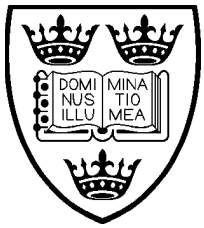
Oxford University Computing Services



Programming in C++

Languages l9.3/1

Typographical Conventions

Listed below are the typographical conventions used in this guide.

– Example C++ code and commands to be typed by the user are in non-bold characters in `typewriter` font.

– Items where the user has to supply a name or number are given in lower-case *`italic`* characters in typewriter font.

– Sections marked with a ‡ describe features that are also available in ANSI C.

Acknowledgements

I would like to thank Francis Cameron and Adrian Cox, who have both acted as demonstrators for the OUCS C++ course, for their many suggestions and ideas on improving these notes and the OUCS C++ course.

Contents

References

[1] The C++ Programming Language, Bjarne Stroustrup, Addison Wesley, Second Edition 1992

Author: Stephen Gough

Revision History:

l9.3/1 April 1994 Original publication

C makes it easy for you to shoot yourself in the foot. C++ makes it harder, but when you do it blows your whole leg away!

Bjarne Stroustrup

1 C++ as a Better C

Features marked with a ‡ are available in ANSI C.

1.1 Comment to End of Line

```
a += 6;         // this is a comment
```

The new style comments are useful as it is possible to comment out code containing comments, e.g.

```
// a += 6       // this is a comment
// b += 7;
```

With C style comments (which are still available in C++) problems occur:

```
/*
a += 6;         /* this is a comment */
b += 7;
*/
```

The close comment symbol in the original comment ends the new comment, leaving code uncommented and a `*/` symbol which will cause the compiler distress.

1.2 Enumeration and Structure Names

The name of an enumeration or structure is a type name. The keyword `enum` or `struct` is not required in subsequent use of the type, e.g. in variable declarations or in other type constructors.

```
enum primary_colour  { red, green, blue };
                     // primary_colour is a enumerated type

primary_colour       colour;
                     // colour can only be red,
                     // green or blue
enum primary_colour  colour2;
                     // old fashioned declaration

struct complex       { double re, im; } fred;
                     // complex is a structured type
                     // fred is a variable of type complex

complex harry;       // harry is a variable of
                     //   type complex
struct complex jim;  // old fashioned declaration
```

1.3 Declarations within Blocks

C++ permits declarations of variables and types anywhere within a block, not just at the start of a block. Variables can be declared closer to their point of use, reducing the risk of forgetting the type or misspelling the name. The ability to declare and initialise variables after statements also means that more complication structures are not initialised twice. For example, an array will be initialised to contain zeros without an initialiser, if the data is not known until later, then the array will be filled twice. It is more efficient to declare and initialise once the data required becomes available.

1.4 Scope Qualifier

```
#include <stdio.h>

int   i = 0;

int main()
{
   for (int i = 0; i < 10; i++)
      printf("%5d%5d\n", i, ::i);
   return 0;
}
```

The program produces the following output:

```
    0    0
    1    0
    2    0
       .
       .
    9    0
```

`i` refers to the local variable, `::i` refers to the global variable.

1.5 *const* ‡

The keyword `const` can be used to freeze the value of an object. `const` objects will be initialised (as their value cannot otherwise be set!). `const` can also be used as a type qualifier on function parameters to prevent accidental modification of the parameter within the function.

```
void fred(const int x);

int main()
{
   const int   i = 15;
   int         j = 16;

   i = 5;         // illegal

   fred(j);

   return 0;
}

void fred(const int x)
{
   x = 6;         // illegal
}
```

When applying `const` to a pointer we can indicate that the pointer should be unmodifiable, or that the data pointed to should remain constant. The alternatives are given below:

```
const char        *ptr1 = "Hi";  // data pointed to is const
char *const       ptr2 = "Hi";   // pointer is constant
const char *const ptr3 = "Hi";   // both pointer and data
                                 //   are constants
```

`const` objects can be used in array declarations, they cannot in C. The following is legal C++, but illegal C:

```
const int   size = 10;
float       vals[size];
```

1.6 Explicit Type Conversion

Used to convert a value of one type to a value of another. Can be used as a replacement for old style cast operators.

```
int main()
{
   int      i;
   float    f;

   i = (float) f * 6;
   // old style cast to float
   // - does cast refer to (f * 6) or just to f?

   i = float (f * 6);
   // new style cast to float
   // - clear what is being converted

   return 0;
}
```

Type conversions of this form can only be used when the type has a simple single name, that is

```
str = char * (ptr);
```

is not legal, but if a new type *name* is created with `typedef`, then the conversion will be legal C++.

```
typedef char   *string;

str = string(ptr);
```

1.7 Function Definitions and Declarations ‡

C++ uses the same format of function definition as defined in the ANSI C standard. The types of the parameters are specified within the round brackets following the function name. C++ function declarations are used to provide checking of the return type and parameters for a function that has not yet been defined. C++ *declarations* are the same as ANSI C *prototypes*.

The advantage of using function declarations and the new style definitions is that C++ will check that the type of the actual parameters are sensible — not necessarily the same. Either the types must be the same, or all the applicable standard and user defined type conversions are tried to see if a match can be found.

Some C++ implementations will *not* accept the old K&R declaration style at all.

```
double minimum(double a, double b)
// C++ and ANSI C function definition
{
   return a < b ? a : b;
}

double maximum(a, b)
// old fashioned K&R format
double   a, b;
{
   return a > b ? a : b;
}

int main()
{
   minimum(1, 2);
      // correct usage minimum(1.0, 2.0);
      // C++ will convert 1 to 1.0 and 2 to 2.0

   minimum("hi", "there");
      // non-sensical, C++ will complain

   maximum(1, 2);
      // call traditional C function
      // no errors - wrong answer

   maximum("hi", "there");
      // again no complaints - strange results
}
```

Note that an ANSI C function declaration (where information about the parameters is omitted) is interpreted by C++ as a declaration of a function that takes *no* parameters. For example, the following declaration states that `fred` returns an `int` and can take *any* number of parameters of *any* type in ANSI C, whilst in C++ it declares a function that returns `int` and takes *no* parameters in C++.

```
int fred();
```

1.8 Overloading of Functions and Operators

Several different functions can be given the same name. C++ will choose one of the functions at compile time (but see 2.3.4.2 *Virtual Functions*), given the type and number of parameters in the function call.

The following `print` functions print an `int`, a string and an array of `int`:

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

void print(int i)
// print an integer
{
   printf("%d\n", i);
}

void print(char *str)
// print a string
{
   printf("%s\n", str);
}

void print(int a[], int elem)
// print an array of integers
{
   for (int i = 0; i < elem; i++) printf("%d\n", a[i]);
}

int main()
{
   int   i = 6;
   char  *str = "hello";
   int   vals[] = { 1, 2, 3, 4 };

   print(i);         // call print(int)
   print(str);       // call print(char *)
   print(vals,sizeof(vals)/sizeof(int));
                     // call print(int [], int)
   return 0;
}
```

It should be noted that there are better ways of handling printing of user defined types in C++ (namely `operator<<`, see section 1.15 and exercise 2.2.20...(2)).

1.9 Default Values for Function Parameters

Functions can be called with fewer actual parameters than formal parameters. The unspecified parameters are given default values.

