More C++

Tim Love

April 21, 2016

Abstract

This document aims to provide people who have done an introductory course in Python or C++ the skills required to write bigger C++ programs. Online versions are at `http://www-h.eng.cam.ac.uk/help/tpl/languages/C++/doc/doc.html` (HTML); `http://www-h.eng.cam.ac.uk/help/tpl/languages/C++/doc/doc.pdf` (PDF); and `http://www-h.eng.cam.ac.uk/help/documentation/docsource/index.html` (LaTeX).

Contents

1 Requirements

You will need a C++ compiler that can cope with C++11 code. The code in this document was compiled using `g++` version 4.8.5 (using `g++ -std=c++11`). It's free.

Online compilers like the one at `http://www.tutorialspoint.com/compile_cpp11_online.php` are also available.

2 Introduction

This document is aimed at those who have done 1A and 1B Python or C++ computing courses at CUED, but it should also be useful to others who've already programmed and want a fast-track path to C++ knowledge.

C (and hence C++) can be used as a low-level language to interface with hardware or write speed-critical number-crunching applications (not least operating systems), but C++ also offers high level facilities like classes. C++11 (which is what this document assumes) is a newer release of C++ that added a few more language facilities and support for threads, regular expressions, and statistical distributions to the standard library.

Python users will note many similarities between C++ and Python, though the syntactic details differ. C++ has fewer easy-to-use "packages" (add-ons). In particular, graphics is less easy. The main conceptual difference is that Python is interpreted whereas C++ code needs to be compiled. The difference is analogous to the difference between using a simultaneous interpreter when interacting with a person who speaks a different language to you, and using a translator, who rather than converting word by word wait for the original work to be completed.

C++ can be used in various styles: procedural (like Fortran or C); Object-orientated; Generic, etc. You may need to work in only one of these ways (in which case large sections of this document are no use to you), or you can mix styles. This document covers the techniques to support each of these programming paradigms, dealing with

- Classes
- Containers and Algorithms
- Techniques useful when writing big, project-sized programs.

To aid readability, the examples in the text are mostly code fragments. Full program listings - `http://www-h.eng.cam.ac.uk/help/tpl/languages/C++/examples/` are online. On CUED's C++ page `http://www-h.eng.cam.ac.uk/help/tpl/languages/C++.html/` you'll find several articles dealing with selected topics in more detail than there's space to go into here.

A few programming exercises are suggested in this document. Any computing book will offer more.

3 Review

CUED's C++ Summary `http://www-h.eng.cam.ac.uk/help/mjg17/teach/CCsummary/` covers many points. What follows is a brief summary

3.1 Keywords

Many of these keywords you'll never need. They're included here for completeness because you can't use them for variable names, etc.

alignas	alignof	and	and_eq	asm	auto
bitand	bitor	bool	break	case	catch
char	char16_t	char32_t	class	compl	const
constexpr	const_cast	continue	decltype	default	delete
do	double	dynamic_cast	else	enum	explicit
export	extern	false	float	for	friend
goto	if	inline	int	long	mutable
namespace	new	not	not_eq	nullptr	operator
or	or_eq	private	protected	public	register
reinterpret_cast	return	short	signed	sizeof	static
static_assert	static_cast	struct	switch	template	this
thread_local	throw	true	try	typedef	typeid
typename	union	unsigned	using	virtual	void
volatile	wchar_t	while	xor	xor_eq	

3.2 Built-in Types and Enumerations

There are integral types (`char`, `short`, `int`, `long`) which can be intended to hold signed (the default) or unsigned values. So for example "`unsigned char c`" creates a one-byte variable big enough to hold any integer from 0 to 255. There are also floating point types (`float`, `double`, `long double`).

If it's clear to the compiler what type a variable should be, the `auto` keyword can be used. i.e.

```
auto i=5;
```

is possible.

If you want an integer variable to be restricted to containing just a few specific named values (it makes your code easier to read, and safer) you can create an enumerated type

```
enum user_type {ugrad, pgrad, staff, visitor};
user_type u;
```

creates an enumerated type called `user_type`. The variable `u` can only be set to `ugrad`, `pgrad`, `staff` or `visitor` (which have the values 0, 1, 2 or 3. The named values can be set explicitly to be other than the default values; for example

```
enum user_type {ugrad=2, pgrad=7, staff=9, visitor=3};
user_type u;
```

Enumerated types can be used wherever integer types can be, though some operations, u++ for example, aren't allowed.

3.3 Operators

The lines of the table are in order of precedence, so '`a * b + 6`' is interpreted as '`(a * b) + 6`'. When in doubt put brackets in!

The **Associativity** column shows how the operators group. E.g. '<' groups left to right, meaning that $a < b < c$ is equivalent to $(a < b) < c$ rather than $a < (b < c)$. Both are pretty useless expressions - $(a < b)$ evaluates to `true` (1) or 0 depending on whether it's true or not.

Associativity	Operator
left to right	`::`
left to right	`() [], ->, ., typeid`, casts,
right to left	`!` (negation), `~` (compl)
	`new, delete ++, --, - (unary) , * (unary), & (unary), sizeof`
right to left	`(type)`
left to right	`*, /, % (modulus)`
left to right	`- +`
left to right	`<<, >>`
left to right	`<, <=, >, >=`
left to right	`==, !=`
left to right	`&` (bitand), `|` (bitor)
left to right	`^` (xor)
left to right	`&&` (and)
left to right	`||` (or)
right to left	`?:`
right to left	`=, +=, -=, /=, %=, >>=, &=`
left to right	`throw`
left to right	`,`

3.4 Selection

3.4.1 if

```
...
if (i==3) // checking for equality; '!=' tests for inequality
   // no braces needed for a single statement
   j=4;
else{
  //  the braces are necessary if the
  //  clause has more than one statement
   j=5;
   k=6;
}
...
```

3.4.2 switch

```
...
// switch is like a multiple 'if'.
// The values that the switching variable is compared with
// have to be integer constants, or 'default'.

switch(i){
case 1: printf("i is one\n");
        break; // if break wasn't here, this case will
               // fall through into the next.
case 2: printf("i is two\n");
        break;
default: printf("i is neither  one nor two\n");
        break;
}
...
```

3.5 Loops

Note that if you use the Standard Library's containers and vectors (see later), explicit loops might not be needed very much.

3.5.1 while, do

...

```
while(i<30){    // test at top of loop
  something();
...
}

...
do {
   something();
} while (i<30); // test at bottom of loop
...
```

3.5.2 for

The '`for`' construction in C++ is very general. In its most common form it's much like `for` in other languages. The following loop starts with i set to 0 and carries on while `i<5` is true, adding 1 to i each time round.

```
...
for(int i=0; i<5; i=i+1){
   something();
}
...
```

The general form of '`for`' is

```
for (expression1; expression2; expression3)
     something();
```

where all the expressions are optional. The default value for `expression2` (the while condition) is `1` (`true`). Essentially, the `for` loop is a `while` loop. The above `for` loop is equivalent to

```
...
expression1;         // initialisation
while (expression2){ // condition
   something();
   expression3;      // code done each iteration
}
...
```

E.g. the 2 fragments below are equivalent (except that the `i` in the "while" example still exists after the looping has finished). '`i`' is set to 3, the loop is run once for `i=3` and once for `i=4`, then iteration finishes when `i=5`.

```
int total=0;
for (int i = 3; i < 5; i=i+1)
   total = total + i;
```

```
int total=0;
int i = 3;
while(i < 5){
  total = total + i;
  i=i+1;
}
```

Where the range of the variable is clear to the compiler, a range-based "for" loop can be used - e.g.

```
int total=0;
int arr[]={1,2,3};
for (auto e:arr)
   total = total + e;
```

Within any of the above loop constructions, `continue` stops the current iteration and goes to the start of the next, and `break` stops the iterations altogether. E.g. in the following fragment `i = 0` and `2` will be printed out.

```
...
int i=0;
while (i<5){
   if (i==1){
     i = i+1;
     continue;
   }
```

```
   if (i==3)
     break;
   cout << "i = " << i << endl;
   i=i+1;
}
...
```

If you want a loop which only ends when `break` is reached, you can use '`while(true)`' or '`for(;;)`'.

3.6 Aggregation

Variables of the same or different types can be bundled together into a single object.

3.6.1 arrays

Variables of the same type can be put into arrays.

```
char letter[50];
```

defines an array of 50 characters, `letter[0]` being the first and `letter[49]` being the last character. C++ arrays have no subscript checking; if you go off the end of an array C++ won't warn you.

Multidimensional arrays can be defined too. E.g.

```
char values[50][30][10];
```

defines a 3D array. Note that you can't access an element using `values[3,6,1]`; you have to type `values[3][6][1]`.

The Standard Library offers various alternatives (`vector`, for example) which are often preferable to the basic arrays described above because

- They can be made to grow on demand
- Many routines for copying and modifying them exist.

3.6.2 structures and classes

Variables of different types can be grouped into a *struct*ure or a *class*.

```
class person {
public:
     int age;
     int height;
     string surname;
};

person fred, jane;
```

defines 2 objects of type `person` each of 3 fields. Fields are accessed using the '`.`' operator. For example, `fred.age` is an integer which can be used in assignments just as a simple variable can.

Structure and Class objects may be assigned, passed to functions and returned, but they cannot (by default) be compared, so continuing from the above fragment `fred = jane;` is possible (the fields of `jane` being copied into `fred`) but you can't then go on to do

```
if (fred == jane)
  cout << "The copying worked ok\n";
```

- you have to compare field by field.

Classes can contain member functions as well as data. The data can be made `private` so that other objects can't access it directly.

