hand side.

```
#include <iostream.h>
const int NUMB = 5;
const int TRUE = 1;
const int FALSE = 0;

class Foo
{
  int data[NUMB];
  int size;
 public:
  Foo(const int=99);           // an initial value for the argument initval
  void set_data(const int);    // a member function
  friend int operator==(const Foo &, const Foo &); // overloaded ==
};
main()
{
  // define three Foo instances.  x and z are the same. y is different
  Foo x(1);
  Foo y(2);
  Foo z(1);

  // uses of the overloaded == operator
  cout << "Before resetting data[] of x" << endl;
  cout << "does x == y? " << (x==y) << endl;
  cout << "does x == z? " << (x==z) << endl;
  cout << "does y == z? " << (y==z) << endl;
  x.set_data(2); // now x and y are the same.  z is different
  cout << "After resetting data[] of x" << endl;
  cout << "does x == y? " << (x==y) << endl;
  cout << "does x == z? " << (x==z) << endl;
  cout << "does y == z? " << (y==z) << endl;
}
```

```
Foo::Foo(const int initval) : size(5)  // constructor
{ for (int i=0; i<NUMB; i++) data[i] = initval; }
  // definition of overloaded == as a non-member
  // function which is a friend of class Foo
int operator==(const Foo & lhs,const Foo & rhs)
{
  int test = FALSE;

  test = (lhs.size == rhs.size);  // this is the built in ==
  for (int i=0; i<NUMB; i++)
    test = test && (lhs.data[i] == rhs.data[i]); // so is this
  return test;
}

void Foo::set_data(const int val)   // a member function
{ for(int i=0; i<NUMB; i++) data[i] = val; }
```

The output of the above program is:

```
Before resetting data[] of x
does x == y? 0
does x == z? 1
does y == z? 0
After resetting data[] of x
does x == y? 1
does x == z? 0
does y == z? 0
```

7.2.3 Math Operators

The programmer must determine the meaning of the math operators in the context of classes. For example, what does it mean to "add" two instances of the `Foo` class? Let's say that it means "the integer equal to the sum of the `size` members of the two instances." However, we might also like to be able to add an integer and a Foo instance. We therefore will need a variety of overloaded `+` operators. The following example implements the overloaded `+`.

```
#include <iostream.h>
const int NUMB = 5;

class Foo
{
  int data[NUMB];
  int size;
 public:
  Foo(const int=99);            // an initial value for the argument initval
  void set_size(const int);     // a member function
  friend int operator+(const Foo &, const Foo &);  // overloaded operator +
  friend int operator+(const Foo &, const int);    // done with
  friend int operator+(const int, const Foo &);    // friend functions
};
main()
{
  // define some Foo instances
  Foo x(1);
  Foo y(2);

  cout << x + y << endl;
  cout << x + 10 << endl;
  cout << 13 + x << endl;
  y.set_size(17);
  x.set_size(29);
  cout << x + y << endl;
}
  // definitions of overloaded + operator as non-member
  // functions which are friends of class Foo
int operator+(const Foo & lhs,const Foo & rhs)
{  return (lhs.size + rhs.size); }

int operator+(const Foo & lhs,const int rhs)
{  return (lhs.size + rhs); }

int operator+(const int lhs,const Foo & rhs)
{  return (lhs + rhs.size); }
```

```
Foo::Foo(const int initval) : size(5)
{
  for (int i=0; i<NUMB; i++)
    data[i] = initval;
}

void Foo::set_size(const int val) { size = val; }
```

The output from the program is:

```
10
15
18
46
```

7.2.4 I/O Operators

The insertion operator `<<` and the extraction operator `>>` are overloaded by the definitions in the include file `iostream.h`. For example, the `<<` operator is overloaded by functions like:

```
    ostream& operator<<(char c)
    ostream& operator<<(int a)
    ostream& operator<<(long)
    ostream& operator<<(double)
```

There is one such function for every built-in data type. An `ostream` is a class of which `cout` is an instance. The functions are public member functions of the class so they take just one argument. Notice that they return a reference to an `ostream` class instance. In fact, they all return `*this`, the stream on the left-hand side of the expression.