```
void fred(int one, float two = 1.23, char three = 'c')
{
}
```

The arguments with default values must be the last parameters in the parameter list.

In this example, `fred` can then be called with 1, 2 or 3 parameters, e.g.

```
fred(6,7.2,'z');
fred(6,7.2);
fred(6);
```

When *declaring* functions that take parameters with default values, subsequent declarations cannot specify the default values already specified in earlier declarations, but they can add new default values. When declaring a function before use, specify the default values in the first declaration, e.g.

```
void fred(int one, float two = 1.23, char three = 'c');

int main()
{
   fred(6, 7.2);
}

void fred(int one, float two, char three)
{
}
```

1.10 Name Mangling

As it is possible to overload functions (e.g. in the example above there are three `print` functions) and as it is possible to use C++ with your existing linker, there must be some mechanism to generate a unique name for each overloaded function. Each function's name, number and type of parameters are combined to produce a unique name. This process is called *name mangling*. Occasionally name mangling is not required, one example is linking with C code. To turn the mangling off, a *linkage specification* can be used when declaring the C routines that will be used:

```
extern "C" void fred(int);
```

`fred(int)` will now no longer be mangled. If the linkage specification is not used, and `fred` is declared as `void fred(int)` an error such as `Undefined: fred_Fi` will be produced when linking. `fred_Fi` is the mangled name of `void fred(int)`

When a group of C functions need to be declared, they can be placed in a linkage specification block:

```
extern "C" {
   int printf(char *fmt ...);
   int scanf(char *fmt ...);
}
```

1.11 Functions with a Variable Number of Parameters ‡

Functions can be written which can take any number of parameters. It is now possible to write your own equivalent of `printf`[1]. To write functions with a variable number of parameters it is necessary to use the macros, `va_start` and `va_end`, which are defined in the header file, `stdarg.h`. Individual unnamed parameters can be obtained with the `va_arg` macro, or the `vprintf` range of print routines can be used to print a list of unnamed parameters.

The first of the following examples uses `vfprintf` to print the unnamed parameters, the second extracts the unnamed parameters one by one using `va_arg`.

```
#include <stdio.h>   // for fprintf
#include <stdlib.h>  // for exit
#include <stdarg.h>  // for variable num of args macros

void error(char *format ...)
{
   va_list  args;

   va_start(args, format);
   // make args point to first unnamed parameter

   fprintf(stderr, "ERROR: ");
   // print start of error message

   vfprintf(stderr, format, args);
   // print all unnamed arguments

   fprintf(stderr, "\n");
   // move onto newline

   va_end(args);
   // tidy up

   exit(1);
   // return to OS with exit status of 1
   }

int main()
{
   int   i = -1;

   error("invalid value %d encountered", i);
   return 0;
}
```

[1] But you shouldn't! Use `operator<<` instead (see section 1.15 and exercise 2.2.20...(2)).

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

void sum(char *message ...)
{
   int      total = 0;
   va_list  args;
   int      arg;

   va_start(args, message);
   while ((arg = va_arg(args, int)) != 0)
      total += arg;
   printf(message, total);
   va_end(args);
}

int main()
{
   sum("The total of 1+2+3+4 is %d\n", 1, 2, 3, 4, 0);
   return 0;
}
```

We must make sure that the function knows which is the last actual parameter, otherwise we may try to remove too many parameters from the stack. In the first example, we used the `vfprintf` function, which is the `printf` function to use in conjunction with a variable number of arguments. Like `printf`, the number of conversion characters must match the number of extra parameters. In the second example a special trailing value is used to indicate the last parameter.

In ANSI C the parameter list is written

```
func(named parameters, ...)
```

That is a comma precedes the ellipsis (`...`).

1.12 Inline Functions

Function definitions that are qualified with the `inline` keyword act like macros, that is, the code is inserted at *each* invocation of the function. However, unlike preprocessor macros, `inline` functions are much safer. Copies of the parameters are still taken and type checking of the parameters is performed. Compilers are at liberty to ignore an `inline` instruction, and many will do so for functions that are too long or complicated (each compiler will have its own idea about what this means!). Also, if the address of the function is ever used (perhaps assigned to a pointer), then the compiler will have to generate a normal function (it is difficult to take the address of a function that does not exist!), though this may be in addition to the insertion of the code wherever the function is called.

```
inline int strlen(char *str)
{
   int   i = 0;

   while (str++ != '\0')
      i++;
   return i;
}
```

1.13 *new* and *delete* Operators

`new` and `delete` are used for dynamic allocation of memory space. They have a major advantage over `malloc` and `free`: constructors and destructors (see 2.2.4) for the objects created or destroyed will be called.

There are two forms of `new` and `delete`, depending on whether a single object or an array of objects is required:

`pf = new float;`	for a single `float`
`pf = new float [ num ];`	for an array of `floats`

You should use the corresponding form of `delete` when destroying the objects created with `new`, i.e.

`delete pf;`	for a single object
`delete [] pf;`	for an array of objects

This will ensure that the destructor is called for each object in the array.

`new` returns `0` (`NULL`) on failure.

The following example program illustrates the use of `new` and `delete`.

```
#include <stdio.h>

int main()
{
   int   num_chars;
   char  *name;

   printf("how many characters in your string? ");
   scanf("%d",&num_chars);

   name = new char [num_chars + 1];
   // allocate enough room for num_chars characters
   // + 1 for the '\0'

   if (name != NULL) {
      // do something with name

      delete [] name;
      // remove name from the heap
   }

   return 0;
}
```

It is possible to set up a function to handle out of memory errors when using `new`; the function `set_new_handler` (declared in `new.h`) is used to register the handler:

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

void newError()
{
   fprintf(stderr,"new failed: out of memory\n");
   exit(1);
}

int main()
{
   char  *ptr;

   set_new_handler(newError);
   ptr = new char [ULONG_MAX];
   return 0;
}
```

This program sets up a handler that prints an error message and aborts the program if the heap is full.

1.14 Void Type ‡

Functions can be declared as returning `void`, meaning they do not return a value.

`void` can also be used in the parameter list of a function. C++ differs from ANSI C on the meaning of a function declaration with a null argument list, e.g. `fred();`, in ANSI C this means that the function can take any number of arguments, in C++ it means that this function does not take *any* parameters, i.e. C++'s `fred();` is equivalent to ANSI C's `fred(void);`. C++ accepts `fred(void);` for compatibility with ANSI C.

The generic pointer type is `void *` as it is in ANSI C, but again there is a difference. C++ will allow the assignment of any pointer to a `void *` variable or parameter, but the reverse always requires a cast. In ANSI C, both assignments are permitted without a cast.