3.7 Pointers

Even if you don't use pointers yourself, the code you'll learn from is likely to have them. Suppose `i` is an integer. To find the address of `i` the `&` operator is used (`&i`). Setting a pointer to this value lets you refer indirectly to the variable `i`. If you have the address of a variable and want to find the variable's value, then the dereferencing operator `*` is used.

```
...
int i=0;
cout << i << " is at memory location " << &i << endl;

// The next statement declares i_ptr to be a pointer pointing
// to an integer. The declaration says that if i_ptr is
// dereferenced, one gets an int.

int *i_ptr;
i_ptr = &i; // initialise i_ptr to point to i

// The following 2 lines each set i to 5
i = 5;
*iptr = 5; // i.e. set to 5 the int that iptr points to
```

Pointers aren't just memory addresses; they have types. A pointer-to-an-`int` is of type `int*`, a different type to a pointer-to-a-`char` (which is `char*`). The difference matters especially when the pointer is being incremented; the value of the pointer is increased by the size of the object it points to. So if we added

```
iptr=iptr+1;
```

in the above example, then `iptr` wouldn't be incremented by 1 (which would make it point somewhere in the middle of `i`) but by the length of an `int`, so that it would point to the memory location just beyond `i`. This is useful if `i` is part of an array. In the following fragment, the pointer steps through an array.

```
...
int numbers[10];

numbers[0] = 1;
numbers[1] = 2;
numbers[2] = 3;

int *iptr = &numbers[0]; // Point iptr to the first element in numbers[].
                    // A more common shorthand is   iptr = numbers;
// now increment iptr to point to successive elements
for (int i=0; i<3; i++){
   cout << "*iptr is " << *iptr << endl;
   iptr= iptr+1;
}
...
```

Pointers are especially useful when functions operate on structures. Using a pointer avoids copies of potentially big structures being made.

```
class {
public:
     int age;
     int height;
     string surname;
} person;
```

```
person fred, jane;

int sum_of_ages(person *person1, person *person2){
int sum;  // a variable local to this function

    // Dereference the pointers, then use the '.' operator to get the
    // fields
    sum = (*person1).age + (*person2).age;
    return sum;
}
```

Operations like `(*person1).age` are so common that there's a special, more natural notation for it: `person1->age`.

3.8 Functions

The form of a function definition is

<function return type> *<function name>* `(` *<formal argument list>* `)`
{
<local variables>
<body>
}

E.g.

```
int mean(int x, int y)
{
int tmp;

tmp = (x + y)/2;
return tmp;
}
```

or, if the function is a member of class `stats`, `::`, the scope resolution operator can be used to specify which particular function of that name to access.

```
int stats::mean(int x, int y)
{
int tmp;

tmp = (x + y)/2;
return tmp;
}
```

You can provide default values for the trailing arguments. E.g.

```
int mean(int x, int y=5)
{
int tmp;

tmp = (x + y)/2;
return tmp;
}

...
int foo=mean(7)
```

is legal, setting `foo` to 6.

You can have more than function of a particular name provided that the compiler knows unambiguously which one is required for a function call, so the functions with identical names have to take different types of arguments or must only be visible one at a time. It's not enough for functions to differ only in the return type.

By default, the variables given to a function won't have their values changed when the function returns. In the following fragment `number` *won't* be increased to 6.

```
void add (int x)
{
  x = x+1;
}

int number = 5;
add(number);
```

When `add(number)` is called, the `x` variable is set to the current value of `number`, then incremented, but `number` is unchanged. The following slight variation that uses "call by reference" *will* increase `number`

```
void add(int& x)
{
  x = x+1;
}

int number = 5;
add(number);
```

3.9 Declarations

First, a note on terminology. A variable is *defined* when it is created, and space is made for it. A variable is *declared* when it already exists but needs to be re-described to the compiler (perhaps because it was *defined* in another source file). Think of *declaring* in C++ like *declaring* at customs – admitting to the existence of something. In C++ you can only define a variable once but you can declare it many times.

C++ declarations are not easy to read. You shouldn't need to use complicated declarations so don't worry too much if you can't 'decode' them. Keep a cribsheet of useful ones. The following examples show common declarations.

`int *p`	pointer to an int
`int x[10]`	an array of 10 ints
`const int *p`	pointer, p, to int; int can't be modified via p
`int * const p`	a constant pointer to an int
`int (*x)[10]`	a pointer to an array of 10 ints
`int *x[10]`	array of 10 pointers to ints
`int (*f)(int)`	pointer to a function taking and returning an int
`void (*f)(void)`	pointer to a function taking no args and returning nothing
`int (*f[])(int)`	An array of pointers to a functions taking and returning an int

Note the importance of the brackets in these declarations. On the CUED Central System the `c++decl` program can help. E.g.

```
c++decl explain "char *&x"
```

prints

```
declare x as reference to pointer to char
```

`typedef` can be used to create a shorthand notation for a verbose type.

```
typedef int (*PFI)(int) // declare PFI as pointer to function that
                        //   takes and returns an int.
PFI f[];  // declare f as an array of pointers of type 'PFI'.
```

3.10 Getting command line arguments

When a program is called from the Unix command line its name is sometimes followed by strings. For example

```
g++ -v cabbage.cc
```

calls the g++ program with an argument and a filename. There needs to be a way for g++ to get hold of these arguments and filenames. Also it's useful for g++ to be able to pass back to Unix a return value. There's a standard method for this. The first function called when a C++ program is run is always

```
int main(int argc, char *argv[])
```

`argc` is how many strings were on the command line. `argv[0]` is the first string (which will be the program name) and the other strings (if any) are `argv[1]`, etc. When `main` returns an integer, this is passed back to the calling process. To see how this works in practise, compile the following to produce a program called `test1`

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {
   cout << "The command line strings are -\n";
   for (int i=0;i<argc;i++)
      cout << argv[i] << endl;
   return argc;
}
```

Typing

```
./test1 -v foo
echo $?
```

should print out the strings, then print "3" - the value returned to the command line process.

3.11 Scope and Namespaces

A variable's scope is the part of the program where the variable's name has a meaning. The more localised the variables are, (the smaller the scope) the better. There's

- file scope - entities can be made visible only to entities in the same file.
- function scope - entities can be made visible only to the function they're created in.
- block scope - delimited by '{' and '}'.
- class scope - entities can be made visible only to the class they're created in.
- namespaces - A namespace is like a class that has no functionality. Stroustrup uses them a lot. You can put an entity into a namespace by doing something like

  ```
  namespace test {
   int i;
  }
  ```

  then using `test::i` to access the variable. The command "`using namespace test`" will make all the entities in the namespace available for the rest of the unit that the statement is in, without the "`test::`" being necessary. The standard library names are in the `std` namespace. It's tempting to put "`using namespace std`" at the top of each file to access all the standard routines, but this "pollutes the global namespace" with many routine names you'll never use, so consider using more specific commands like "`using std:string`"

It's possible for a local variable to mask a global variable of the same name. If, in a function that has a local variable `i` you want to access a global variable `i`, you can use `::i`, but it's better to avoid such nameclashes in the first place.

3.11.1 Exercises

- Write a `main` routine in one file that calls a function called `fun` that's in another file. `fun` just prints out a message. Put `fun` within a `namespace` and see if you can compile the code by appropriate use of `using`.

4 Object-orientated programming

Object-orientated languages support to a greater or lesser extent the following concepts

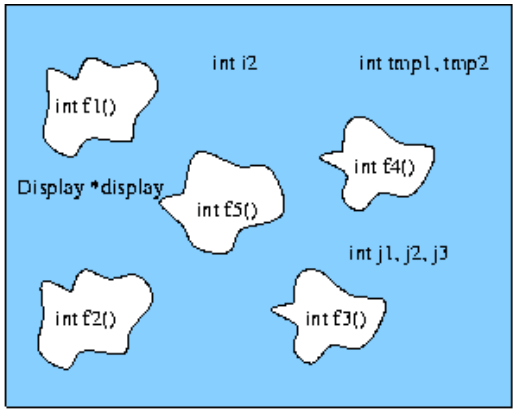
- Encapsulation (including in an object everything it needs, hiding elements that other objects needn't know about). This keeps data and related routines together and unclutters the large-scale organisation of the program. Each object has a 'public' set of routines that can be called, and these routines are all that other objects need to know.
- Inheritance (creating new types of objects from existing ones). Rather than having many seemingly unrelated objects, objects can be organised hierarchically, inheriting behaviour. Again, this simplifies the large-scale organisation.
- Polymorphism (different objects responding to the same message in different ways). Rather than having a different routine to do the same thing to each of many different types of objects, a single routine does the job. An example of this is how the + operator can be overloaded in C++ so that it can be used with new classes.

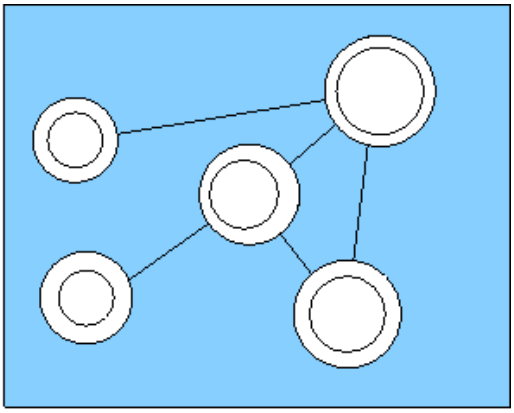
Using an Object-orientated language often means that

- program entities can more closely model real-world entities. As Stroustrup wrote, "For small to medium projects there often is no distinction made between analysis and design: These two phases have been merged into one. Similarly, in small projects there often is no distinction made between design and programming."
- complexity is more localised
- code re-use is easier

The aim is to create objects that have tightly related parts but few connections with anything else. As an analogy consider a lightbulb: as long as it has a standard interface other components don't have to worry about how it's designed internally, and the lightbulb can easily be replaced by one that's internally very different.

What you want to avoid is a primordial stew of global variables with routines floating in it that need the variables (see the diagram, left). This kind of organisation makes it harder to use the routines in other programs and leads to more errors. The O-O approach (see the diagram, right) tries to keep data (and related functions) tucked away safely inside objects.





Classes are at the heart of C++'s object-orientation. `http://www-h.eng.cam.ac.uk/help/tpl/languages/C++/examples/fileptr.cc` is an example of a simple class built from scratch. Constructor functions can be created that are called when an object is created. The default constructor claims enough memory. Just as local variables of type `int` are destroyed when no longer required, so are local objects. The destructor (the routine that's run when an object is destroyed) can be defined to do extra work.

This class has 2 constructors (one for the situation when it's given 2 strings, and the other when it's given a pointer to an open file). It also has a destructor that's called when the object goes out of scope (when the routine it's created in ends, for example). Here, the destructor closes the file automatically.

