Lecture Notes on C++ for Java Programmers

Jonathan G. Campbell
Department of Computing,
Letterkenny Institute of Technology,
Co. Donegal, Ireland.

email: jonathan dot campbell (at) gmail.com, jonathan.campbell@lyit.ie
URL:http://www.jgcampbell.com/cpp4jp/cpp4jp.pdf
Report No: jc/09/0011/r
Revision 3.0, minor edits, new references, new later chapters, 2009-08-12

12th August 2009

Contents

Chapter 1

Introduction

1.1 Scope

These are course notes for a lecture series on C++ for Java Programmers, given in second year of of the course BSc in Computer Games Programming, School of Computing, Letterkenny Institute of Technology.

This course originated in a course on Object-oriented Programming using C++ developed at University of Ulster during 1997—1999. In revising the course during 2006–2009, I have done my best to modernise the C++ to include more the *standard library* (STL) and other additions to C++ that have occurred since 1999. In that respect, the course is strongly influenced by (Koenig & Moo 2000) and (Stroustrup 2009).

We note that the *C++ standard library* allows one to program at a much higher level. Roughly translated, *higher level* means that program statements: (a) do a lot more than the equivalent *low level* statement; (b) are less error prone; (c) are easier for humans to read and understand. Our chief encounter with standard library features will be in the use of *collection*s like `vector` and with *algorithms* with which one can operate on those collections. Java programmers who have used `List` and `ListArray` and algorithms such as `sort` will have no difficulty becoming familiar with this aspect of C++.

The earlier course contained a few chapters on the principles of *object-oriented programming* (OOP); I'm leaving those out, preferring to believe that students of this course will already know that, or know nothing other than OOP, or will get a dose of OOP theory from some other lecturer. If you participated in the Games Programming 1 team project, you have encountered nearly as many principles of OOP as you will ever need.

1.2 Recommended Reading

1.2.1 Programming and Object-oriented Programming through C++

If I had to recommend a *teach-yourself modern C++* book, I'd go for either (Stroustrup 2009) or (Koenig & Moo 2000). (Glassborow 2006) is also useful. (Dawson 2004) is a good introduction to the basics from a games point of view,

If you need a reference on C++, see (Stroustrup 1997a), (Lippman 2005), and (Lischner 2003). The C++ FAQ (Cline, Lomow & Girou 1999a) is useful, and there is an online version (use Google to find it).

I have learned a lot of what I know about OOP and C++ from (Budd 1997a). Budd produces very fine books. I can also recommended his book on Java (Budd 1999b) and his book on general object-orientation (Budd 1997b).

Budd's book on conversion from C++ to Java (Budd 1999a) is useful for this course.

You will already be familiar with Horstmann's book Java Concepts (Horstmann 2005) (or Big Java which is the same with four chapters added). Horstmann's (with Budd) Big C++ (Horstmann & Budd 2005) is a fine book.

We note Horstmann's handy website (Horstmann accessed 2006-08-28b) provides another good reference for those who need to move between C++ and Java; and Horstmann's C++ Pitfalls website at (Horstmann accessed 5006-08-28a), though some of the items paint C++ is a very bad light.

There are very useful guidelines in: (Meyers 2005) and (Meyers 1996); a companion book (Meyers 2001) covers now the C++ standard library.

Regarding the standard library and algorithms, I note that we have a third year module on *Algorithms and Data Structures for Games Programmers* (Campbell 2009); two of the chapters at the end of these notes overlap with the beginning chapters of the notes for that module.

Standard Library (STL) When I'm using the *C++ Standard Library (STL)* I always have (Reese 2007), (Josuttis 1999), and (Lischner 2003) at my right hand.

Other Books Other top class C++ books that I regularly use are (in no particular order): (Sutter & Alexandrescu 2005) (Dewhurst 2005) (Wilson 2004) (Romanik & Muntz 2003) (Sutter 1999) (Sutter 2002) (Sutter 2004) (Dewhurst 2003) (Josuttis 1999) (Eckel 2000) (Eckel 2003).

If you want fancy (general programming) ideas for a final year project and to see the way C++ and OOP is headed, see: (Alexandrescu 2001) and (Czarnecki & Eisenecker 2000).

1.2.2 Games Software Engineering

The following cover software engineering and design principles applied to games: (McShaffry 2005) (Dickheiser 2007) (O'Luanaigh 2006).