```
void fred();
   // A function that takes no parameters and
   // does not return a value.

void *harry();
   // A function that takes no parameters and
   // returns a pointer to void

void *jim(void *);
   // A function that takes a void * parameter
   // and returns a pointer to void

int main()
{
   int      i;
   int      *ptr2int;
   float    *ptr2float;

   i = fred();
      // illegal, fred does not return a value

   fred(i);
      // illegal, fred takes no parameters

   ptr2int = harry();
      // illegal, cast required for (void *) to (int *)

   ptr2int = (int *)harry();
   ptr2float = (float *)harry();
      // OK, casts used

   ptr2int = (int *)jim(ptr2float);
      // OK, no cast needed for pointer to anything to
//   void *

   return 0;
}
```

1.15 Stream Library

The stream library is used for a common interface to input and output routines across built-in types and your own types.

There are three streams available (`cin`, `cout` and `cerr` for *standard input*, *standard output* and *standard error* streams respectively) and they are used in the following way:

```
#include <iostream.h>

int main()
{
   int      i = 45;
   float    f = 34.678;

   cout << i;         // print i
   cout << f;         // print f
   cout << "\nEnter an integer: ";
                      // print a string
   cin >> i;          // read an integer
   cout << i << f;    // print i and f

   return 0;
}
```

A range of manipulators exist to allow formatting of output. [Remember that if all else fails, `printf` is still there!] Standard manipulators include:

`oct`	use octal
`dec`	use decimal
`hex`	use hexadecimal
`endl`	newline (`'\n'`)
`ends`	end of string (`'\0'`)
`flush`	flush stream
`ws`	skip white space
`setbase(int)`	use base specified
`setfill(int)`	set fill character
`setprecision(int)`	set number of digits printed
`setw(int)`	set field width

Declarations for these manipulators are in `iomanip.h`. For example:

```
cout << setw(8) << 1 << setw(8) << setfill('#') << 2
     << endl;
cout << hex << 123 << ' ' << oct << 123 << endl;
```

prints:

```
        1#######2
7b 173
```

The `setw` manipulator applies only to the next numeric or string value printed, hence the need for two calls to `setw` above. The `setfill` manipulator modifies the state of the stream, and will apply to all subsequent output on that stream.

1.16 References

1.16.1 Pointers vs. References

Pointers in C++ operate in precisely the same way as pointers in C. To declare a pointer, you must state the type of the object that the pointer will point to, e.g. to declare a variable `i` as being a *pointer to* `int`:

```
int   *i;
```

The pointer is then used in one of two ways: by making the pointer point to some already existing data; or by making the pointer point to some currently unused memory and then storing data via the pointer. Examples of both methods are given below:

```
int   i = 10;
int   *ptr2int;

ptr2int = &i;         // make ptr2int point to i
*ptr2int = 20;        // change the value of i

ptr2int = new int;    // ask for enough memory to
                      //   store an integer
cin >> *ptr2int;      // store data in this newly
                      //   allocated memory
```

The `*` and `&` operators are more or less the opposites of each other. `&` is used to generate a pointer, and `*` is used to follow a pointer to find what is at the other end. The opposite nature of the two is shown by the fact that `*&i` is the same as `i` (given the declarations above), but `&*i` is not legal as you are attempting to treat `i` as a pointer.

References are ways of introducing a new name for an existing object. References *must* be initialised to refer to some existing object, and this *cannot* be changed! For example, consider the code below:

```
int   i = 1, j = 2;
int&  k = i;                        // k refers to i, i.e. k
                                    //   is an alias for i

cout << setw(2) << i << j << k;  // prints " 1 2 1"
k = j;                              // assigns 2 to i
cout << setw(2) << i << j << k;  // prints " 2 2 2"
```

Note that the assignment to `k` did not make `k` refer to `j`, instead it modified the value of the object that `k` referred to.

1.16.2 Reference Parameters

References are also sometimes used as parameters to functions. In this case, the initialisation of the reference takes place when the *actual* parameters are copied into the *formal* parameters, e.g.

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

void fred(int i, int& j)
{
   i = 3;            // assign to a local copy of the
                     //   first argument
   j = 4;            // modify variable in caller's scope
}

int main()
{
   int   a = 1, b = 2;

   cout << setw(2) << a << b; // prints "1 2"
   fred(a,b);
   cout << setw(2) << a << b; // prints "1 4"
   return 0;
}
```

Needless to say, references are actually implemented using pointers!

1.16.3 Functions Returning References

Functions can return a reference to an object, but make sure that the reference is not to a local object that will be destroyed when the function terminates.

The following function returns a reference to a character within the string parameter.

```
#include <iostream.h>

char& strchr(char *str, char c)
{
   while (*str != '\0' && *str != c)
      str++;
   return *str;
}

int main()
{
   char  s[] = "hello world";

   strchr(s,'l') = 'z';
   cout << s;
   return 0;
}
```

The program should print `hezlo world`.

1.16.4 References and *const*

`const` references can be initialised with non-`const` objects; the object referred to will not be modifiable through the reference. Plain references *cannot* be initialised with a `const` object, as that would allow a `const` object to be changed.

```
{
   int         i = 1;
   const int   j = 2;
   const int&  k = i;   // OK
   int&        l = j;   // illegal

   k = 3;               // illegal, even though reference is
                        // to i which is not const
```

1.17 Exercises

(1) Write a function that will print an `unsigned int` in any given number base. If the base is omitted then the base should default to 10. To print the number use the following recursive algorithm (*n* is the number to print and *base* is the base):

```
if (n > 0) {
   print(n / base, base);
   cout << n % base;
}
```

Of course this will only work for *base* ≤ 10.

(2) Write a function that will swap its two `double` parameters. Use reference parameters. Test with the following program:

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

int main()
{
   double   x = 1.2, y = 1.3;

   cout << x << ' ' << y << endl;
   swap(x,y);
   cout << x << ' ' << y << endl;
   return 0;
}
```

2 Object Oriented Programming

2.1 Rationale

C++ started life as "C with classes". A class contains the data *and* the operations needed on the data, and so the class defines the interface to the data. Classes permit the definition of *abstract data types* (ADTs) in a similar way to the *packages* of ADA and the *modules* of Modula-2. If the interface to the object is well designed then all the details will be within the class, making it easier to identify code that is accessing the data incorrectly. This is part of way to an *object oriented programming language* (OOPL).

For a programming language to fully support OOP it should also support *inheritance* and *polymorphism*.

inheritance — defining a class in terms of another; adding features to the existing class

polymorphism — applying the same operation to objects of different classes (but each must be in the same family, derived from a common base class), the behaviour will be different, even though the operation is the same

The classic example of inheritance and polymorphism is that of shapes in a graphics library. A base class, `shape`, stores the colour and location, and new classes are derived from the base class inheriting the common features of all shapes. If any shape is asked to draw itself, the behaviour will be different (circles and squares look different) even though they are undergoing the same operation; this is polymorphism.

Much work is being done in the fields of *object oriented analysis* (OOA) and *object oriented design* (OOD), but unfortunately this is beyond the scope of this course (and possibly beyond the capabilities of the author!). However, the following rules of thumb for object oriented programming in C++ from Stroustrup [1] will suffice:

- If you can think of "it" as a separate idea, make it a class (see 2.2).
- If you can think of "it" as a separate entity, make it an object of some class (see 2.2.3).
- If two classes have something significant in common, make that commonality a base class (see 2.3.1).
- If a class is a container of objects, make it a template (see 3).

Deriving a class from another (inheritance) and including a class object within another are often referred to as *is-a* and *has-a* relationships. In C++:

is-a — publicly derived from another class

has-a — implemented in terms of; either has a class object as a member or is privately derived from another class

For example, a Ford Mondeo *is-a* car, and a car *has-a* engine (sorry about the grammar!). Therefore, `engine` could be a member of `car`, and `FordMondeo` could be publicly derived from `car`.

2.2 Classes

2.2.1 The Class Construct

A class definition gives the data *and* the operations that apply to that data. The data can be *private, public* or *protected* (see 2.2.8) enabling *data hiding* and construction of *Abstract Data Types* (ADTs).

An example of a simple class is given below:

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

class string {
   char  *str;

public:
   string()       // default constructor, takes
                  //   no parameters
   {
      str = NULL;
   }

   string(const char *s)
                  // constructor with `char *' argument
   {
      str = new char [strlen(s) + 1];
      strcpy(str,s);
   }

   string(const string& t)
                  // copy constructor, takes a reference
                  //   to another string as the parameter
   {
      str = new char [strlen(t.str) + 1];
      strcpy(str,t.str);
   }

   ~string()      // destructor
   {
      delete [] str;
   }

   void print() const
                  // print a string
   {
      cout << str;
   }
};

int main()
{
   string   s1;   // use default constructor
   string   s2("hello world");
                  // use constructor with `char *' argument
   string   s3(s2);
                  // use copy constructor

   // print the strings
   s1.print();
   s2.print();
   s3.print();

   return 0;
}
```

Note that the class definition is terminated with a semi-colon.