```
class File_ptr {
FILE *p;
```

```
public:
// 2 constructors
File_ptr (const char* n, const char* a) { p=fopen(n,a); }
File_ptr (FILE *pp) {p=pp;}

// destructor
~File_ptr() {fclose(p);}
// redefinition of (), providing a way to access the file pointer
operator FILE* () {return p;}
};


void use_file(const char* fn)
{
File_ptr f(fn,"r");
// file will be closed when f goes out of scope
// ...
}
```

4.1 Derived classes

The more you use C++ the more you'll be developing classes which require greater sophistication. Often you'll need to add extra functionality to an existing class. C++ provides a mechanism to build new classes from old ones

```
class More : public Base {
int value;
...
};
```

Here More inherits the members of Base. The `public` keyword means that the members of Base are as accessible to More as Base allows. By default, derivation is `private` (which means that even Base's `public` members can't be accessed by More), but usually derivation should be `public`, privacy being control by the base class.

When an object of type More is created, first the Base constructor is called, then More's constructor. When the object is destroyed, the destructors are called in reverse order. The constructors/destructors are not inherited, but you can say

```
More::More(int sz):Base(sz){}
```

passing arguments to a base member class constructor (or to a number of constructors using a comma-separated list).

A derived class can be assigned to any of its public base classes, so the following is possible

```
class More : public Base {
int value;
...
};
Base b;
More m;
b=m;
```

It will fill the fields of `b` with the corresponding values in `m`. However, `m=b` isn't possible.

You can derive new classes from a derived class, and you can derive many different classes from a base class so you can develop a family tree of related classes. As we'll see soon, the "parent" classes can keep parts of themselves private, not only to unrelated classes but to derived classes too.

Big classes don't necessarily imply big objects: all objects of a class share member functions, for instance.

It's possible for a class to be derived from 2 or more other classes. It's called Multiple Inheritance but it introduces complications, so don't use it unless you know what you're doing.

4.2 Friend classes and functions

It is sometimes useful to let the functions of one class have access to the components of another class without making the components `public` and without the overhead of having to call member functions to get private data.

The following shows how one class (in this case `Another`) can let another class (in this case `Base`) be a "friend" class

```
class Another {
friend class Base;
int value;
...
}
```

For the sake of clarity it's a good idea to have the `friend` lines at the start of the class definition. Note that friendship is granted by the class being accessed, not claimed by the class wanting access.

A friend may be a non-member function, a member function of another class, or an entire class. Friends and derived classes differ in that derived classes might not be able to access all the members of the base class.

4.3 Class member privacy

Class members can have 3 types of privacy

private - can be used only by member functions and friends

protected - like private except that derived classes have access to them too. Try not to use this - it can help performance but it breaks encapsulation.

public - available to any function.

By default class members are `private`. You often want the data to be private (so that they can't be tampered with from the outside) but the functions public (so that they can be called by other objects). If you want other objects to be able to change the data, write a member function that the objects can call and make sure that the member function does validity checking. Then other member function that use the data won't have to check for validity first.

A `private` function is often called a helper or utility function because its for the benefit of other member functions. Note that a member function

1. can access private data
2. is in scope of class
3. must be invoked on an object

If you have the choice of writing a friend or member function, choose a member function.

Note that `structs` are just classes which default to public members and public inheritance. By convention they're used (if at all) for data-only classes.

4.4 Static members

A static member is part of a class, but not part of an object of a class, so there's only one instance of them however many objects of that class are created. This is useful if, for instance, you want to keep a count of how many objects are created. A static member function

1. can access private data
2. is in scope of class
3. *but can't* be invoked on an object - *objectname.staticmemberfunction()* can't access any object-specific data.

4.5 Const members

The value of a const data member can't be changed. A const member function can't change the data of the class.

```
int day() const { return d};
```

doesn't modify state of the class.

4.6 Static Const members

Details regarding how to initialise static const members changed late in the C++ specification. A static const data member of integral or enum type can be initialized in the class definition, but you still need to provide another definition (without an initializer) for it. So

```
class C {
  const static int csi = 5;
};

const int C::csi;
```

is legal, but you can't do the same for floats, non-statics or non-consts. With some old compilers you have to initialise outside the class.

4.7 Overriding behaviour

As well as adding members a derived class can override behaviour simply by redefining a member function. For instance, if you are developing objects for a graphics editor, you'd like each kind of object to have a `draw` function, but you'd like each kind of object to have its own version.

Later (section 12) more details of class derivation will be described.

4.8 Exercises

1. Without writing any code, sketch out how you might create classes to represent Points, Triangles, Rectangles and Squares. How might the class hierarchy be extended to deal with Cuboids, Cubes and Prisms?
2. Write a program to show the order in which constructors and destructors are called in a hierarchy of derived classes.
3. Create a class containing 2 integers, then create an object of that class. See if the address of the object is the same as the address of one of the fields.

5 Generic programming

The Standard Library is a collection of functions many of which use templates, so it's worth knowing what they are

5.1 Templates

A class template is a "type generator": a way to fabricate many similar classes or functions from one piece of code. This is how the Standard Library can support many types of `vector`, etc. C++'s Templates are parameterized types. They support generic programming by obviating the need for explicit switch statements to deal with various data types. Let's look at an example. The following code (which doesn't use templates) swaps 2 integers in the usual C++ way

```
void swap (int& a, int& b) {
   int tmp = a;
   a = b;
   b = tmp;
}
```

If you wanted to swap other types of variables (including those you've defined) you could copy this code, replacing `int` by (say) `float`. But in situations like this, where the code is potentially generic, templates can be used.

```
template <class T>
void swap (T& a, T& b) {
   T tmp = a;
   a = b;
   b = tmp;
}
```

Here the "T" name is arbitrary (like a formal parameter in a function definition) and the `class` keyword can be replaced by the newer `typename` keyword. Note that the function is written much as before. When the compiler is now given the following code

```
int a = 3, b = 5;
float f=7.5, g=9.5;
swap (a, b);
swap (f, g);
```

it will create a version ("instantiation") of `swap` for each type required. These versions have nothing in common except that they were created from the same template - they won't share members, not even `static` ones. Templates are an example of source code re-use not object code re-use.

You can have template functions, but also template classes and derived classes.

The Standard Library includes container classes: classes whose purpose is to contain other objects. There are classes for vectors, lists, etc. Each of these classes can contain any type of object (as long as the object can be copied safely - if it can't, use pointers to elements instead). You can, for example, use a `vector<int>` (meaning 'a vector of `int`s') in much the same way as you would use an ordinary C array, except that `vector` eliminates the chore of managing dynamic memory allocation by hand.

```
vector<int> v(3);    // Declare a vector (of size 3 initially) of ints
v[0] = 7;
v[1] = v[0] + 3;
v[2] = v[0] + v[1];  // v[0] == 7, v[1] == 10, v[2] == 17
v.push_back(13);     // add another element - the vector has to expand
```

Note the use of the `push_back` function - adding elements this way will make the container expand.

The Standard Library also includes about 70 algorithms that manipulate the data stored in containers - reverse, insert, unique, transform etc. Note that these operations act on the elements without explicit use of loops.

To use these features you need to include the corresponding header file. For example, `#include <vector>` is needed for vectors. See the "C++ and the STL" document for examples - `http://www-h.eng.cam.ac.uk/help/tpl/talks/C++.html`.

5.2 Iterators

Iterators are like pointers. They save algorithm writers the worry of having to deal with the internals of containers. They come in various types: random access, bidirectional, forward, reverse, input, output. They can be `const`. Some containers support only some Iterator types (you can't for example randomly access a list). See the documentation for details.

5.3 Strings

In older C++ books and in C books, strings are arrays of characters - C-style strings. Except when speed is vital and safety isn't, C++ `string`s are preferable to these. For some operators (concatenation, for example) `string`s may be a lot faster than C's character arrays. Also

- They grow on demand
- The Standard Library algorithms work on them
- Other containers use them
- Natural constructions like `s=s1+s2;` are possible

String objects have many features in common with other Standard Library objects. Since strings are familiar to you, it's useful to look at the facilities in more detail to prepare you for other Standard Library components.

5.3.1 Iterators

The following routines provide iterators (pointers) of various kinds, depending on whether you want to go forwards or backwards through the string and whether you want to change the string. The same kind of routines exist for `vector` etc.

```
// Iterators - note the various types
 iterator       begin();
 iterator       end();
 reverse_iterator       rend();
 ...
```

5.3.2 Size

These routines (similar ones exist for other containers) give you size information. `reserve()` doesn't change size or initialize elements - `resize()` does. `capacity()` − `size()` = reserved size.

```
// Capacity
   size_type       size() const;
   size_type       length() const;
   void            resize(size_type);
   bool            empty() const;
   ...
```

5.3.3 Routines

These string-specific routines (only a selection is shown below) provide the functionality that C people are used to. The names aren't too hard to remember, and the default arguments are reasonable. The routines are overloaded, so `find` can be used to look for strings, characters or even C-strings.

Note also that the `c_str` routine returns a C string.