1.2.3 General Software Engineering

Apart from the good advice in the programming books, see (McConnell 2004) and (Maguire 1993).

1.2.4 Design Patterns

Software *design patterns* are important. Horstmann's (Horstmann 2006) and (Freeman & Freeman 2004) are good, even though the examples are in Java. See also (Budd 1999b, chapter 5), the original *gang-of-four* book (Gamma, Helm, Johnson & Vlissides 1995) and (Vlissides 1998).

1.2.5 The C Programming Language

When we come to OpenGL we will sometimes write C programs; C and C++ *are different*. The way we will do it is learn C++ first and then pint out the major differences. (Kernighan & Ritchie 1988) is the bible of C; (Harbison & Steele 2005) is a good reference.

1.2.6 Games Programming using C++

The following are as much on principles and software engineering as on construction of specific games and game components: (McShaffry 2005), (O'Luanaigh 2006) and (Dickheiser 2007).

(Dawson 2004) would be a good way to learn basic C++ (i.e. just plain text programming) before getting into the difficulties of video games.

(Penton 2003) is a good way to learn simple video game programming; at the end of this course we will work on a simple 2D sprite game from that book.

1.2.7 Cross Platform Windows Programming

wxWindows (Smart & Csomor 2005) and Qt (Blanchette & Summerfield 2008); (Blanchette & Summerfield 2008) contains a nice chapter on using OpenGL (the API that we use for our graphics modules) in a Qt framework.

1.2.8 Specification and Correctness

For future reference, I note that specification and correctness are important issues. Meyer uses the *contract metaphor* — Design by Contract (Meyer 1996); see also (Mitchell & McKim 2002), (Tennent 2002) and (Fitzgerald & Larsen 1998).

1.3 Plan for the course

The first thing to be done is get comfortable with the basics of C++ and with whatever C++ compiler we choose; we will probably be Visual Studio Express, see chapter 2 and we will use the examples in chapter 2.

We will have a quick look at designing procedural programs, see Appendix A.

Then we will look at some more details of C++ up to programming simples classes; then more complex classes: a dynamic array and a linked list.

Finally we will look at a C++ version of the game software we finished off with last year, for example Sprites and Animations — just to see how C++ handles objects, rather than develop real game code.

Chapter 2

Tutorial Introduction to C++

This chapter is based on the first chapter of Kernighan & Ritchie's book on C (Kernighan & Ritchie 1988), but with the code changed to C++. In this chapter and the next few, you will be exposed to the basic syntax of C++ – mostly the underlying imperative language features.

In the early days, most programmers had learned C before they came to C++, hence, it was often assumed that the best way to learn C++ was to first learn C. It is fairly clear now that this is not ideal; indeed, there is reason to believe that object-oriented programming can be quite difficult for experience *imperative language* programmers to grasp. Moreover, there are difficult parts of C that can be avoided until much later in C++ – and if proper design is used, they can be fairly well hidden in localised parts of the code.

2.1 Get Started

Objectives

- Get used to editor, compiler, linker etc. In our case the Visual Studio Express IDE (free) or Visual Studio Professional.
- Get a simple C++ program running.
- Dissect this simple C++ program.

All the programs in these notes and programs mentioned in practicals and assignments, plus other bits & pieces will be available in my public folder. For example, programs from this chapter will be in `P:/cpp4jp/progs/ch02`.

2.1.1 Your First C++ Program

The program `hello.cpp` is a C++ program that prints a simple message on the screen.

```
//----- hello.cpp ---------------------------------
//Your first C++ Program illustrating stream output.
//-------------------------------------------------
#include <iostream>
/* this is a begin-end comment*/
int main()
{
  std::cout << "Hello from C++.\n";
  std::cout << "Hello from C++." << std::endl;
  return 0;
}
```

Dissection of hello.cpp

1. Every program must have a function called `main`, execution starts there. `main` may call other functions. As with Java or any other high-level language, these functions can be taken from one of the following sources:

   - As written by the programmer in the same file as `main`.
   - As written by the programmer in separate file(s).
   - Called from predefined libraries.