In the following example, the function used to overload `<<` will be a friend function. It therefore takes two arguments, the first of which is is a reference to `ostream` (the left-hand side of the `<<` expression) and the second of which is a reference to `Foo` type. The function returns its first argument so that the value of `<<` expressions will be the stream instance itself (see Section 4).

```
#include <iostream.h>
const int NUMB = 5;

class Foo
{
  int data[NUMB];
  int size;
 public:
  Foo(const int=99);          // an initial value for the argument initval
  friend ostream & operator<<(ostream &, const Foo &);
};
```

```
main()
{
  // define some Foo instances
  Foo x(1);
  Foo y(2);

  cout << x << y << endl;
}
  // definition of overloaded <<
ostream & operator<<(ostream & out, const Foo & rhs)
{
  cout << "--- A Foo instance ---" << endl;
  cout << "size is " << rhs.size <<  endl;
  cout << "data[0] is " << rhs.data[0] << endl;
  cout << "----------------------" << endl;
  return out;
}

Foo::Foo(const int initval) : size(5)
{
  for (int i=0; i<NUMB; i++)
    data[i] = initval;
}
```

Output from the above program is:

```
--- A Foo instance ---
size is 5
data[0] is 1
----------------------

--- A Foo instance ---
size is 5
data[0] is 2
----------------------
```

Overloading of the input operator is done in an analogous way. One difference is that the second argument, of type `Foo &`, is not a constant because it will be changed as a result of the input.

7.2.5 Subscript Operator

The following example shows overloading of the subscript operator `[]`. C++ requires that this be done with a member function. Subscripting is considered a binary operator. The left operand is the object being subscripted. The right operand is the index. The function tests for the index being in range (exits with an error message if not), then returns a reference to the `data` element. A reference is returned to allow assignment to be made to the element.

Example of overloading `[]` to allow indexing into a `private` member of a `Foo` instance.

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

const int MAX = 5;

class Foo
{
  int *data;
  int size;
 public:
  ~Foo() { delete data; };
  Foo(const int=99);      // an initial value for the argument initval
  int * get_data();
  int & operator[](const int); // overloaded subscript operator
  void print();
};
main()
{
  Foo x(MAX);
  int i;

  for (i = 0; i < MAX; i++) x[i] = i;
  x.print();
  for (i = 0; i < MAX; i++) x[i] = x[i] - 1;
  x.print();
  x[MAX] = x[0];   // MAX is out of range for x index
}
int & Foo::operator[](const int index)
{
  if ( (index < 0) || (index >= size) )
    {
      cerr << "Index [" << index << "] out of range [0.." << size-1;
      cerr << "]" << endl;
      exit(1);  // bail out
    }
  return data[index];
}
```

```
void Foo::print()
{
  for (int i = 0; i < size; i++)
     cout << data[i] << " ";
  cout << endl;
}
Foo::Foo(const int initval)
{
  size = initval;
  data = new int[size];
}
```

Output from the above program is:

```
0 1 2 3 4
-1 0 1 2 3
Index [5] out of range [0..4]
```

7.2.6 Function Call Operator

The function call operator `()` can be overloaded as a class member function. It is considered a binary operator with left-hand operand of a class instance and right-hand operand of a list of parameters to the function. The following example overloads `()` to output all `data` values between a low and a high index, passed to the function as parameters. The function call `x(3,7)` is interpreted as `x.operator()(3,7)`, and could have been made in that (less elegant) form.

This example is intended only to show a simple use of the function call operator. A general way of iterating through a class instance's data is shown in Section 9.

```
#include <iostream.h>

class Foo
{
  int *data;
  int size;
 public:
  Foo(const int);
  void operator () (const int,const int); // overloaded call operator
};
main()
{
  Foo x(10);

  x(3,7);
}
```

```
Foo::Foo(const int s)
{
  size = s;
  data = new int[size];
  for (int i = 0; i < size; i++)
    data[i] = i;
}
void Foo::operator () (const int low, const int high)
// Prints out data values between indices low and high, inclusive.
// No attempt is made to check validity on low and high.
{
  for (int i = low; i <= high; i++)
    cout << data[i] << endl;
}
```

Output from the program is:

```
3
4
5
6
7
```

7.2.7 Assignment Operator

Overloading of the assignment operator `=` can only be done by a member function. C++ provides a default assignment operator when a class is defined. This default operator does member-by-member copying, and is not always suitable. In particular, the default assignment operator is usually not suitable when a class member is a pointer. The reason is that the *pointer* is copied resulting in multiple class instances having the same pointer member. Changes to the pointer member of one instance affect the pointer member of the other instance. For instance, if the pointer member of one instance is deleted, the other instance ends up with a dangling pointer. This could happen even in the simple case of one instance going out of scope while the other instance does not. The following example shows that the pointer members get copied. Note also the definition `y = 2`. It is essentially the same as `y(2)`. This form of definition involves a type conversion from `int` to `Foo` using the `Foo(const int)` constructor. It does not involve the assignment operator (more on this in Section 7.2.8).