```
   const charT* c_str() const;
   size_type find(const basic_string&,
                  size_type = 0) const;
   size_type find(charT, size_type = 0) const;
   size_type find_first_of(const basic_string&,
                           size_type = 0) const;
   size_type find_last_of(const basic_string&,
                          size_type = npos) const;
   size_type find_first_not_of(const basic_string&,
                               size_type = 0) const;
   basic_string substr(size_type = 0, size_type = npos) const;
   int compare(const basic_string&) const;
```

The case history (section **??**) has many examples of `string` usage. Here are 2 short programs -

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
 string s1= "pine";
 string s2= "apple";
 cout << "length of s1 is " << s1.length() << endl;
 string s3= s1+s2;
 cout << "s1 + s2 = " << s3 << endl;
 string s4= s3.substr(2,4);
 cout << "s3.substr(2,4)=" << s4 << endl;

 cout << "s3.find(\"neap\")=" << s3.find("neap") << endl;
 cout << "s3.rfind('p')=" << s3.rfind('p') << endl;
```

```
 return 0;
}
```

To get a line of text (that may include spaces) from the user, do

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
 string str;
 cout <<"Type a string ";
 getline(cin,str);
 cout <<"you typed " << str << endl;
 return 0;
}
```

5.4 vector

Standard Library Vectors can have various element types and like other containers can be initialised in various ways:

```
vector <int> v; // an empty integer vector
vector <int> v(5); // 5 elements, initialised to the default
                   // value of the elements, in this case 0.
                   // Vector resizing is expensive, so set to
                   // the max size if you can.
vector <int> v(5,13); // 5 elements initialised to 13
vector <int> w(v.begin(),v.end());  // initialised from another container.
```

Many routines are available for vectors. For example, if you have a vector of strings, you can remove all the strings beginning with 'p' by doing the following

```
   // for this method you need to sort the items first
   sort(fruit.begin(), fruit.end());
   // create an iterator p1 for use with a vector of strings
   // and set it to point to the first string to remove
   auto p1= find_if(fruit.begin(), fruit.end(), initial('p'));
   // create and set p2 to point to the first string after
   // those that begin with 'p'
   auto p2= find_if(p1, fruit.end(), initial_not('p'));
   // Now erase
   fruit.erase(p1, p2);
```

Note that you can't do the following (searching from each end of the vector) because forward and reverse iterators are different types, and you can't use `erase` with mixed types.

```
  sort(fruit.begin(), fruit.end());
  auto p1= find_if(fruit.begin(), fruit.end(), initial('p'));
  auto p2= find_if(fruit.rbegin(), fruit.rend(), initial('p'));
  fruit.erase(p1, p2);
```

These above examples use the standard algorithms, which are always useful as a fallback option, but the `vector` class has a more specialised routine which is likely to be faster

```
fruit.remove_if(initial('p'));
```

Remember that

- vector elements can move! `insert`, for instance, changes its 1st argument.

- `vector` doesn't have arithmetic operations: `valarray` (which is described later) does.
- `vector` doesn't have range-checking by default.

You might hope that there's a one-liner to print the elements of a vector. The following implements it using a lambda function

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> v={1,2,3};
    for_each(v.begin(),v.end(), [] (int n) {cout <<n <<endl;});
}
```

5.5 Queues

Variations include

- deque - (pronounced deck) double-ended queue. One of the basic containers.
- queues - items are pushed into the back of the container and removed from the front. The underlying container can be `vector` or a `deque`. The items can be whatever you like. For instance you could create a queue of messages

```
// create a message structure
struct Message{ ... }

// wait until an item appears on the queue, then process it.
void server(queue<Message> &q)
{
  while(!q.empty()) {
    Message& m = q.front();
    m.service();q.pop();
  }
}
```

- priority queues - items are pushed into the back of the container and removed from the front in order of priority. The underlying container can be vector, amongst other things. In the following example the priority is determined by comparing the value of the integers in the queue using `less`.

```
#include <queue>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
  // Make a priority queue of int using a vector container
  priority_queue<int, vector<int>, less<int> > pq;

  // Push a couple of values
  pq.push(1);
  pq.push(7);
  pq.push(2);
  pq.push(2);
  pq.push(3);
  // Pop all the values off
  while(!pq.empty()) {
```

```
    cout << pq.top() << endl;
    pq.pop();
  }
return 0;
}
```

5.6 list

Lists are useful when items are likely to be inserted/deleted from the middle of long sequences. This example creates a vector then copies it into a list, reversing as it does so.

```
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
vector<int> V;
V.push_back(0);
V.push_back(1);
V.push_back(2);

list<int> L(V.size());
reverse_copy(V.begin(), V.end(), L.begin());
exit(0);
}
```

5.7 map

A `map` stores a mapping from one set of data (e.g. Names) to another (e.g. Phone numbers). The following example shows how to set up an "associative array" – an array where you can use a string as an index. It records how many times each word appears on `cin`, the standard input.

```
int main()
{

string buf;
map<string, int> m;
while (cin>>buf) m[buf]++;
 // write out result (see next example)
}
```

A `map` keeps items in order and has no duplicate keys. Variations include

- `hash_map` - an unsorted map.
- `multi_map` - can have duplicate keys
- `set` - a map with ignored values

```
// read in lines of the form
//  red 7
//  blue 3
//  red 4

void readitems(map<string, int>& m)
{
  string word;
```

```
  int val=0;
      while(cin>>word>>val) m[word]+=val;
}

int main()
{
map<string, int>tbl;
readitems(tbl);
int total=0;
// We want to create an iterator p for use with this type of map.
// Since we're not going to change the map values using it,
// we'll make a const_iterator. We could do this using
//    map<string, int>::const_iterator p
// but if we wanted to create other iterators of this type
// it would be tedious. So we use 'typedef' to make 'CI' a
// shorthand for the long expression.

typedef map<string, int>::const_iterator CI;
   for (CI p=tbl.begin(); p!=tbl.end(); ++p) {
      total+=p->second;
      cout<<p->first << '\t' << p->second << '\n';
   }
   cout << "total\t" << total << endl;
   return !cin;
}
```

Here's a fragment using `multimap` showing how to print out all the values associated a particular key (in this case the string "Gold"). It uses `typedef` as in the above example, and the useful `equal_range` function which returns an iterator to the first and last elements with a particular key value. Pairing up variables is so common that a facility called `pair` declared in the `<utility>` file is available to assist in their creation. The pair used here holds 2 iterators, but it can hold 2 items of different types too.

```
void f(multimap<string,int>&m)
{
typedef multimap<string, int>::iterator MI;
pair <MI,MI> g = m.equal_range("Gold");
   for (MI p=g.first; p!=g.second; ++p) {
      // print the value out
   }
}
```

In maps, `m["blah"]=1;` overwrites, `m.insert(val)` doesn't.

5.8 bitset

This provides efficient operators on bits.

```
#include <iostream>
#include <bitset>
using namespace std;

int main()
{
const unsigned int width = 32;
bitset<width> b1=23;
bitset<width> b2 = 1023;
bitset<width> b3;
 cout << "Type in a binary number (type a non-binary digit to end) ";
 cin >> b3;
 cout << "You typed " << b3 << endl;
 cout << "b2 flipped=" << b2.flip() << endl;
```

```
 cout <<  "b1=" << b1 << endl;
return 0;
}
```

5.9 valarray

`valarrays` are designed for doing fast vector calculations. They can be initialised in various ways

```
// v1 is a vector of 2000 integer elements each initialised to 7
valarray<int> v1(7,2000);

// v2 is initialised to the first 4 elements of d
const double d[]= {1.0, 1.0, 2.0, 3.0, 5.0};
valarray<double> v2(d,4);

// v3 is a copy of v2
valarray<double> v3=v2;
```

Operations include

```
v2 = 5;           // sets each element to 5
v3 *= .5;         // halve each element
v2 = v3.shift(2); // copy the elements of v3 shifted 2 to the left, into v2
                  // filling spaces with 0.0
v3 = v2.cshift(-2) //copy the elements of v2 shifted 2 to the right, into v3
                  // filling the spaces with elements that have fallen off
                  // the other end
v2= v2*cos(v3);
```

A `slice` is what you get when you take every nth element of a `valarray`. This is especially useful if the `valarray` represents the values in a 2D array - slices are the rows or columns. A slice needs to know the index of the first element, the number of elements and the 'stride' - the step-size.

```
slice_array<double>& v_even= v2[slice(0,v2.size()/2,2);
slice_array<double>& v_odd= v2[slice(1,v2.size()/2,2);
v_odd *=2 ; // double each odd element of v2
```

A `gslice` lets you extract subarrays from a valarray that represent the values in a 2D array.

A mask array lets you pick out arbitrary elements. It needs to contain bool values and shouldn't be bigger than the valarray it's used with. When used as a subscript, the valarray elements corresponding to the true elements of the mask array will be chosen.

```
bool b[]= { true, false, true, true};
valarray<bool>mask(b,4);
valarray<double>v4= v[mask];
```

There's also an indirect array.

`valarrays` are built for speed - they aren't intended to grow, and operations aren't error-checked - if you multiply together 2 `valarrays` of different lengths the behaviour is undefined.

5.10 Algorithms

Algorithms (in `<algorithm>`) operate on containers, including `strings`. They may not always be the fastest option but they're often the easiest. They fall into 6 groups

- **Search algorithms** - `search()`, `count_if()`, etc.
- **Sorting algorithms** - `sort()`, `merge()`, etc.
- **Deletion algorithms** - `remove()`, `unique()`, etc.
- **Numeric algorithms** - `partial_sum()`, `inner_product()`, etc.

- **Generation algorithms** - `generate()`, `for_each()`, etc.
- **Relational algorithms** - `equal()`, `min()`, etc.