2. C++ allows you to create programs *without* any *class*; thus the function `main` above does not need to be enclosed in a class called `Hello`;

3. C++ takes the view that `main` is called by the operating system; the operating system may pass arguments to `main`, as well as receive return values (e.g. `return 0;`); however, above, we choose to make `main` take no arguments – either `main(), or \verb`main(void)+ means 'takes no arguments. Note, in C, the explicit `void` is essential if that is what you want.

4. `/*... */` is a comment that has start and end delimiters. The C++ standard says that comments cannot be nested. Also, comments like these *must* be terminated explicitly, i.e. newline does not terminate them – this is a common source of compiler errors, and, for the unwary, can be very difficult to trace.

5. `//` comments end at end of line.

6. `iostream` is the *header* for the hierarchy of stream classes which handle buffered and unbuffered I-O for files and devices. `cout` is an object, corresponding to the standard output stream.

7. Although you `#include <iostream>` which contains declarations of `cout` and `endl`, these are contained in what is called a `namespace`; that namespace is `std`, for *standard library*.

8. A `namespace` is a bit like a Java `package`, but not completely.

9. If you want to avoid using the `std::` qualifier, you can insert a `using namespace std;` directive at the top of your program. `using namespace std;` has some similarities to a Java `import`.

10. `#includes` like `#include <iostream>` occur rather like Java `imports`, but, deep down, they are quite different.

11. We'll not go into the full details of `#include` and `namespace` here; they full story will eventually become clear.

12. The topic we are touching on here, but avoiding for the meantime, is called **scope**; we'll address that in detail in a later chapter (chapter 5); essentially, the *scope* of a variable or of a function is the parts of a program where the name of the variable can be used (i.e. is meaningful). Because `std` is a `namespace` all the variables and functions in it are hidden to the outside world, except if one uses the qualifier `std::`, or gives the overall directive `using namespace std`.

13. So the following program is equivalent to the previous `hello.cpp`.

```
//----- hello.cpp ----------------------------------
#include <iostream>
using namespace std; // brings whole std namespace into scope
int main()
{
  cout << "Hello from C++.\n";
  cout << "Hello from C++." << endl;
  return 0;
}
```

14. C++ allows you to write functions as *operators*; so, for example, you can write your own + operator instead of a function called `plus`.

15. The operator << is one such operator; note, it is *nothing* to do with bit shifting; the bit shifting operator has the same *name* but it does something very different; the program context allows the compiler to select between operators of the same name — as context allows resolution between functions of the same name.

16. The operator << (call it **put to**) takes *the value that follows it* (the string) and *puts it to the object* `cout`.

17. Recall Java's `System.out.print`.

    ```
    cout << "Hello from C++.\n";
    ```

    is very similar to

    ```
    System.out.print("Hello from C++.\n");
    ```

    where `cout` is equivalent to `System.out` and the method (function) `print` is equivalent to (the operator) <<.

18. Pronounce << as *put to*.

19. The two output lines are equivalent, except that `endl` ensures that the stream buffer is flushed.

20. The default extension for C++ program source files is `.cpp`; you will also see `.cc` used.

21. Header files, e.g. `iostream` contain function declarations and the like.

22. `#include <filename>` simply inserts the contents of *filename*, replacing the `include` pre-processor statement.

23. `<filename>` tells the pre-processor to search for *filename* in a system directory.

24. `"filename"` tells the pre-processor to search in the current directory, where this source file is.

25. The pre-processor acts on the source file before the compiler.

26. Typically, one has a `xxx.h` file which contains the declarations for the `xxx.cpp` file, which contains the executable code (implementation).

27. The `xxx.h` file is `included`. It is not compiled separately, though, obviously, it is compiled as part of the program into which it is included.

28. The `xxx.cpp` file is compiled, and its object code linked.

29. For your own sanity the heading comment should include:

    - Name of the program – to correspond to the filename,
    - Authors name – even if copied!
    - Date.
    - Brief indication of function and purpose of the program, e.g. assignment number, or project, and what it does.

30. (You already know all this.) Program layout is very important; I suggest two spaces indentation for each block; I suggest that you avoid tabs because the width of tabs are different according to different editors and printers. Certainly avoid having the program pressed up against the right margin, with all the white space on the left hand side,

31. In the definition of a function, `()` encloses the argument list; in this case `main` expects no arguments; `(void)` denotes *takes no arguments*, or, as mentioned, empty brackets, `()` signify the same thing..

32. However, empty parenthesis `()` is taken to signify the same thing. Beware, not the case in C!.

33. `\n` represents a single *new-line* character.