Members of the class (whether data or functions) are assumed to be in the *private* section, unless stated otherwise.

2.2.2 Structures

Structures are similar to classes, but the default is for members to be *public* rather than *private*. Therefore structures can contain both data and functions.

2.2.3 Objects

An *object* is an *instance* of a *class*. For *object* read variable!

Every time we define a variable we get a new object which has all the properties of the associated class. This is the same as with the built-in types of C (`float`, `int` etc.) — they have certain properties (range of values, operations etc), the bonus is that we can make up our own as well!

2.2.4 Constructors and Destructors

Constructor methods are called when an object is created, destructor methods are called when the object is destroyed. For automatic objects this will be on entry and exit from the block in which the object is defined. Constructors are frequently used for memory allocation and initialisation, destructors could be used for deallocation of storage in a data structure.

For a class called `string`, the constructor is called `string` and the destructor is called `~string`.

Constructors cannot return a value, but it is also invalid to declare them as returning `void`.

Several constructors can be specified, as long as they have different parameter lists.
A constructor that takes no parameters is called the *default constructor*, and a constructor that takes an object of the same class as its parameter is called the *copy constructor*.

The copy constructor takes a reference to an object of the same class. So that `const` objects can be used as an initialiser, the parameter should be declared to be `const`.

```
class string {
   char  *str;

public:
   string() {}    // default constructor
   string(const string& s) {}
                  // copy constructor
   ~string() {}   // destructor
```

If a constructor can take parameters, then the parameters can be specified using the following syntax:

```
string   s(parameters);
```

There are also two methods of calling the copy constructor after having first constructed a new temporary object:

```
string   s = string(parameters);
```

or the abbreviated form:

```
string   s = parameter;
```

For example, the following are all valid ways of initialising a `string` using a `char *` initialiser; the first uses a constructor that takes a `char *` parameter, the others use the `char *` constructor to create a new object, and then use the copy constructor to initialise the object:

```
string   s1("one");
string   s2 = string("two");
string   s3 = "three";
```

The copy constructor is used for passing parameters by value; constructors are also needed when returning values from functions.

```
string search(const string s, char c)
{
   return string(strchr(s.str,c));
}
```

[For the above function to work it would have to be declared to be a `friend` of the `string` class; see 2.2.14]

When the `search` function is called, the argument is copied and a new `string created` using the copy constructor to initialise it. When the `string` value is returned from `search`, again a new `string` is created, but this time the constructor used is the one that takes a `char *` parameter.

2.2.5 Member Functions

Member functions are functions that are defined or declared within the body of a class definition. They form part of the class description. Member functions are normally used to provide operations on the class data. As they form part of the class definition, each instance (variable) of the class will have associated with it the operations defined within the class. By making the functions that modify or read the data contained in the class *member functions* we know that any code that sets or reads the data incorrectly *must* be within the class itself. This helps debugging and makes it easier to produce *abstract data types*.

Member functions are also referred to as class *methods*.

If the code for a member function is defined within the class, then that member function will be treated as an inline function (see 1.12); more normally the member function is declared within the class and defined elsewhere. The scope resolution operator, `::`, is used to specify the class to which the function belongs.

An alternative way of defining the `print` method from the `string` class shown above would be:

```
class string {
  private:
   char  *str;

  public:

    constructors and destructors

    // declare the print method
    void print() const;
};

// define the print method
void string::print() const
{
    cout << str;
}
```

2.2.6 Operators as Member Functions

It is possible to define operators for classes. These operators can be infix binary operators or unary operators. It is only possible to define operators that use an existing C++ operator symbol, and the precedence of the symbol cannot be changed. There are some operators that cannot be defined, these are:

```
.     .*    ::    ?:  sizeof  #     ##
```

Unary operators are defined as a member function taking no parameters, binary operators are defined as member functions that take one parameter — the object on the right hand side of the operator.

When defining an assignment operator (`operator=`) always declare it to take a *reference* to the object on the right hand side of the assignment operator and return a *reference* to the object on the left of the operator; this will permit assignments to be chained (`a = b = c;`). Also declare the parameter (the object on the right of the operator) to be `const` to prevent accidental modification and also to permit `const` objects to be used on the right hand side of the assignment operator.

```
class string {
   char  *str;

public:

   string& operator= (const string& rhs)
   {
   // delete the current string
   delete [] str;

   // allocate enough room for the new one
   str = new char [ strlen(rhs.str) + 1 ];

   // copy in the string
   strcpy(str, rhs.str);

   return *this;  // see 2.2.12
   }
};
```

The following shows a concatenation operator for the `string` class:

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

class string {
  private:
   char  *str;

  public:

   constructors and destructors as before

   string operator+ (const string& rhs) const
   {
      char  *s;

      // create a new C style string holding
      // the concatenated text
      s = new char [strlen(str)+strlen(rhs.str)+1];
      strcpy(s,str);
      strcat(s,rhs.str);

      // create a new `string', initialising it with the C
      // style string
      string   newstring(s);

      // delete the C style string, no longer needed
      delete [] s;

      // return a copy of the newly created string
      return newstring;
   }

   void print() const
   {
      cout << str;
   }
};
```

This operator can then be used in two different ways (assuming that an assignment operator has also been defined; see exercise 2.2.9...(2)):

```
int main()
{
   string   s1("hello");
   string   s2("world");
   string   s3,s4;

   s3 = s1.operator+(s2);
   s4 = s1 + s2;

   return 0;
}
```

cfront 3 (see 6) introduced a mechanism for defining both prefix and postfix `++` and `--` operators, until then only postfix was possible.

```
class X {
   X& operator++() { }     // prefix
   X& operator++(int) { }  // postfix
};
```

The `int` argument of the postfix operator is a dummy parameter used only to distinguish the two operators.

2.2.7 Default Member Functions

All classes get the following three default member functions:

`X::X()`	default constructor
`X::X(const X&)`	copy constructor (bitwise copy)
`X::operator=(const X&)`	assignment of another object of same class (bitwise copy)

The type of the argument to the copy constructor is `const X&` if either there are no object members or all object members of X have copy constructors that accept `const` arguments, otherwise the type is `X&` and initialisation by a `const` object is illegal. The same is true of the assignment operator.

The copy constructor and assignment operator perform a bitwise copy member by member into the new object.

2.2.8 Access Specifiers

Access to class members is controlled by the access specifiers `private`, `protected` and `public`.

private	can only be used by member functions
protected	can only be used by member functions and by member functions of derived classes (see 2.3.1)
public	can be used by anyone! Functions and data that you wish to be exported/published should be placed in the public section

Multiple private (or protected or public) sections are permitted. In classes any unmarked sections are deemed to be private, in structures they are assumed to be public.