Algorithms usually take as their first 2 arguments iterators to mark the range of elements to be operated on. For example, to replace 'A' by 'B' in a string `str` one could do

```
replace(str.begin(), str.end(), 'A', 'B');
```

Here's one way to count how many instances of a particular character there are in a string using the `find` algorithm.

```
int count (const string& s, char c)
{
auto i=find(s.begin(), s.end(),c);
int n=0;
  while(i!=s.end()){
    ++n;
    i=find(i+1, s.end(),c);
  }
  return n;
}
```

Here's a way to count the number of 'z' characters in a C-style string using the `count` algorithm. Note how the character array can be used like other containers.

```
void g(char cs[], int sz)
{
  int i1 = count(&cs[0], &cs[sz], 'z');
}
```

Here's the `count` algorithm again, this time being used to find a particular complex number in a list.

```
void f(list<complex>&lc)
{
int i1 = count(lc.begin(), lc.end(), complex(1,3));
}
```

Other algorithms include `next_permutation` and `prev_permutation`

```
int main()
{
char v[]="abcd";
  cout << v << '\t';
  while (next_permutation(v, v+4))
      cout << v << '\t';
}
```

`random_shuffle`,

```
char v[]="abcd";
  cout << v << '\t';
  for (int i=5;i;i--) {
     random_shuffle(v, v+4);
     cout << v << '\t';
  }
```

and a routine to cycle elements

```
rotate (first, middle, last)
```

`rotate` "swaps" the segment that contains elements from first through middle-1 with the segment that contains the elements from middle through last.

5.11 Set algorithms

Set algorithms include

- `includes()`
- `set_union()`
- `set_intersection()`
- `set_difference()`
- `set_symmetric_difference()`

The following, while demonstrating some of these routines, uses 2 ideas that are often useful

- an output stream can be thought of as a container, so you can "copy" other containers onto it. The line

  ```
  copy(s1.begin(), s1.end(), ostream_iterator<int>(cout," "));
  ```

  also puts spaces between the integers being printed. For now, don't worry if you don't understand the mechanics - it's useful so just use it!

- Sometimes you won't know how many resulting elements will be produced by an operation, you just want them all added to a container. `inserter` (which calls the container's `insert()` routine), `front_inserter` (which calls the container's `push_front()` routine - which vectors for example don't have) or `back_inserter` (which calls `push_back()`) do that for you.

```
#include <algorithm>
#include <set>
#include <iostream>
#include <iterator> // needed for ostream_iterator
using namespace std;

int main()
{
  //Initialize some sets
  int a1[10] = {1,2,3,4,5,6,7,8,9,10};
  int a2[6]  = {2,4,6,8,10,12};

  set<int> s1(a1, a1+10), s2(a2, a2+6), answer ;
  cout << "s1=";
  copy(s1.begin(), s1.end(),
       ostream_iterator<int>(cout," "));
  cout << "\ns2=";
  copy(s2.begin(), s2.end(),
       ostream_iterator<int>(cout," "));

  //Demonstrate set_difference
  set_difference(s1.begin(), s1.end(),
                 s2.begin(), s2.end(), inserter(answer,answer.begin()));
  cout << "\nThe set-difference of s1 and s2 =";
  copy(answer.begin(), answer.end(),
       ostream_iterator<int>(cout," "));
  cout << endl;
  return 0;
}
```

5.12 Using member functions

Algorithms can use the member functions of objects they're manipulating.

```
void print_list(set<Employee*>& s)
{
for_each(s.begin(), s.end(), mem_fun(&Employee::print));
}
```

5.13 Predicates

Some algorithms can be given a function to modify their behaviour. For example, the `find_if` algorithm can find items that conform to a particular condition as specified by a function. A predicate is a function object (an object that behaves like a function) that returns a `bool` showing whether or not an item matches the condition.

5.13.1 Creating predicates

`<functional>` supplies common conditions that can operate on various types: e.g., `greater_equal<int>()`, `plus<double>()`, `logical_or<int>()` etc. The following does a vector multiplication of `a` and `b`, putting the answer in `res` to which elements are added one at a time.

```
void discount(vector<double>& a, vector<double>& b, vector<double>& res)
{
transform(a.begin(), a.end(), b.begin(), back_inserter(res), multiplies<double>());
// vector multiplication
}
```

5.13.2 Adapters

Note: These are sometimes useful but look complicated, so don't worry if you don't understand them. Examples are online - `http://www-h.eng.cam.ac.uk/help/tpl/talks/C++.html` .

Adapters are ways to adapt existing predicates - they take and return a function that we can then use with Standard Library algorithms. There are 4 types -

binder - to use a 2-arg function object as a 1-arg function. `less` is a binary predicate. If we want make a unary predicate from it we can do `bind2nd(less<int>(),7))`.

member function adapter - to use a member function

pointer to function adapter - to use a pointer to a member function

negator - to negate an existing predicate

These can be combined -

```
... find_if ( ..... not1(bind2nd(ptr_fun(strcmp), "funny")));
```

5.14 Exercises

Some of the exercises below can be answered in a dozen or so lines of code using the Standard Library, but you'll need to explore the documentation first.

1. Write a program that given a string by the user determines whether it reads the same both ways
2. Each line in the unix password file (`/etc/passwd`) begins with a uid (up to the first ':'). Write a program that reads the uids into a `vector` of `string`s, sorts them and then prints out those beginning with 'q'.
3. Write a program that you can play tic-tac-toe (noughts and crosses) with.

6 Input/Output

As with C, Input/Output is not part of the language, but support is provided by a library. Most programs need to use `#include <iostream>`. You may also need `#include <fstream>` for file I/O, `#include <iomanip>` for greater control over formatting, and `#include <sstream>` for I/O to and from `string`s.

6.1 Simple I/O

The "C++ Summary" `http://www-h.eng.cam.ac.uk/help/mjg17/teach/CCsummary/node19.html` shows simple usage.

The following illustrates the use of `putback()` for an input stream. `istream`s also have an `ignore` function (`is.ignore(5)` would ignore the next 5 characters) and a `peek()` function which looks at the next character without removing it from the stream.

```
istream& eatwhite(istream& is)
{
char c;
  while(is.get(c)) {
    if (!isspace(c)) {
      is.putback(c);
      break;
    }
    return is;
  }
}
```

The following illustrates the use of I/O to and from files. Given the name of 2 files on the command line (which are available to `main` as C-style character arrays), it does a file copy.

```
void error(string p, string p2="")
{
 cerr<<p << ' ' << p2 << endl;
 std::exit(1);
}

int main (int argc, char* argv[])
{
 if (argc != 3) error("wrong number of arguments");

 std::ifstream from(argv[1]);
 if (!from) error("cannot open input file", string(argv[1]));

 std::ofstream to(argv[2]);
 if (!to) error("cannot open output file", string(argv[2]));

 char ch;
 while (from.get(ch)) to.put(ch);

 if (!from || !to) error("something strange happened");
}
```

You can force the screen/file output to be up-to-date by using

```
cout.flush()
```

or

```
cout << flush;
```

but this isn't usually necessary.

If you want to write or read raw (binary) data (rather than strings that represent numbers) you can use `write`, `read` and casts (section 11.1) to keep the compiler happy. The following program writes 100 random integers into a file then reads them back.

```
#include <iostream>
#include <fstream>
#include <cstdlib>  // to use rand
using namespace std;
```

```
int main()
{

 ofstream outfile("myresults",ios::out|ios::binary);
 if (outfile.good() == false) {
   cerr << "Cannot write to 'myresults'" << endl;
   exit(1);
 }

 int num;
 for (int i=0; i<100; i++){
   num=rand();
   outfile.write(reinterpret_cast<const char*>(&num), sizeof(num));
 }
 outfile.close();

 ifstream infile("myresults");
 if (infile.good() == false) {
   cerr << "Cannot open 'myresults'" << endl;
   exit(1);
 }

 int count=0;
 while(infile.read(reinterpret_cast<char*>(&num), sizeof(num))) {
   cout << num << endl;
   count++;
 }
 cout << count << " numbers read in" << endl;
 infile.close();
 return 0;
}
```

6.2 Formatting

The way that text and numbers are output can be controlled.

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  int i=10;
  cout << "i = " << i << " (default)\n" ;
  cout << hex << "i = " << i << " (hex)\n";

  double d = sqrt(7.0);
  cout << "d=sqrt(7.0)=" << d << endl;
  cout.precision(3);
  cout << "After cout.precision(3), d=" << d << endl;

  return 0;
}
```

There many other routines too, amongst them

```
cout.setf(ios_base::oct,ios_base::basefield); // set base to 8

const ios_base::fmtflags myopt = ios_base::left|ios_base::oct;
```

```
ios_base::fmtflags old_options = cout.flags(myopt); //set base to 8
// and alignment to left.

cout.precision(8); // set precision to 8
cout.setf(ios_base::scientific, ios_base::floatfield);

cout.width(4);  // output at least four characters
cout.fill('*'); // fill gaps with '*'
cin.noskipws(); // don't skip white space
```

6.3 Stream Iterators

Iterators can be used on streams, providing an elegant way to integrate I/O with container classes. You've seen them already in the Standard Library section. The following prints out "HelloWorld"

```
#include <iterator> // needed for ostream_iterator
...
ostream_iterator<string> oo (cout);
int main()
*oo = "Hello";
++oo;
*oo = "World";
```

6.4 Output of User-Defined types

One way to do this is by overloading the stream insertion operator. The following defines how complex numbers should be printed out -

```
ostream& operator<<(ostream&s, complex z) // returns ostream& so can be chained
{
  return s << '(' << z.real() << ',' << z.imag() << ')';
}
```

6.5 Input of User-Defined types

The following code reads in a complex number that's provided in the form (a,b) or (a)

```
istream& operator>>(istream&s, complex &a) // returns istream& so can be chained
{
 double re=0, im=0;
 char c=0;

 s>>c;
 if (c== '(') {
     s>>re >> c;
     if (c == ',') s >> im >> c;
     if (c != ')') s.clear(ios_base::badbit);

 }
 else {
   s.putback(c);
   s>> re;
 }

 if (s) a = complex(re,im);
 return s;
}
```

6.6 String streams

`stringstreams` are streams that are attached to a `string` rather than a file and letting you use the same syntax as file I/O. They are defined in `<sstream>`.

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main()
{
int i = 7;
string s1 = "He is ";
string s2 = " years old";

ostringstream ostring;
ostring << s1 << i << s2;

cout << "ostring =" << ostring.str() << endl;

return 0;
}
```

7 Performance

The quality of the compiler and libraries can greatly affect the speed of the resulting code, but there's a lot you can do to help.