```
#include <iostream.h>

class Foo
{
  int *data;
  int size;
 public:
  ~Foo() { delete data; };
  Foo(const int=99);      // an initial value for the argument initval
  int * get_data();
};
main()
{
  Foo x(1);
  Foo y = 2;

  cout <<  "     x      " <<  "      y      " << endl;
  cout << x.get_data() << " " << y.get_data() << endl;
  y = x;
  cout << x.get_data() << " " << y.get_data() << endl;
}
int * Foo::get_data() { return data; }

Foo::Foo(const int initval)
{
  size = initval;
  data = new int[size];   // allocation from the heap
}
```

The output from a particular execution of the above program is as follows. Note that the `data` pointer in the `y` instance is the same as that in the `x` instance after the assignment. The storage pointed to by the former `y data` pointer is no longer accessible.

```
     x          y
0x10003000 0x10004000
0x10003000 0x10003000
```

The following example overloads the assignment operator `=` for `Foo` so that the old `data` storage is returned to the heap and new storage is allocated. The `x` and `y` `data` pointers are not the same after the assignment.

```
#include <iostream.h>

class Foo
{
  int *data;
  int size;
 public:
  ~Foo() { delete data; };
  Foo(const int=99);      // an initial value for the argument initval
  int * get_data();
  Foo & operator=(const Foo &); // overloaded assignment operator
};
main()
{
  Foo x(1);
  Foo y(2);

  cout <<  "     x     " <<  "     y     " << endl;
  cout << x.get_data() << " " << y.get_data() << endl;
  y = x;
  cout << x.get_data() << " " << y.get_data() << endl;
}
Foo & Foo::operator=(const Foo & rhs)
{
  size = rhs.size;
  delete data;
  data = new int [size];
  for (int i=0; i<size; i++) data[i] = rhs.data[i];
  return *this;
}
int * Foo::get_data() { return data; }

Foo::Foo(const int initval)
{
  size = initval;
  data = new int[size];
  *data = initval;
}
```

The output from the above program is as follows.

```
     x          y
0x10003000 0x10004000
0x10003000 0x10003008
```

7.2.8 Type Conversion Operators

Section 7.2.7 showed a type conversion from `int` to `Foo`. The compiler makes the conversion by using the constructor which takes an `int` argument. Type conversion can be overloaded further. For example, to convert from `Foo` to `int` can be done as in the following example. The output is `43`. The type conversion operator `int()` is in the preferred C++ form (see Section 2).

```
#include <iostream.h>

class Foo
{
  int *data;
  int size;
 public:
  ~Foo() { delete data; };
  Foo(const int=99);      // an initial value for the argument initval
  operator int(); // overloaded type conversion to int
};

main()
{
  Foo x(43);

  int i = x;
  cout << i << endl;
}
Foo::operator int() { return size; }

Foo::Foo(const int initval)
{
  size = initval;
  data = new int[size];
}
```

8 Inheritance

C++ allows a class to inherit from another class. The inherited class is called a *derived* class. The class from which inheritance is derived is called the *base* class. Since inheritance is hierarchical, it is possible for a derived class to be some other class' base class.

The derived class inherits all the members of its base class (except that constructors and destructors are not inherited). Instances of the derived class do not have access to `private` members of the base class.

In the following example, the base class `Person` deals with information relevant to all persons. The derived class `Student` deals with information relevant only to students. Since students are persons, it is useful to be able to inherit the members of the `Person` class rather than have to repeat them.