34. `\` is an *escape* character – announces a control character.

35. Other control characters are:

    - `\t` *tab*.

- `\a` *alert* – beep, bell.
- `\"` *double-quote*.
- `\r` *carriage-return*.
- `\\` backslash itself.

36. `return 0;` As mentioned above, in C++, programs can return values to the operating system; in UNIX, `0` signifies that the program terminated normally; other values indicate error conditions. This is useful if you intend putting your program in a UNIX shell script; likewise DOS, Windows `.bat` files.

Exercise 1.

This may look trivial and boring, but you *must* complete it before you proceed; doing so will advance you significantly far up the C++ learning-curve. In fact, when you need to do C programming on an unfamiliar system, it is always a good idea to get `hello.cpp` working first — just to make sure that you and the system are operating on the same wavelength!

- Type in the program `hello.cpp`;
- Compile it;
- Build (link) it; notice: *link*, this is something new in C++; we'll discuss that, and the related differences between C++ and Java, in a later chapter.
- Or, compile & build together;

I think we will decide to use the Microsoft Visual Studio Express compiler; Microsoft Visual Studio Professional will also be available, but the advantage of Express is that it is free and therefore available to those of you who have your own computers.

I'll give more detailed instructions in the handouts for the first few practicals.

I use Linux and so use the GNU C++ compiler. All programs mentioned in this course will be compilable on any system.

Safety and Warnings Most compilers issue both *error* messages and *warning* messages.

Error signifies a serious flaw in the source code, which the compiler cannot circumvent. In the event of any *error*, the compiler can produce no usable object code.

Warning, as the name suggests, is the compilers way of saying *are you sure about this*; examples are: variables defined but never used (fairly benign), type conversions that look a bit flaky, etc.

I maintain that C and C++ compiler warnings must never be ignored, and insist that students heed this principle, e.g. I refuse to help to check faulty code until the code compiles without warnings.

The good news is that C++ has much stricter type checking than C, so that many warnings in C, become errors in C++.

2.2 Variables and Arithmetic

Program `ptog.cpp` shows some simple arithmetic, involving a function, and a `while` loop.

```
//----- ptog.cpp ----------------------------
//Convert pounds to grams
//-------------------------------------------
#include <iostream>
using namespace std;

const double pToG = 453.592; // const == Java final

double convert(double p)
{
  return p*pToG;
}

int main()
{
  int pds = 0; //good idea to always initialise variables

  cout << "Input pounds: "<< endl;
  //really, we should handle input more carefully than this
  //if you type a number with a fractional part, e.g. 2.5
  //you will get an error and odd things will happen
  //we'll sort out this later
  cin >> pds;
  while(pds > 0){
     double g = convert(pds);
     int gint = int(g); // could have used simply gint = g;
     cout << "\nWeight = "<< gint << " grams."<< endl;
     double k = g/1000;
     cout << "\nWeight = "<< k << " Kg."<< endl;
     cout << "Input pounds: "<< endl;
     cin >> pds;
  }
  return 0;
}
```

Dissection of ptog.cpp

1. All variables must be declared. They may be declared anywhere – before they are used. They stay *in scope* until the end of the function or block `{ ... }` in which they were declared.
2. In C declaration can be done only at the beginning of a function or block.
3. Some programmers retain the C style (declarations at beginning); however, when large *objects* are involved, which may use large complex constructors, declaration at the beginning may

avoid duplication of effort because, at the beginning, appropriate initialisation may not be available, and the object will be initialised twice.

4. Built-in types in C++:

   - `int`: integer, can be 16-bits, usually 32-bits.
   - `float`: floating point, normally IEEE format, 32 bits.
   - `char`: single text character; really it is a small integer taking on values in [0..255]; also C++ allows arithmetic on `char`s.
   - `short`: `short integer`, possibly 8 bits, maybe 16.
   - `long` : `long integer` – 32 bits.
   - `double`: double precision floating point – more significant digits, larger exponent.
   - C++ allows you to qualify integer types a `unsigned`, meaning that they range from 0 to some large value; for example a 16-bit `unsigned int` would range 0 to 65536, whereas a 16-bit `int` (signed) would range -32768 to $+32767$.

5. `const` qualifier means compiler disallows modification.