- Read about the available optimiser options. For example, using '+O4' instead of no optimisation can speed up some code by orders of magnitude.
- Try not to use call-by-value for big objects. Unless you need to change the object, use a reference to a `const` value
- Some C++ features have compile time implications, others have run time, program size or debugging implications. For example, making a function virtual can add 15% to the cost of a function call whereas templates have no runtime implications. They do, however, make things more difficult for debuggers and compilers. The Standard Template Library, for example, avoids virtual functions for performance reasons.
- Inlined routines are faster, but they'll bloat the code. Note that a member function defined in the class declaration is automatically inline. In particular, note that an inline destructor can be inserted at each exit point of a function.
- Though Standard Library algorithms like `reverse` work on both `list` and `vector` containers, the choice of container is still important for performance.
- Though vectors can grow, it's more efficient to create them with their maximum size.
- Try to create and initialise a variable in one operation.

For more details, see "Faster C++ - `http://www-h.eng.cam.ac.uk/help/tpl/languages/C++/fasterC++.html`

8 Debugging

"Code Complete" by Steve McConnell (Microsoft Press, 1993) is in the CUED library. It has very useful sections on testing and debugging, and quotes results from investigations of software quality assurance. On p.567 it notes that the industry average for code production is 8-20 lines of *correct* code per day. On p.610 it notes that industry

average experience suggests that there are 15-50 errors per 1000 lines of delivered code. The recommendation is to design and code defensively - it saves time in the end.

C++ is a strongly-typed language with many features to help write safe code, but you still need to be cautious

- Let the compiler help you as much as possible! Many compilers generate warning messages if you ask them to.
- Assume when you're writing your program that it won't be bug-free. Write it so that you (and others) can easily debug it: make routines small and control flow simple.
- Concrete examples are easier to debug. You can make them into a template once they're working.
- Write a test suite (for you and future users)
- Don't get stuck at one level - zoom in and pan out. Your code should be understandable locally (parts should be as isolated from other parts as possible) and at the top level.
- Write "defensive" code. Always assume the worst.
- It's not uncommon for beginners to prove to themselves on paper that their broken code is correct and the compiler is wrong. Add `cout` statements to print out intermediate values and test assertions.
- If you're using floating point arithmetic, check divisions to see if you're dividing by zero.
- Check that you're not going off the end of arrays.
- To comment-out large chunks of code bracket it with

```
#if 0
...
#endif
```

See the Debugging handout `http://www-h.eng.cam.ac.uk/help/tpl/languages/debug/debug.html` for details.

9 Common Difficulties and Mistakes

Obscure Error Messages - small errors in a simple-looking line can produce complicated messages like the following

```
Error 226: "set1.cc", line 16 # No appropriate function found for call of
    'operator <<'. Last viable candidate was "ostream &ostream::operator
    <<(char)" ["/opt/aCC/include/iostream/iostream.h", line 509]. Argument of
    type 'class set<int,less<int>,allocator>' could not be converted to
    'char'.
     cout << "s1 = " << s1;
     ^^^^^^^^^^^^^^^^^^^^
```

The `^^^^` symbols show which part of the line is at fault. In this case the compiler can't narrow down the problem, so next read the first (easiest) part of the error message. There's something wrong with the use of `<<`. Remember that in C++ many functions can be invoked by an operator - it all depends on the number and type of operands. In this case `s1` (a set) isn't something that there's a matching function for - hence the error message. The compiler, trying to be helpful, gives the next best option - a function tied to `<<` that can deal with a `char`.

Another example: the line `double d = sqrt(4);` produces the following error because the integer given to `sqrt` has to be promoted to a real, but the compiler doesn't know what kind of real.

```
Error 225: "foo.cc", line 6 # Ambiguous overloaded function call; more than
    one acceptable function found. Two such functions that matched were "long
    double sqrt(long double)" ["/opt/aCC/include/cmath", line 107] and "float
    sqrt(float)" ["/opt/aCC/include/cmath", line 57].
```

Obscure declarations and definitions - C++ has fewer keywords than many languages but it makes up for this by having many meaning attached to a symbol. For example '`<`' is used in `#include` lines, `cout`, in template definitions as well as meaning 'less-than'. This makes deciphering declarations hard. `c++decl` can sometimes help.

Deriving from Container Classes - the container classes weren't designed to act as base classes: they don't have virtual functions. Use them as members of other classes instead.

Object creation - The first 2 lines below create a string variable `s`. The 3rd doesn't create an empty string - it declares a function `s` that returns a string.

```
String s;
String s("initial");
String s();
```

Input and newline characters - If you try to get a character then a string from the user with

```
cout << "Type a character: ";
cin >> ch;
cout << "Type a string: ";
cin.getline(str,20);
```

you're likely to get an empty string because the `>>` operator leaves the typed newline in the input stream. One way round this is to use `istream::ignore(INT_MAX, '\n');`.

Unrecommended features C++ is a flexible language, with support for many old C features than have been superceded. The following points are worth bearing in mind

- Think in terms of objects rather than functions. Groups of objects are easier to organise (using inheritance) than are groups of functions.
- Use references rather than pointers when you can.
- Create variables just before first use. In particular, create `for` loop index variables within the loop construction.
- Avoid pre-processor macros - `const`, `inline`, `template` and `namespace` replace most of `#define` usage, leading to stronger typing and better control of scope.

10 Program Development

Stroustrup writes that the last section of his book aims

> *to bridge the gap between would-be language-independent design and programming that is myopically focussed on details. Both ends of this spectrum have their place in a large project, but to avoid disaster and excessive cost, they must be part of a continuum of concerns and techniques.*

Programs vary considerably in their composition. I've seen these figures quoted

- Coding takes 10% of the total time. Debugging takes 50%.
- The Graphical User Interface is 80% of the code
- Error handling can be 50% of the code

Complexity is the enemy, so

- Divide and conquer.
- Use modules - namespaces or files (helps the optimiser too).

Don't re-invent the wheel

- Copy models
- Adapt existing parts
- When making new parts design them for re-use

10.1 Style

It helps if you decide upon a uniform style for writing code. It's common to suggest that for all except the most trivial classes it's wise to define

- a void constructor -
- a copy constructor - to create a new object using values from an existing one. Use call-by-reference to avoid infinite recursion.
- the assignment operator - that returns a reference.

And then there's the "the rule of three" - if you write any one of a copy constructor, copy assignment operator or destructor, then you will almost certainly need to write all three for your class to function properly.

Stanley Lippman in Dr.Dobb's Journal, October 99, (p.40) noted that unnecessary definition of these functions is likely in practise to make the code bigger and/or slower - he got a 40% speed improvement by removing all 3 from a piece of code, the compiler's default alternatives being more efficient.

When deriving classes

- if using public inheritance, most base class member functions should be declared `virtual` to ensure that the derived class customisations will override the base class behaviour even in a context where a base class object is expected.
- use virtual inheritance if you are not especially concerned about performance or if you might use multiple inheritance later.

10.2 Makefiles

If you have many source files you don't need to recompile them all if you only change one of them. By writing a `makefile` that describes how the executable is produced from the source files, the `make` command will do all the work for you. The following makefile says that `pgm` depends on two files `a.o` and `b.o`, and that they in turn depend on their corresponding source files (`a.cc` and `b.cc`) and a common file `incl.h`:

```
pgm: a.o b.o
    g++ a.o b.o -o pgm
a.o: incl.h a.cc
    g++ -c a.cc
b.o: incl.h b.cc
    g++ -c b.cc
```

Lines with a '`:`' are of the form

```
target : dependencies
```

`make` updates a target only if it's older than a file it depends on. The way that the target should be updated is described on the line following the dependency line (Note: this line needs to begin with a `TAB` character).

Here's a more complex example of a `makefile` for a program called `dtree`. First some variables are created and assigned. In this case typing '`make`' will attempt to recompile the `dtree` program (because the default target is the first target mentioned). If any of the object files it depends on are older than their corresponding source file, then these object files are recreated.

The targets needn't be programs. In this example, typing '`make clean`' will remove any files created during the compilation process.

```
# Makefile for dtree
DEFS =  -O2 -DSYSV
CFLAGS = $(DEFS) -O
LDFLAGS =
CC = g++
LIBS = -lmalloc -lXm -lXt -lX11 -lm

BINDIR = /usr/local/bin/X11
MANDIR = /usr/local/man/man1
```

```
OBJECTS_A = dtree.o Arc.o Graph.o  #using XmGraph

ARCH_FILES = dtree.1 dtree.cc Makefile Dtree Tree.h TreeP.h \
   dtree-i.h Tree.cc Arc.cc Arc.h ArcP.h Graph.cc Graph.h GraphP.h

dtree: $(OBJECTS_A)
        $(CC) -o dtree $(LDFLAGS) $(OBJECTS_A) $(LIBS)

Arc.o: Arc.cc
        $(CC) -c $(CFLAGS) Arc.cc

Graph.o: Graph.cc
        $(CC) -c $(CFLAGS) Graph.cc

dtree.o: dtree.cc
        $(CC) -o dtree.o -c $(CFLAGS) -DTREE dtree.cc

install: dtree dtree.1
        cp dtree $(BINDIR)
        cp dtree.1 $(MANDIR)

clean:
        rm -f dtree *.o core tags a.out
```

11 Specialist Areas

11.1 Casts

C++ is fairly strict about converting from one type to another. You can use C-style casting but it's not recommended - instances are hard to find in the code, and it's less type-safe. Try to use the least dangerous (most specific) casting alternative from those below

const_cast - used to remove the `const` qualifier.

dynamic_cast - does careful, type-sensitive casting at run time. Can't deal with `void*`.

reinterpret_cast - produces something that has the same bit pattern as the original.

static_cast - doesn't examine the object it casts from at run time. This is fast if it's safe - and the compiler should tell you if it isn't safe. Can deal with `void*`. `static_cast` can be used to reverse any of the implicit conversions (like int to float).