```
class Person
{
 protected:
  char name[50];
  char address[50];
  char city[15];
  char state[2];
  char zip[10];
 public:
  void print_name();  // prints name
  Person();           // Constructor
  ~Person();          // Destructor
};
class Student : public Person
{
  char course[7];
  char grade[3];
  char major[4];
 public:
  void print_major();

};
```

In this example, `class Student : public Person` is the declaration of the derived class `Student`. The base class `Person` is shown after the `:`. The *base class access* is declared `public` in this case. Base class access can be `private`, `protected`, or `public`. Base class access relates to the access base class members will have in the derived class. It does *not* relate to the access category of base class members. The rule is that the inherited members will have access category no higher than the base class access. Any base class members of higher access will be converted down to the base class access in the derived class. Thus, a base class access of `public` results in the same access in the derived class as in the base class for all inherited members.

8.1 Virtual Functions

When a member function in a base class is declared to be `virtual`, its definition may be overridden by an identically declared virtual function in a derived class. It is not required to override base class virtual functions. The type of the virtual function is declared in the base class and cannot be redeclared in the derived classes. The virtual function must be defined for the base class and will be the "default" function for a derived class which does not redefine it.

Section 8.2 gives an example of virtual functions used with pointers to class instances.

8.2 Pointers, Inheritance, and Virtual Functions

A pointer to an instance of `Student` can be assigned to a variable of type pointer to `Person` without an explicit type conversion. In general, a pointer to an instance of a class derived from a public base class can be assigned to a variable of the base class type. An instance of a derived class can be treated as an instance of its base class when access is through pointers.

The following example shows a use of virtual functions with pointer-accessed class instances.

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

class Person
{
 protected:
  char name[50];
 public:
  Person() { strcpy(name,"noname"); };
  Person(char *n) { strcpy(name,n); };
  virtual void print();
};
class Student : public Person
{
  char course[8];
 public:
  Student(char *c, char *n) { strcpy(course,c); strcpy(name,n);};
  void print();
};
class Professor : public Person
{
  char dept[5];
 public:
  Professor(char *d, char *n){ strcpy(dept,d); strcpy(name,n);};
  void print();
};
class Staff : public Person
{
  char job[25];
 public:
  Staff(char *j, char *n) { strcpy(job,j); strcpy(name,n); };
};
```

```
main()
{
  Person person("Adams");
  Student student("CMSC123","Jones");
  Professor prof("CMSC","Smith");
  Staff staff("Librarian","Green");
  Person *people[4];

  people[0] = &person;
  people[1] = &student;
  people[2] = &prof;
  people[3] = &staff;

  for (int i = 0; i < 4; i++)
    people[i]->print();
}
void Person::print()
{
  cout << "Name: " << name << endl;
}
void Student::print()
{
 Person::print();
 cout <<  "  Course: " << course << endl;
}
void Professor::print()
{
 Person::print();
 cout <<  "  Dept: " << dept << endl;
}
```

The output from this program shows that each instance of a Person class or class derived from Person has been printed using its own `print` function.

```
Name: Adams
Name: Jones
  Course: CMSC123
Name: Smith
  Dept: CMSC
Name: Green
```

9 Iterators

Section 7.2.6 showed a primitive way to access the elements of a class instance in sequence. The methods shown in the following sections are much less invasive, and allow multiple actions.

Section 9.1 sets up a friend class `FooIterator` which controls the iteration. `FooIterator` class instances keep track of the next data element in the iteration as well as which instance is being iterated. The function call operator of `FooIterator` takes a `Foo` instance as an argument, allowing multiple simultaneous iterations.

Section 9.2 uses a `FooIterator` class also, but does not require it to be a friend of the `Foo` class. The penalty for this is that `Foo` must provide access functions to its non-public members and to the instance itself. Since such access functions are not unusual, it can be argued that this method is less invasive than the "friend-class" method.

9.1 Iteration Using A Friend Class

In the following example, two iterators `iter1` and `iter2` are declared. They iterate the same `Foo` instance `x`. The example shows that the iterations are independent. Note that the iterators are made to return `int *` to be able to distinguish end of list from the integer 0.

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

class Foo
{
  int * data;
  int size;
 public:
  ~Foo() { delete data; };
  Foo(int);
  friend class FooIterator;
};
class FooIterator
{
  Foo *foo_instance;
  int pos;    // current position in data
 public:
  FooIterator (Foo & f) { foo_instance = &f; pos = 0; }
  int * operator () ()
    { return (pos < foo_instance->size) ?
                          &foo_instance->data[pos++] : 0; }
};
```

```
main()
{
  Foo x(16);
  FooIterator iter1(x);
  FooIterator iter2(x);
  int * p, *q;

  while ( (p=iter1()) && (q = iter2()) )
    {
      cout << *p << " " << *q << endl;
      q = iter2();
    }
}
Foo::Foo(int m)
{
  size =  m;
  data = new int[size];
  for (int i=0; i<size; i++)
    data[i] = i;