```
class X {
   char  *s1;  // private
public:
   char  *s2;  // public
private:
   char  *s3;  // private
};
```

2.2.9 Exercises

(1) Using enumerated types define a class `date`, which can be initialised by writing:

```
date  d(Mon, 13, Nov, 1989);
```

Write methods for: constructor (used as above), copy constructor and printing.

The enumerated types for the day and month will need to be visible outside of the class to make the initialisation above possible, therefore make the two enumerated types global.

(2) Add an append operator (`operator+=`) to the string class in the example file `string.cpp`.

2.2.10 Methods and Messages

In the jargon of OOP the functions defined in a class are called *methods*. Methods are invoked by sending an object a *message*.

In the string example given above we used the statement:

```
s4 = s1 + s2;
```

This actually involves sending the following messages to the following objects

Object	**Message**
`s1`	`+ s2`
`s4`	= *result of above*

More normally in C++ (the above example uses infix operators) the syntax for passing a message to an object will be

```
object.method(other_info,...);
```

e.g.

```
x.times(3);    // multiply x by 3
```

The full form of the statement `s4 = s1 + s2;` is:

```
s4.operator=(s1.operator+(s2));
```

2.2.11 Order of Evaluation

The order of evaluation in expressions is determined by the precedence of the operators used (just as it is in C). Note that, whilst a programmer can define functions that act as operators, they cannot change the precedence or associativity of that operator. In the following example, evaluation will proceed from left to right:

```
z = a + b + c;
```

A *new* temporary object will be created to hold the value of `a + b`, this new object will then receive the message to add itself to `c`. Once the value has been assigned to the object `z`, the temporary variable will be destroyed.

2.2.12 The Implicit Argument – *this*

When an object receives a message we can use the reserved word `this` to gain access to the object itself. `this` is a pointer to the object, so `*this` is the object itself. For example, the following function returns a reference to the object that received the message:

```
complex& operator=(const complex& rhs)
               // assignment
{
   real = rhs.real;     // assign to private data
   imag = rhs.imag;     // assign to private data
   return *this;        // return the object
}
```

`this` can also be used in `operator=` to prevent unnecessary, or even destructive[2], assignment when the same object is on both sides of the assignment operator, e.g.

```
complex  a;
complex& b = a;   // b refers to a

a = a;
a = b;
```

Both of the assignments above involve an assignment back to original object. This can be prevented by comparing pointers to the two objects, if they are the same, then the objects are the same:

```
complex& operator=(const complex& rhs)
{
   if (&rhs != this) {  // compare pointers to rhs
                        // and ourselves
      assign to private data
   }
   return *this;
}
```

2.2.13 *const* members

Member functions that have the qualifier `const` after their parameter list are not able to modify the data stored within the class unless the `const` is "cast away". This is easier to explain with an example:

[2] When assigning to a `string` object (section 2.2.6) the code deleted the current string, and then made a copy of the string on the right hand side of the operator. If the same object appears on both sides of the operator, then the string will be destroyed without a copy being kept!

```
class complex {
   double   real, imag;

public:

   void tom() const
   {
      real = 1.0;    // illegal
   }

   void dick() const
   {
      ((complex *)this)->real = 1.0;
                     // OK
   }

   void harry()
   {
      real = 1.0;    // OK
   }
};
```

`const` member functions can be called for both `const` and non-`const` objects, non-`const` members can *only* be called for non-`const` objects. In fact, this can be used to call different functions for `const` and non-`const` objects, e.g.

```
class string {
   char  *str;

public:

   const char& operator[](int index) const
   // operator[] for const objects
   // returns a ref to a constant char
   {
      return str[index];
   }

   char& operator[](int index)
   // operator[] for non-const objects
   // returns a ref to a char
   {
      return str[index];
   }
};

int main()
{
   const string   s1("hello");
   string         s2("world");

   s1[0] = 'z';   // calls first operator[], and is illegal
                  // as we're trying to modify through a
                  // const reference
   s2[0] = 'z';   // calls second operator[], and sets the
                  // first character of s2 to 'z'

   return 0;
}
```

The attempt to modify the first character of `s1` will fail, as a reference to a constant character is returned.

2.2.14 Friends

Functions and other classes can be declared to be *friends* of a class, they can then access private data and private functions of the class specified.

2.2.15 Operators — Member Function vs Global Function

Binary operators are often defined as a global function accepting two parameters. [If the function needs to access private members of the class, then it will also need to be declared to be a *friend* of the class.] This is done in cases where it is not possible to modify the source code for the class that the function would normally need to be a member of. A good example of this is the definition of a stream output operator. It is not possible to modify the code for the stream classes, so instead we overload the operator `<<` further by defining a function called `operator<<` with two parameters, and make it a `friend` of the class we are working on. See exercise 2.2.20...(2) below.

binary operator	
member	one parameter
global	two parameters
unary operator	
member	no parameters
global	one parameter

As member functions have one implied argument (the object receiving the message), member functions used as infix operators have one less parameter than a non-member function that is declared to be a `friend`. For example:

```
class complex {
   double   real, imag;

public:              // everything below is public

   complex(double re = 0.0, double im = 0.0)
   {
      real = re; imag = im;
   }

// method declarations

   complex operator+(const complex&) const;
   friend complex operator-(const complex&, const complex&);
   complex operator=(const complex&);
};

// complex methods

complex complex::operator+(const complex& arg) const
// member function
{
   return complex(real + arg.real, imag + arg.imag);
}

complex operator-(const complex& arg1, const complex& arg2)
// global friend function
{
   return complex(arg1.real-arg2.real, arg1.imag-arg2.imag);
}

complex& complex::operator=(const complex& rvalue)
// member function
{
   real = rvalue.real;
   imag = rvalue.imag;
   return *this;
}

int main()
{
   complex  a,b,c;

   a = b + c;        // uses member function operator+
   a = b - c;        // uses a global function
   return 0;
}
```

Global functions can also be used to reduce the number of member functions needed, as the compiler will use constructors to cast parameters to the correct type if a declaration for the function exists. For example, consider dealing with the following expression:

```
a + b
```

where `a` and `b` could be either of integral or floating point type or of our own type `complex`. Using member functions, then we would not be able to deal with the situation where `a` is not complex, the object receiving the message would need to be `complex`. This restriction does not apply to global functions. Also using an ordinary function we need only write one

function to cope with all the above circumstances, one where both the parameters are of type `complex`, that is:

```
complex operator+ (const complex& arg1, const complex& arg2);
```

If we have a `complex` constructor that can form a `complex` from an integer or floating point value (and we do!) then values of these types will be promoted to type complex (using the constructor) automatically.

2.2.16 *static* Class Data

For `static` data only one copy per class instead of one copy per object will be stored. `static` data is initialised outside of the class.

The following code shows a class where arrays of values are shared between all instances of the class.