To convert from a `const void *` to a `File_entry**` (a common kind of thing to do in C) I had to do

```
File_entry**fe2= reinterpret_cast<File_entry**>(const_cast<void*>(entry2));
```

11.2 Limits

Use `<climits>` to access information on the system's limits. E.g. to check if a `long` variable can fit into a `short` you could try

```
long i;
if (i<numeric_limits<short>::min() || numeric_limits<short>::max() <i)
     cout << i << "cannot fit into a short"
```

11.3 Exceptions

In C every call had to be checked for error values - which could double the code size. C++ exceptions are an alternative to traditional techniques when they are insufficient, inelegant, and error-prone. Three keywords are involved

try - specifies an area of code where exceptions will operate

catch - deals with the exceptional situation produced in the previous `try` clause.

throw - causes an exceptional situation

When an exception is 'thrown' it will be 'caught' by the local "catch" clause if one exists, otherwise it will be passed up through the call hierarchy until a suitable `catch` clause is found. The default response to an exception is to terminate.

There are many types of exception - they form a class hierarchy of their own. Here just a few will be introduced.

```
try {
// code
}
// catch a standard exception
catch(std:exception& e){
 // order of the catch clauses matters; they're tried
 // in the given order
}
// catch all the other types
catch(...) {
// cleanup
 throw;   // throw the exception up the hierarchy
}
```

Here's an example that sooner or later will produce an exception.

```
try{
  for(;;)
    new char[10000];
}
catch(bad_alloc) {
  cerr << "No memory left";
}
```

An exception example is at `http://www-h.eng.cam.ac.uk/help/tpl/languages/C++/examples/exception.cc`

You can override the routine that handles problems encountered by the `new` operator

```
set_new_handler(&my_new_handler);
```

For safety, you can declare which exceptions a function should be able to throw

```
int f() throw (std::bad_alloc)
// f may only throw bad_alloc
```

A useful exception is `out_of_range` (header `<stdexcept>`), which is thrown by at() and by `bitset<>::operator[]()`.

Exception handling code can slow a program down even when no exceptions happen. The code size will also increase.

11.4 Maths

If you're using any of the maths routines (`sqrt`, `sin`, etc) remember that you'll need to include `cmath` to declare the routines.

Before you start writing much maths-related code, check to see that it hasn't all been done before. Many maths routines, including routines that offer arbitrary precision are available from netlib - `http://www.hensa.ac.uk/netlib/master/readme.html`. Other resources are listed on CUED's maths page - `http://www-h.eng.cam.ac.uk/help/tpl/maths.html`. Before you do any heavy computation, especially with real numbers, I suggest that you browse through a Numerical Analysis book. For example, "4.0/3.0 - 1.0/3.0 - 1.0" is unlikely to produce the same answer as "4.0/3.0 -1.0 - 1.0/3.0". Operations to avoid include

- Finding the difference between very similar numbers (if you're summating an alternate sign series, add all the positive terms together and all the negative terms together, then combine the two).

- Dividing by a very small number (try to change the order of operations so that this doesn't happen).
- Multiplying by a very big number.

Common problems that you might face are :-

Testing for equality :- Real numbers are handled in ways that don't guarantee expressions to yield exact results. Just as in base 10 to 2 decimal places, $3 * (10/3)$ doesn't equal 10, so computer arithmetic has its limitations. It's especially risky to test for exact equality. Better is to use something like

```
d = max(1.0, fabs(a), fabs(b))
```

and then test `fabs(a - b) / d` against a relative error margin. Useful constants in `<climits>` are `FLT_EPSILON`, `DBL_EPSILON`, and `LDBL_EPSILON`, defined to be the smallest numbers such that

```
1.0f + FLT_EPSILON  != 1.0f
1.0  + DBL_EPSILON  != 1.0
1.0L + LDBL_EPSILON != 1.0L
```

respectively.

Avoiding over- and underflow :- You can test the operands before performing an operation in order to check whether the operation would work. You should always avoid dividing by zero. For other checks, split up the numbers into fractional and exponent part using the `frexp()` and `ldexp()` library functions and compare the resulting values against `HUGE` (all in `<cmath>`).

Floats and Doubles :- You can use the information in `<limits>` or use `sizeof(double)`, etc to compare the size of float, double and long double.

- Keep in mind that the `double` representation does not necessarily increase the *precision* over `float`. Actually, in most implementations the worst-case precision decreases but the *range* increases.
- Do not use `double` or `long double` unnecessarily since there may a large performance penalty. Furthermore, there is no point in using higher precision if the additional bits which will be computed are garbage anyway. The precision one needs depends mostly on the precision of the input data and the numerical method used.

```
#include <iostream>
using namespace std;

int main(int argc, char* argv[])
{
float f1;
double d1;
long double ld1;

f1=d1=ld1=123.45678987654321;
cout.precision(50);
cout << "f1=" << f1 << ", f1**2=" << f1*f1 << endl;
cout << "d1=" << d1 << ", d1**2=" << d1*d1 << endl;
cout << "ld1=" << ld1 << ", ld1**2=" << ld1*ld1 << endl;

return 0;
}
```

Infinity :- The IEEE standard for floating-point maths recommends a set of functions to be made available. Among these are functions to classify a value as `NaN`, `Infinity`, `Zero`, `Denormalized`, `Normalized`, and so on. Most implementations provide this functionality, although there are no standard names for the functions. Such implementations often provide predefined identifiers (such as `_NaN`, `_Infinity`, etc) to allow you to generate these values.

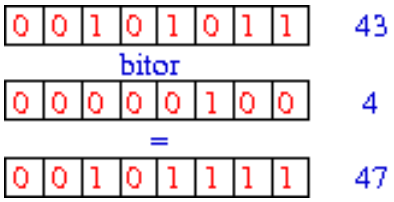
If `x` is a floating point variable, then `(x != x)` will be TRUE if and only if `x` has the value `NaN`. Some C++ implementations claim to be IEEE 748 conformant, but if you try the `(x!=x)` test above with `x` being a `NaN`, you'll find that they aren't.

An increasing amount of maths work is being done with C++. One development is the Template Numerical Toolkit - `http://math.nist.gov/tnt/index.html` built around the Standard Library.

11.5 Hardware Interfacing: bit operations and explicit addresses

If you're interfacing with hardware and need to operate on bits you can use `bitset` but you may prefer to use C++'s low-level operations.

Setting a bit :- Suppose you wanted to set bit 4 of `i` (a `char`, say) to `1`. First you need to create a mask that has a `1` in the 4th bit and `0` elsewhere by doing '`1<<4`' which shifts all the bits of `1` left 4 bits. Then you need to do a bit-wise `OR` using '`i = i bitor (1L<<4)`'.



Unsetting a bit :- Suppose you wanted to set bit 4 of `i` (a `char`, say) to `0`. First you need to create a mask that has a `0` in the 4th bit and `1` elsewhere by doing '`1<<4`' then inverting the bits using the `compl` operator. Then you need to do a bit-wise `AND` using the `bitand` operator. The whole operation is '`i =i bitand compl(1<<4)`' which can be contracted to '`i &= ~(1<<4)`'.

Suppose because of hardware considerations the `i` variable needed to be the value of memory location 2000. The following code will do what you want.

```
long *memory_pointer = reinterpret_cast<long*>(2000);
long i = *memory_pointer;
```

11.6 Calling Python from C++

The details of how to do this are system dependent. The following example worked on the CUED Central System in April 2016.

Create a file called `multiplycode.py` containing

```
def multiply(a,b):
    print ("Will compute", a, "times", b)
    c = 0
    for i in range(0, a):
        c = c + b
    return c
```

This is what we're going to run from the following C++ file. Call it `call.cc` and create it in the same folder as `multiplycode.py` is in

```
#include <Python.h>

int main(int argc, char *argv[])
{
    PyObject *pName, *pModule, *pDict, *pFunc;
    PyObject *pArgs, *pValue;
    int i;

    if (argc < 3) {
        fprintf(stderr,"Usage: call pythonfile funcname [args]\n");
        return 1;
```

```
    }

    Py_Initialize();
    pName = PyUnicode_DecodeFSDefault(argv[1]);
    /* Error checking of pName left out */

    pModule = PyImport_Import(pName);
    Py_DECREF(pName);

    if (pModule != NULL) {
        pFunc = PyObject_GetAttrString(pModule, argv[2]);
        /* pFunc is a new reference */

        if (pFunc && PyCallable_Check(pFunc)) {
            pArgs = PyTuple_New(argc - 3);
            for (i = 0; i < argc - 3; ++i) {
                pValue = PyLong_FromLong(atoi(argv[i + 3]));
                if (!pValue) {
                    Py_DECREF(pArgs);
                    Py_DECREF(pModule);
                    fprintf(stderr, "Cannot convert argument\n");
                    return 1;
                }
                /* pValue reference stolen here: */
                PyTuple_SetItem(pArgs, i, pValue);
            }
            pValue = PyObject_CallObject(pFunc, pArgs);
            Py_DECREF(pArgs);
            if (pValue != NULL) {
                printf("Result of call: %ld\n", PyLong_AsLong(pValue));
                Py_DECREF(pValue);
            }
            else {
                Py_DECREF(pFunc);
                Py_DECREF(pModule);
                PyErr_Print();
                fprintf(stderr,"Call failed\n");
                return 1;
            }
        }
        else {
            if (PyErr_Occurred())
                PyErr_Print();
            fprintf(stderr, "Cannot find function \"%s\"\n", argv[2]);
        }
        Py_XDECREF(pFunc);
        Py_DECREF(pModule);
    }
    else {
        PyErr_Print();
        fprintf(stderr, "Failed to load \"%s\"\n", argv[1]);
        return 1;
    }
    Py_Finalize();
    return 0;
}
```

Compile this using the following (which is one long line)

```
g++ $(/usr/local/apps/anaconda3/bin/python3-config --includes) call.cc -o call
        $(/usr/local/apps/anaconda3/bin/python3-config --ldflags)
```

Now run

```
PYTHONPATH=. ./call multiplycode multiply 3 2
```

You should get output of

```
Will compute 3 times 2
Result of call: 6
```

12 More on Classes

To make best use of Classes it's sometimes useful to redefine the operators as well as define member functions. For instance, if you were going to invent a class to deal with strings, it would be nice for '+' to join strings together. Or if you wanted to add range-checking to arrays, you'd have to modify the `[ ]` behaviour. Below are some examples of overloading, but first more on the mechanism of inheritance.