}
```

The output from the above program is

```
0 0
1 2
2 4
3 6
4 8
5 10
6 12
7 14
```

9.2 Iteration Without A Friend Class

The following example shows an iterator done without a friend class, but with access functions in the `Foo` class. Use of the `!` operator allows the iteration functions to return `int` rather than `int *`. Use of the increment operator `++` separates incrementing from accessing. Finally, the `init` function allows iterators to be reset at any time. In this example, the iterator `iter2` is reset during the looping.

The `Foo` constructor `Foo::Foo(Foo &)` is needed during construction of `FooIterator` instances.

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

class Foo
{
  int * data;
  int size;
 public:
  Foo * get_this() {return this;}
  int get_size() { return size; }
  int *get_data() { return data; }
  ~Foo() { delete data; };
  Foo(int);
  Foo(Foo &);  // needed for constructing a FooIterator
};
class FooIterator
{
  Foo * foo_instance;
  int pos;
 public:
  FooIterator (Foo &f) { foo_instance = f.get_this(); pos = 0;  }
  int operator () ()
    { return (pos < foo_instance->get_size()) ?
        (foo_instance->get_data())[pos] : 0; }

  int operator ! () { return (pos < foo_instance->get_size()); }
  int operator ++ () {pos++; return (foo_instance->get_data())[pos]; }
  int init () { pos = 0; return (foo_instance->get_data())[pos]; };
};
main()
{
  Foo x(16);
  FooIterator iter1(x);
  FooIterator iter2(x);
  int p,q;

  for (p = iter1(), q = iter2(); !iter1; p = iter1++, q = iter2++)
    {
      if (p == 8)
        q = iter2.init();
      cout << p << " " << q << endl;
    }
}
```

```
Foo::Foo(int m)
{
  size =  m;
  data = new int[size];
  for (int i=0; i<size; i++)
    data[i] = i;

}
Foo::Foo(Foo & f)
{
  size = f.size;
  data = new int [size];
  for (int i=0; i<size; i++)
    data[i] = f.data[i];
}
```

The output from the above example is:

```
0 0
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 0
9 1
10 2
11 3
12 4
13 5
14 6
15 7
```

10 Templates

In most of the examples in this guide (for instance, in Section 7.2.5) the `Foo` class has a `data` member of type `int*`. `Foo` can be generalized to be a *template class* for which `data` can be of any type. The following example, similar to the example of Section 7.2.5, shows such a generalization. The `Foo` class now becomes a template of class `T`, where `T` is some built-in or user-defined type. `T` can be used in `Foo` anywhere any other type-declarator can be used. Compare the declaration `T *data` here to `int *data` in Section 7.2.5. Also note the new syntax for function definitions.

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

template<class T> class Foo
{
  T *data;
  int size;
 public:
  Foo(const int);
  int get_size();
  T & operator[](const int);
  void print();
};
main()
{
  Foo<int> x(5);
  Foo<float> y(6);
  Foo<char *> z(3);


  for (int i=0; i<x.get_size(); i++)
    x[i] = i;
  for (i=0; i<y.get_size(); i++)
    y[i] = float(i);
  z[0] = "abc"; z[1] = "def"; z[2] = "ghi";
  x.print();
  y.print();
  z.print();
}
template<class T> Foo<T>::Foo(const int initval)
{
  size=initval;
  data=new T[size];
}

template<class T> int Foo<T>::get_size() {return size;}
```

```
template<class T> T& Foo<T>::operator[](const int index)
{
  if ( (index < 0) || (index >= size) )
    {
      cerr << "Index [" << index << "] out of range [0.." << size-1;
      cerr << "]" << endl;
      exit(1);  // bail out
    }
  return data[index];
}
template<class T> void Foo<T>::print()
{
  for (int i = 0; i < size; i++)
     cout << data[i] << " ";
  cout << endl;
}
```

The output from the above program is:

```
0 1 2 3 4
0 1 2 3 4 5
abc def ghi
```

11 Suggested References

1. Stanley B. Lippman
   C++ Primer
   Addison-Wesley, 1989

2. Greg Perry
   Moving From C to C++
   Sams Publishing (Prentice-Hall), 1990

3. Bjorne Stroustrup
   The C++ Programming Language 2nd edition
   Addison-Wesley, 1992