```
enum day    { Mon, Tue, Wed, Thu, Fri, Sat, Sun };
enum month  { Jan, Feb, Mar, Apr, May, Jun,
              Jul, Aug, Sep, Oct, Nov, Dec };

class date {
   static char *days[7];
   static char *months[12];
   static int  dayspermonth[12];

public:

   // declaration of constructor
   date (day aDay = Mon, int aDate = 1,
      month aMonth = Jan, int aYear = 1900);
};

char *date::days[] =       {
                           "Monday",
                           "Tuesday",
                           "Wednesday",
                           "Thursday",
                           "Friday",
                           "Saturday",
                           "Sunday"
                           };

char *date::months[] =     {
                           "January",
                           "February",
                           "March",
                           "April",
                           "May",
                           "June",
                           "July",
                           "August",
                           "September",
                           "October",
                           "November",
                           "December"
                           };

int date::dayspermonth[] = { 31,28,31,30,31,30,
                             31,31,30,31,30,31 };
```

2.2.17 Conversions

Classes can also have member functions that convert an object into a value of another type. For example, `complex` might have an `operator double()` conversion method that returns the real part of the complex number. Such an operator will be used wherever an explicit cast operator is used, but also when needed to match with function declarations.

```
#include <math.h>

class complex {
   double   real,imag;

public:

   ....        // other members as above

   operator double()
   {
      return real;
   }
};

void main()
{
   complex  c(1.0,2.0);
   double   d;

   d = double(c); // assign real part to d
   d = exp(c);    // calc e raised to power of real part
                  //   c has been converted to double
}
```

2.2.18 Class Objects as Members

Classes can have members which are objects of another class, e.g.

```
class fred {
   complex  c;

public:
   // constructor, declaration only
   fred(double re = 0.0, double im = 0.0);

   // destructor
   ~fred() {}

   // print
   void print() {}
};
```

When initialising an object member, there are two methods:

- assignment
- member initialisation

```
// use assignment to initialise the complex member
fred::fred(double re, double im)
{
   c = complex(re,im);
}

// use member initialisation to initialise
// the complex number
fred::fred(double re, double im) : c(re,im) { }
```

[Remember that these two constructors are alternatives, they *cannot* coexist.]

With the first method:
- the complex member is initialised with the default constructor of class `complex`
- a temporary `complex` object is created and initialised with `re` and `im`
- the complex assignment operator (`operator=`) is called

With the second method:
- the complex member is initialised using the `complex` constructor that takes two `double` arguments

Therefore the second method is usually preferred, it is much more efficient.

Note that the second method is the only method that can be used with `const` members.

2.2.19 An Example Class — Complex Numbers

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

// complex declaration

class complex {            // complex number
                           // class description

   double    real, imag;   // this is private

public:                     // everything below is public

   complex(double re = 0.0, double im = 0.0)
       // Constructor #1, definition function
       // - used to allocate memory for variables of
       //   type complex
       // - uses parameters with default values
       // - is an inline function
   {
      real = re; imag = im;
   }

   complex(const complex& othercomplex)
       // Constructor #2, copy initialiser
       // - takes a reference to another complex number as
       //   its argument
       // - inline function
   {
      real = othercomplex.real; imag = othercomplex.imag;
   }

   operator double() const
      // conversion into a double
   {
      return real;
   }

   // method declarations

   complex& operator=(const complex&);
       // assignment
       // - takes a reference to a complex number
       // - returns a reference to a complex number

   // and finally, our friends

   friend complex operator+(const complex&, const complex&);
       // addition
       // - takes two references to complex numbers
       // - returns a complex number

   friend complex operator-(const complex&, const complex&);
       // subtraction
       // - takes two references to complex numbers
       // - returns a complex number

}; // class description ends
```

continued...

...continued

```
// complex methods

complex& complex::operator=(const complex& rhs)
// assignment
{
   real = rhs.real;
   imag = rhs.imag;
   return *this;
}

// friends

complex operator+(const complex& lhs, const complex& rhs)
// addition
{
   return complex(lhs.real + rhs.real, lhs.imag + rhs.imag);
}

complex operator-(const complex& lhs, const complex& rhs)
// subtraction
{
   return complex(lhs.real - rhs.real, lhs.imag - rhs.imag);
}


int main()
{
   complex  a;
            // declaration & definition of a complex number
            // uses definition function (constructor #1)
            // will be initialised to 0 + i0

   complex  x(1.0,2.0), y(20.0), z(11.0,12.0);
            // use definition function

   complex  u(x);
            // uses copy initialiser function

   complex  v = x;
            // uses copy initialiser function;
            // it is identical to definition of u above

   complex  w = x + y - z;
            // uses copy initialiser function, after
            // producing a temporary unnamed variable
            // holding the result of x + y - z

   a = x + y - z;
            // use overloaded operators
            // uses +, - & = functions above

   cout << double(a) << endl;
            // use the double conversion operator

   cout << exp(a) << endl;
            // print e to the power of the real
            // part of a; uses the double conversion
            // operator
   return 0;
}
```

2.2.20 Exercises

(1) The *modulus* of a complex number z, where z is $x+iy$, is written as $|z|$ and is given by

$$|z|=\sqrt{x^2+y^2}$$

Make a copy of the example file `complex.cpp` in your own directory, and add a `mod` method to the `complex` class that returns the modulus of the complex number as a `double` value.

(2) Add a print routine to the `complex` class. Use a global friend function; add the following declaration to the class definition:

```
friend ostream& operator<< (ostream& stream, complex& c);
```

and then use the following outline for the function itself; which will appear *outside* the class definition:

```
ostream& operator<< (ostream& stream, complex& c)
{
}
```

(3) The file `stack.cpp` contains the bare bones of a stack class definition. Make a copy of the file and then complete the definition.

(4) The files `list.h` and `list.cpp` contain a class which implements a linked list of objects of any class derived from `Object` in the Borland class library. Use them to construct a linked list of names. Use Borland's `String` class to store the strings. The example file `uselist.cpp` gives a solution to this exercise.

The `list.cpp` program is loosely based on an example program in *An Introduction To Object Oriented Programming And C++* by Wiener & Pinson.

2.3 Inheritance and Polymorphism

2.3.1 Derived Classes

It is possible to derive classes from existing ones. The derived class *inherits* all the features of the existing *base* class, and can then add new features.

The main programming benefit from deriving classes are *code reusability* and *ease of maintenance*.

Reusability — A linked list base class can be used in many different situations, all of which require slight changes to the feature list. For example, we might use the linked list class to derive classes that handle stacks, queues etc. We are automatically reusing the code from the base class in all of the derived classes.

Maintenance — After we have coded and debugged a class, unless there are extenuating circumstances we would normally leave the class alone. If we require a class that behaves slightly differently then derive a new class from the old one. This allows us, at last, to follow the adage "*if it ain't broke don't fix it*"!

Inheritance also pays huge dividends in the area of libraries. Normally we do not have the source code for the library functions. If we want to alter a class to fit our own needs we can derive a class from the library class adding extra features, or overriding functions as necessary.

2.3.2 Deriving a New Class

There are three ways of deriving a new class from a base class:

```
class derived : public base {};
class derived : private base {};
class derived : protected base {};
```

The difference between these declarations are:

`public base`	*public* members of base become *public* members of derived, *protected* members of base become *protected* members of derived
`private base`	*public* and *protected* members of base become *private* members of derived
`protected base`	*public* and *protected* members of base become *protected* members of derived

If the specifier `public`, `private` or `protected` is omitted then `private` is assumed for classes and `public` is assumed for structures.

There is *no* way for the derived class to get its hands on the private data or member functions of the base class.