12.1 Virtual members

As mentioned earlier, it's easy to redefine a base class's function in a derived class, but to fully exploit polymorphism (the ability to deal with many types transparently), a new concept needs to be introduced.

Suppose that we were designing a graphics editor. We might have a base class Shape and various derived classes Triangle, Rectangle, etc, each with their own `print` member function. We'd like to collect these objects together into groups sometimes. An array of pointers to the objects would be appropriate, but what sort of pointers could we use? Fortunately, it's possible to assign a derived class pointer to a base class pointer, so the following is possible, creating an array of pointers to the base class.

```
Triangle t1, t2;
Rectangle r1, r2;

typedef ShapePtr Shape*;
vector<ShapePtr> shapes(4);
shapes[0]=&t1;
shapes[1]=&r1;
shapes[2]=&t2;
shapes[3]=&r2;
```

Then we could draw all the shapes in the group by calling the `print` function of each element. But there's a problem - because the elements are pointers to Shape, it's Shape's `print` function which is called, rather than the appropriate derived object's function.

The solution to this is to define `print` in the base class as a virtual function by having something like

```
virtual void print();
```

which lets us deal with the `shapes` array without having to worry about the real type of each component; the derived object's `print` function will always be called.

12.2 Abstract Classes

Some classes represent abstract concepts for which objects cannot exist. The functions *have* to be defined in the derived classes. Such functions are called pure virtual functions and are denoted by the following notation

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

in the base class. A class with one or more pure virtual functions is an Abstract Class.

12.3 Redefining operators

If you need to redefine operators remember to redefine then consistently (if you redefine '`+`' don't forget '`+=`') and remember that you can't change precedence order. The default assignment operator does a simple copy of members, which might not be what you want.

Examples of redefining operators are in most books. Some things to remember when redefining assignment operators are

- that reference parameters help to overload operators efficiently (objects aren't copied) while keeping their use intuitive (pointers would require the user to supply addresses of objects).
- to deal with situations where a variable is assigned to itself. A pointer/reference to the object on the right of the = will be given to the function explicitly. A pointer to the object that the function is a member of is supplied implicitly to all non-static member functions. It's called `this`, so if `this` equals the supplied pointer, you'd usually want to do nothing.
- to return a `const`. This catches errors like `(i+j)=k`
- to return `*this` so that assignments can be chained (i.e. `i=j=k` becomes possible)

12.4 A class definition example

The following example highlights some of the more subtle issues associated with Constructors and Destructors.

It's the job of Constructors to allocate the resources required when new objects are created. If these are created using `new` then the Destructor should free the resources to stop memory being wasted. For example, if we want to store information about criminals we might define an object as follows

```
class criminal {
 string name;
 char* fingerprint;
 int fingerprint_size;
}
```

where `fingerprint` points to an image. The memory (if any) allocated for the image would be freed by the destructor.

assignment operator

That works fine until the following kind of situation arises

```
void routine (criminal a)
{
  criminal b;
  b=a;
  ...
}
```

The assignment `b=a` copies the bytes of `a` to `b`, so the `fingerprint` field of both objects will be the same, pointing to the same memory. At the end of this routine `criminal`'s destructor will be called for `b`, which will `delete` the memory that `b.fingerprint` points to, which unfortunately is the same memory as `a.fingerprint` points to.

To cure this we need to redefine the assignment operator so that the new object has its own copy of the resources.

```
void criminal::operator=(criminal const &b)
{
 name=b.name;
 fingerprint_size=b.fingerprint_size;
 delete fingerprint;
 fingerprint = new char[fingerprint_size];
 for (int i=0;i<fingerprint_size;i++)
    fingerprint[i]=b.fingerprint[i];
}
```

Note that space used by an existing fingerprint is freed first before a new fingerprint image is created - memory would be wasted otherwise.

this

But problems remain. If the programmer writes `a=a`, the `delete` command above frees `a.fingerprint`, which we don't want to free. We can get round this by making use of the `this` variable, which is automatically set to be the address of the current object. If the code above is bracketted by `if (this != &b) { ... }` then it's safe.

There's another improvement that we can make. So that assignments like `a=b=c` can work, it's better not to return `void`.

```
criminal const& criminal::operator=(criminal const &b)
{
 if (this != &b) {
   name=b.name;
   fingerprint_size=b.fingerprint_size;
   delete fingerprint;
   fingerprint = new char[fingerprint_size];
   for (int i=0;i<fingerprint_size;i++)
      fingerprint[i]=b.fingerprint[i];
 }
 return *this;
}
```

copy constructor

But there's still a problem! Suppose the programmer writes `criminal a=b;`. `a` is being created, so a constructor is called - not an assignment. Thus we need to write another constructor routine using similar ideas to those used in assignment. Note that in this case resources don't need deleting, and that `this` needn't be checked (because `criminal a=a;` is illegal).

Putting all these ideas together, and sharing code where possible we get

```
class criminal {
 string name;
 char* fingerprint;
 int fingerprint_size;
 void copy (criminal const &b);
 void destroy();
 ~criminal();
 criminal();
 criminal(criminal const &b);
 criminal const& operator=(criminal const &b);
};

void criminal::copy (criminal const &b)
{
 name=b.name;
 fingerprint_size=b.fingerprint_size;
 fingerprint = new char[fingerprint_size];
 for (int i=0;i<fingerprint_size;i++)
    fingerprint[i]=b.fingerprint[i];
};

void criminal::destroy()
{
 delete fingerprint;
};

// Assignment Operator
criminal const& criminal::operator=(criminal const &b)
{
 if (this != &b) {
  destroy();
  copy(b);
 }
```

```
 return *this;
};

// Default Constructor
criminal::criminal()
{
  fingerprint=0;
  fingerprint_size=0;
}

// Copy Constructor
criminal::criminal(criminal const &b)
{
  copy(b);
};

// Destructor
criminal::~criminal()
{
  destroy();
};
```

This seems like a lot of work, but it's necessary if objects use pointers to resources. You might think that by avoiding the use of commands like `a=a` in your own code, you can save yourself the trouble of writing these extra routines, but sometimes the problem situations aren't obvious. For instance, the copy constructor is called when an object is returned or used as an argument to a routine. Better safe than sorry.

12.5 Redefining [ ]

`vector` doesn't have range-checking by default, but if you use the member function `vector::at()`, `out_of_range` exceptions can be trapped, so you can add `out_of_range` checking by redefining `[ ]` in a class derived from `vector`, and using `vector`'s constructors -

```
template<class T> class Vec: public vector<T> {
public:
Vec() : vector<T>() {}
Vec(int s) : vector<T>(s) {}

T& operator[] (int i) {return at(i);}
const T& operator[] (int i) const {return at(i);}
}
```

The following more complicated example creates a mapping (relating a string to a number) without using the Standard Library's `map` facility. It does a word-frequency count of the text given to it.

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

// The Assoc class contains a vector of string-int Pairs.
class Assoc {
  struct Pair {
    string name;
    int val;
    // The following line is a constructor for Pair
    Pair (string n="", int v=0): name(n), val(v) {}
  };

  // create a vector of the Pairs we've just created
  vector <Pair> vec;
```

```
public:
  // redefine []
  int& operator[] (const string&);
  // a member function to print the vector out
  void print_all() const;
};


// This redefines [] so that a ref to the value corresponding to the
// string is returned if it exists. If it doesn't exist, an entry
// is created and the value returned.
int& Assoc::operator[] (const string& s)
{
// The next line's creating an appropriate iterator
// for the vector of Pairs
  for (vector<Pair>::iterator p=vec.begin(); p!=vec.end(); ++p)
     if (s == p->name)
        return p->val;

  vec.push_back(Pair(s,0));
  return vec.back().val;
}

void Assoc::print_all() const
{
for(vector<Pair>::const_iterator p=vec.begin(); p!=vec.end(); ++p)
   cout << p->name<<": " << p->val << endl;
}

int main()
{
string buf;
Assoc vec;
  cout << "Type in some strings then press CTRL-D to end input\n";
  while(cin>>buf) vec[buf]++;
  vec.print_all();
}
```

12.6 Redefining ()

This can be used to provide the usual function call syntax for objects that in some way behave like functions ("function objects" or "functors"). These are used by the Standard Library.

12.7 Redefining ->

Smart pointers are pointers that do something extra (e.g. increment a counter) when they are used.

12.8 Exercises

1. Write a program to demonstrate the effect of making a routine `virtual`.
2. Write a class to deal with dates. Have fields for the day, month and year, a print function, and a way to add an integer to a date to get a future date.
3. Write a class to deal with rational fractions - addition, multiplication, input and output.

Many more exercises are in "C++ objects, containers and maps" - `http://www-h.eng.cam.ac.uk/help/tpl/languages/C++/1BComputingOO/` and "Bridge Exrercise" - `http://www-h.eng.cam.ac.uk/help/tpl/languages/C++/bridgeexercise.php`

13 References

- "The C++ Programming Language" (fourth edition, over 1300 pages), Bjarne Stroustrup, 2013. Also known as 'the bible'. See his homepage `http://www.research.att.com/~bs/homepage.html`
- "C++: How to Program", (9th edition, 1080 pages), Deitel & Deitel, Prentice Hall, 2014. Used by undergraduates at CUED.
- C++ FAQ - `https://isocpp.org/faq`
- CUED's C++ help page - `http://www-h.eng.cam.ac.uk/help/tpl/languages/C++.html`
- A Transition Guide: Python to C++ - David Letscher `http://dehn.slu.edu/courses/spring08/180/transition.pdf`