6. `pToG` is declared outside any function; it has *global* scope; it also has *static lifetime*; we'll discuss *lifetime* in some detail later.

7. Stream I/O can handle a variety of simple values without needing additional formatting information.

8. Pronounce >> as *get-from* – standard-input stream, `cin`.

9. Assignment statement uses =; note possible confusion with test for equality — which is == in C++; but, unlike Java, something like `if(x = y)` is in fact syntactically legal, but most likely a logical error; more later about that.

10. `while` loop:

    - condition is tested;
    - if condition is true – body is executed;
    - go back to beginning;
    - if condition is false – execution continues at the statement following the loop.

11. Note the textual layout style used: as mentioned earlier, my suggestion is to indent each block by 2 characters – some use the next tab, but with that you very quickly get to the right-hand side of the page; on the other hand, we have to be able see what is part of a block and what isn't.

12. For me it is essential that the closing brace lines up with `w` of `while`.

13. Some like to move the begin `\{` to the next line – to line up with the `w` of `while`; Okay by me.

14. Mixing of operands is allowed in C++, e.g. `gint=g;`; most compilers will generate a warning. Normally, it is good practice to make type conversion explicit, this can be done with a type *cast*, thus: `gint=int(g);`

15. More about *cast*s later.

2.3 Do While

`ptog.cpp` is better written using `do ... while`; we can cut out repetition of the input and output.

```
//----- ptogdw.cpp ----------------------------
//Convert pounds to grams -- do .. while
//---------------------------------------------
#include <iostream>
using namespace std;

const double ptog = 453.592;

double convert(double p)
{
  return p*ptog;
}

int main()
{
  int pds;
  do {
     cout << "Input pounds: "<< endl;
     cin >> pds;
     double g;
     g = convert(pds);
     cout << "\nWeight = "<< g << " grams."<< endl;
     } while (pds > 0);
  return 0;
}
```

2.4 Exceptions

I doubt if you have covered *exceptions* yet in Java, but it will happen soon.

As I mentioned, if in `ptogdw` or `ptog` you type something like 2.5, the program will probably give the answer for 2 and then fail in some way; that is because it attempts to decode `" . "` as an `int` and consequently get confused.

The least we can do is detect the failure and inform the user. Later we will show how to make programs like this detect the error and request a new input.

```
//----- ptogdwe.cpp ----------------------------
//Convert pounds to grams -- do .. while, exception
//----------------------------------------
#include <iostream>
using namespace std;
const double ptog = 453.592;

double convert(double p){
  return p*ptog;
}

int main(){
  try{
    int pds = 0;
    do {
      cout << "Input pounds: "<< endl;
      cin >> pds;
      if(not cin) throw std::exception();
      double g = convert(pds);
      cout << "\nWeight = "<< g << " grams."<< endl;
     } while (pds > 0);
  } catch(...){
      cerr<< "An exception was thrown.\n";
    }
  return 0;
}
```

Brief Dissection

1. `if(not cin) throw std::exception();` As well as getting data from the input stream, decoding it, and assigning the result to `pds`, `cin` also returns a *boolean* value that is *true* when `cin` is in an error-free state; *false* if in an error state — like when it attempts to decode a decimal point in an expected `int`.
2. `if(not cin)` tests to see if what is returned is `false`.
3. `if(cin==false)` is equivalent;
4. `if(!cin)` is equivalent; `not` is relatively new to C++; before we always used `!`.
5. The two possible values of a *boolean* variable are `true, false`. `bool` is the type name.
6. The exception message is not very helpful, but it's a lot better than the program acting like a dog whose toe you trod on.
7. Exceptions are relatively new to C++, so some textbooks may not contain them.

2.5 For Loop

A variation on the pounds-to-grams program is shown in program `ptogf.cpp`.

```
//----- ptogf.cpp ----------------------------
//Convert pounds to grams - for loop - table
//--------------------------------------------
#include <iostream>
using namespace std;

const double ptog = 453.592;

double convert(double p)
{
  return p*ptog;
}
int main(){
  cout<< "Pounds \t\tgrams \t\t kg."<< endl;
  for(int pds = 0; pds < 10; pds++){
    double g=convert(pds);
    int gint = int(g);
    cout << pds << "\t\t"<< gint<< "\t\t"<< g/1000 << endl;
  }
  return 0;
}
```