2.3.3 Derived Classes; Constructors and Destructors

Constructors and destructors are *not* inherited (see Stroustrup r.12.1 and r.12.4). However, if a base class has a constructor then that constructor will be called *before* the constructor of the derived class. The same holds for destructors, except that the base class destructor is called after that of the derived class. [See Stroustrup r.12.4]

Base classes behave (in this respect) like members of the derived class. The constructor of the base class is called, or if it requires parameters these can be specified in the following way:

```
class base {
   int   I;

public:
   base(int i) { I = i; }
};

class derived : public base {
   int   J;

public:
   derived(int i, int j) : base(i) { J = j; }
};
```

The first argument to the constructor of `derived` is passed on to the constructor of `base`.

For example, a 3-D point may be derived from a 2-D point. The 3-D point will inherit the x and y values of the 2-D point and will add a z coordinate. The constructor for a 3-D point can use the 2-D constructor, and then concentrate on what is different between a 3-D point and a 2-D one.

```
class 3Dpoint : public 2Dpoint {
   double   z;

public:

   // constructor
   3Dpoint(double X = 0.0, double Y = 0.0, double Z = 0.0)
         : 2Dpoint(X,Y)
   {
      z = Z;
   }
};
```

2.3.4 Virtual Functions and Polymorphism

2.3.4.1 Polymorphism

If the same function can be applied to objects of different types then that function is *polymorphic*. Whilst polymorphism could be implemented in C using a function that takes a pointer to the object to use, in C++ the process is usually implemented by having member functions of the same name in several classes.

2.3.4.2 Virtual Functions

Virtual functions can be used when a member function is called through a pointer of the type *pointer to a base class*. The function called will be the function of that name in the *derived class*, even though the pointer is declared as a pointer to the base class. The following example illustrates the difference between a normal member function and one that is declared to be *virtual*.

```
#include <iostream.h>

class shape {
public:
   virtual void draw() { cout << "draw shape\n"; }
   void paint() { cout << "paint shape\n"; }
};

class square : public shape {
public:
   void draw() { cout << "draw square\n"; }
   void paint() { cout << "paint square\n"; }
};

class circle : public shape {
public:
   void draw() { cout << "draw circle\n"; }
   void paint() { cout << "paint circle\n"; }
};

int main()
{
   circle   c;
   square   s;
   shape    *ptr;

   ptr = &c;      // pointer to derived is compatible
                  //   with pointer to base
   ptr->draw();   // which draw method do we get?
                  //   ans: circle's draw
   ptr->paint();  // which paint method do we get?
                  //   ans: shape's paint

   ptr = &s;
   ptr->draw();   // call square's draw
   ptr->paint();  // call shapes's paint


   shape *shapes[10];
                  // declare an array of 10
                  //  pointers to shape

   shapes[0] = new square;
   shapes[1] = new circle;
                  // set shapes[0] to point to a square
                  // and shapes[1] to point to a circle

   for (int i = 0; i < 2; i++)
      shapes[i]->draw();
                  // call the draw method of the appropriate
                  //   derived class

   return 0;
}
```

Virtual functions require *late binding*, we don't know until run time which method will be called. Until now, even with function and operator overloading the decision over which function or method to call is made at compile time, there is no run-time overhead without virtual functions.

2.3.4.3 Pure Virtual Functions and Abstract Base Classes

Have no body e.g.

```
virtual void draw() = 0;
```

Derived classes *must* redefine this method, overriding this function.

Classes containing one or more pure virtual functions are called *abstract base classes*. It is not possible to create objects of an abstract base class, instead they are used as base classes from which to derive new classes. For example, using the example classes in 2.3.4.2, `shape` could be turned into an abstract base class. This would prevent declaration of `shape` objects. Pointers to abstract base classes (in this case `shape`) are permitted, so the rest of the code in the shape program would not have to be changed.

2.3.4.4 Virtual Destructors

If you are likely to access derived class objects through a pointer to the base class (as in the shape example in 2.3.4.2) then the destructor of the base class should be declared to be *virtual*. That way, when objects are destroyed through the pointer the destructor of the derived class is called rather than the base class destructor.

2.3.5 Exercises

(1) Using the example program `shapes.cpp`, derive a class `filled_rect` using `rectangle` as the base class. The constructor for `filled_rect` should call the `rectangle`'s constructor. Modify the main program so that some filled rectangles are drawn. You can use the *oucsgraph* graphics library function `fillRectangle` to draw the shape.

(2) Derive a class `string` from the `String` class in the Borland class library. This new class should declare an `operator<<` function as a friend, and you should then define this function. This will allow `string`s to be printed using:

```
cout << str;
```

instead of using the `String` method `printOn`, i.e.

```
str.printOn(cout);
```

It will be necessary to define a constructor that takes a `const char *` parameter and simply passes this value on to the `String` constructor. A copy constructor will be provided for us by C++.

To use the `String` class from the Borland class library you should `#include <strng.h>` and link in `tclasss.lib` (you will need to create a project file to do this). You may also need to change the directories that Turbo C++ searches for include files and

libraries (this has been done for you on the system used during the course). Select the directories option of the `options` menu and change the directory paths to the following:

Include directories: `c:\borlandc\include;c:\borlandc\classlib\include`
Library directories: `c:\borlandc\lib;c:\borlandc\classlib\lib`

(3) Use inheritance to define a stack class in terms of the list class used in exercise 2.2.20...(3). Define `put` and `get` operations to add and remove items from the stack.

(4) The following exercise comes from *Advanced Programming and Problem Solving With Pascal* by Schneider & Bruell.

> Consider the following problem. You are given a stack and asked to reverse its contents, as in the following example.
>
> | | | | |
> |---|---|---|---|
> | Top -> | 5 | Top -> | 68 |
> | | 15 | | 19 |
> | | 35 | | 7 |
> | | 2 | | 2 |
> | | 7 | | 35 |
> | | 19 | | 15 |
> | | 68 | | 5 |
> | | (a) | | (b) |
>
> How would you accomplish this task? An elegant solution (although it requires more storage than is actually needed) employs a queue. You simply pop off elements of the stack one at a time and enqueue them (`put`). Then dequeue (`get`) the elements one at a time and push them onto the stack.

Write a program that creates an instance of a queue and a stack (`Queue` and `Stack` from the Borland class library). The program should then read integers until a 0 is entered. Each integer should be pushed onto the stack. This should leave you in position (a) above. The program should then `pop` all the entries from the stack and `put` them in your queue. You should then be able to `get` all the items from the queue and `push` them back onto the stack. The program should then print the contents of the stack, which should be the numbers you entered in the order that you entered them.

[Note that you will need to define an `Integer` class (say) that is derived from `Object`.]

3 Templates

As we've seen through the course, it pays to make functions and data structures as generic as possible. It would be a great shame if we had to write one set of code for a stack of integers and another for a stack of floats.

We've also seen two ways of achieving this.

The first is only to store pointers to our data in the data structures we create. By using the type `void *` we can subsequently store pointers to anything. The major problem with this approach is that we need to remember what kind of object the pointer was pointing to so that we can convert the pointer back to the right type when we remove data from the data structure. For example, if we coded a stack using generic pointers, then we would need to do the following when using `push` and `pop`:

```
stack s;
float *p;

p = new float;
*p = 45.1;
s.push(p);
...
p = (float *)s.pop();
```

The second technique is to make use of the fact that pointers or references to a derived class are type compatible to pointers or references to the base class. This is used to great effect in the Borland class library, where anything that is derived from `Object` can be stored in one of the data structures available in the library. In fact, different types of objects can be stored in the *same* stack or queue.

A third and better option now exists, *templates*. Templates can be used to define classes or functions. The code below specifies a stack class (Stroustrup [1] p. 256):

```
template<class T>
class stack {
   T     *v;
   T     *p;
   int   sz;

public:
   stack(int s) { v = p = new T[sz=s]; }
   ~stack() { delete[] v; }

   void push(T a) { *p++ = a; }
   T pop() { return *--p; }

   int size() const { return p-v; }
};
```

We can then use this template to declare stacks of whatever we want:

```
stack<shape *> ssp(200);   // stack of pointers to shapes
stack<int>     si(400);    // stack of integers
```

Note that we have to specify the type of the object to store in the stack in angle brackets. The type name we use there is used by the template as a replacement for `T` in the template above.

Whilst we have lost the ability to store different objects in the stack (though derived classes are still acceptable), we have not lost the careful type checking of parameters that both of the first two methods suffer from.

As stated above, templates can be used when writing generic functions, for example, a function that swaps two values of the same type:

```
#include <iostream.h>

template<class T> void swap(T& a,T& b) {
   T  t;

   t = a;
   a = b;
   b = t;
}

void main()
{
   double   x = 1.0, y = 2.0;
   int      i = 5, j = 6;

   cout << i << ' ' << j << ' ' << x << ' ' << y << endl;
   swap(x,y);
   swap(i,j);
   cout << i << ' ' << j << ' ' << x << ' ' << y << endl;
}
```

4 Exceptions

Exceptions are a method of dealing with run-time errors. It is not always possible for functions to return an error value (e.g. a function that *pops* an `int` value from a stack, all possible return values are valid, there is no *special* value that can be used to indicate an error condition), and even if they can, it is awkward to always check the return value. Exceptions offer an alternative method of handling errors.

Exceptions are very new to C++, and there are not many compilers that support them.

Exceptions are *thrown* when errors occur; they can then be *caught* and appropriate action taken. Exceptions can only be detected inside `try` blocks, and can only be handled or *caught* inside `catch` blocks. The following program *throws* an error if the buffer cannot be allocated. The exception is *caught* and the program terminated after printing a suitable error message.

```
#include <iostream.h>

int main()
{
   try {
      …
      char *buffer = new char[size];

      if (buffer == 0)
         throw "Out of memory";


      …
   }
   catch (const char *str) {
      cerr << "Exception: " << str << endl;
      exit(1);
   }
}
```

There is no need for the exception to `throw` a string (as in this example), in fact objects of any type can be thrown.

5 Separate Compilation

One of C and C++'s advantages is the separate compilation of groups of functions. There are a few rules of thumb that help this process.

Consider the definition of a class, e.g. a stack. If it is intended to compile the class separately so that it can be linked into whatever programs require it, then the following makes life easier:

– Every source file that uses or defines any of the methods of stack (so this includes *both* the file that uses a stack *and* the file in which the methods are defined) needs to know what the class looks like. This means that they must all include the declaration of the stack class. This can be solved by putting the declaration in a header file and including it (with `#include`) in any source file that refers to stack.

– As there can only be one declaration of a stack in any one source file, prevent multiple including of the header file by defining a preprocessor constant. Only if this constant has not been defined are the contents of the header file used.

– Put the non-inline methods of stack in a separate source file (remembering to `#include` the declaration of stack) and compile it to an object file.

– Compile the program that uses a stack, but when it comes to linking link in the object file produced above.

stack.h

```
#ifndef __STACK__
#define __STACK__

#include <object.h>

class stack {
   Object   **data;  // a pointer to a pointer to an Object
   int      ptr;     // pointer to next free location

public:
   // in-line constructor
   stack()
   {
      data = new Object*[100];
      // create an array of 100 pointers to Object

      ptr = 0;
      // ptr points to first free element in array
   }

   // in-line destructor
   ~stack()
   {
      delete [] data;
   }

   // declarations for push, pop & isEmpty
   void push(Object*);
   Object *pop();
   int isEmpty();
};
#endif
```

stack.cpp

```
#include "stack.h"

void stack::push(Object *obj)
{
   data[ptr++] = obj;
}

Object *stack::pop()
{
   return data[--ptr];
}

int stack::isEmpty()
{
   return ptr == 0;
}
```

usestack.cpp

```
#include "stack.h"
#include <strng.h>

int main()
{
   stack       s;
   String      str1("hello"),str2("world");

   s.push(&str1);
   s.push(&str2);

   while(!s.isEmpty())
      cout << *s.pop();

   return 0;
}
```

To compile the stack class separately using a command line compiler, type:

```
cc -c stack.cpp
```

Where *cc* is your C++ compiler (e.g. `bcc`, `tcc`, `gcc` etc.).

To compile the program that uses a stack using a command line compiler, type:

```
cc usestack.cpp stack.obj tclasss.lib
```

If you are using the Turbo or Borland C++ IDE (Integrated Development Environment) then create a project file that includes the following files:

```
usestack.cpp
stack.obj
tclasss.lib
```

Or replace `stack.obj` with `stack.cpp` if it is likely that the file `stack.cpp` or `stack.h` will change.

Notice also that as both `stack.cpp` and `usestack.cpp` include the same header file, if the class declaration is ever changed both will automatically pick up the *same* new version.

6 C++ Versions

There is not yet an ANSI or ISO standard for C++ (but it is being worked on!). C++ is an evolving language, so some of the features described in this guide may not be available on all C++ compilers. C++ started life at AT&T using a C++ to C translator called *cfront*. AT&T (actually Unix Systems Laboratories now) still sell cfront, and people often refer to the different versions of cfront when talking about new language features. The following description of cfront versions comes from the C++ *Frequently Asked Questions* (FAQ) list posted to the USENET news group *comp.lang.c++*:

1.2	as described in the first edition of Stroustrup's *The C++ Programming Language*
2.0	multiple/virtual inheritance and pure virtual functions
2.1	semi-nested classes and `delete []`
3.0	fully nested classes, templates and syntax to define prefix and postfix increment and decrement operators
4.0(?)	will include exceptions

Borland C++ 4.0 does support exceptions. The Borland Turbo C++ 3.1 compiler used in class is roughly equivalent to cfront 3.0.

7 Bibliography

Books used in the development of this course, and in teaching myself C++ were:

Bjarne Stroustrup: *The C++ Programming Language, Second Edition*, Addison-Wesley 1992, £24.95.

One of the two main C++ references (the other is the *Annotated C++ Reference Manual* or ARM). Extremely useful, but not really a tutorial text.

Robert Lafore: *Object–Oriented Programming In Turbo C++*, Waite Group Press 1991, £26.95.

Fast becoming a favourite of mine.

Scott Meyers: *Effective C++*, Addison-Wesley 1992, £20.95.

Useful collection of hints and tips.

Scott Robert Ladd: *C++ Techniques and Applications*, M&T Books 1990, £19.95.

Some very useful information, but not as good (in my view) an introduction as Lafore.

Cay S. Horstmann: *Mastering C++, An Introduction to C++ and Object–Oriented Programming For C and Pascal Programmers*, John Wiley 1991, £21.50

Small, light, extremely informative, but only really if you know C quite well.

Entsminger & Eckel: *The Tao of Objects*, M&T 1991, £19.95

I personally found this a “right rivetting read”. It has examples in Turbo Pascal and C++, and covers all sorts in interesting aspects of the design of Object Oriented programs.