Dissection

1. Note use of tab `\t` to align the table; this is one occasion where tabs are actually useful.
2. Nothing new here for Java programmers; however, I'm going to go into detail just in case people need revision.
3. For statement and loop:
4. There are three statements contained within the `(.,.,.)` in a `for` statement:
    1 `int pds=0` – initialisation.
    2 `pds<10` – loop continuation test; evaluate the condition – if true execute the body – otherwise finish.
    3 `pds++` – iteration; do this *after* the first iteration; for subsequent loops, do it *before* evaluation of the continuation condition;
5. That is, `for(initialisation; continuation test; iteration);`
6. The `for(...)` body may be a single statement or a block / compound-statement `{...}`.

2.6 Symbolic Constants and the Preprocessor

Traditional C would have replaced

```
const float ptog = 453.592;
```

with

```
#define ptog 453.592
```

The general form of a symbolic definition is:

`#define <symbolic-name> <replacement-text>`.

Note, *no* = or ;

The C++ preprocessor replaces all occurrences of `<symbolic-name>` with `<replacement-text>` – just like an editor global-replace command.

The source code is passed through the preprocessor *before* it reaches the compiler proper. You can do mighty funny things with the preprocessor, but its wide use is not encouraged in C++. In fact, probably the only valid use is in in connection with header files and the `#include` directive:

2.7 Character Input and Output

(*We'll see how much of this section we need to cover; we may run through it rather rapidly. Also, for brevity, I'm leaving exceptions out of these programs.*)

As with C, C++ input-output, including files, deals with streams of characters, or *text-streams*. This is the *UNIX* model of files — which includes keyboard and screen.

- A *text-stream* is a sequence of characters divided into *lines*.
- A *line* is zero or more characters followed by a newline character, `\n`.
- The standard input-output library *must* make each input-output stream conform to this model – no matter what is in the physical file.

Buffered and Echoed Input On most computers, input from the keyboard is *echoed*, and *buffered*.

- Echoed input. When you type a character on the keyboard, the computer input-output system *immediately* echoes it to the screen; *immediately* means *before* it is presented to the reading program, see *buffered*, below. Incidentally, this means that the input-output system, itself, does not display the typed character – what is displayed is what is echoed from the host computer).
- Buffered input. While a program is reading from a keyboard – *or a file*, the computer stores all the input characters in a buffer (array) and presents the array (line) of characters to the reading program *only* after *Enter* has been typed.

- Buffered output. While output is being produced, the computer stores all the characters in a buffer (array) and presents the array (line) of characters to the output device *only* after $new - line$ or `endl` has been reached.

Terminal input-output and I-O Redirection C++, C, and UNIX have unified view of text input-output. Reading from the keyboard is like reading from a file device – `stdin` in C. So, when we talk about *file* I/O below, we include terminal I/O. NB. `ctrl-d` is *end-of-file* for a Linux / UNIX keyboard; I think it might be `ctrl-z` on Windows `cmd`.

If you want to test the programs using files, you can use *input-output redirection*.

Then, `$$ prog < test.dat` (assuming the prompt is $$).

reads from `test.dat`; i.e. the < redirects the program to read from the file instead of the keyboard.

If you want to send the output to a file, say `testout.dat`,

```
prog < test.dat > testout.dat
```

2.7.1 File copying

Program `cio1.cpp` shows how to copy from an input *file* (or keyboard) to an output file (or screen).

```
//----- cio1.cpp -----------------------------------
// copy input to output, version 1.
//  j.g.c. 1/2/97, 2006-09-06
//--------------------------------------------------
#include <iostream>
using namespace std;
int main(){
  char c;

  while(cin.get(c)){
     cout<< c;
  }
  return 0;
}
```

You can execute it simply by `cio1` and typing characters. Terminate by *ctrl-d – end-of-file*, but make sure the program is executing before you do this – otherwise *ctrl-d* logs you out!

Alternatively, create a file `test.dat`, e.g. simply use

```
copy cio1.cpp test.dat
```

and then use input-output redirection, see 2.7:

```
cio1 < test.dat > testout.dat
```

Dissection

1. `while(cin.get(c))` both reads the next character into `c` *and* tests whether `cin.get()` returns `true` – OK, or `false`, indicating some problem, e.g. *end-of-file*.
2. When `false` is returned, we quit the loop.
3. C programmers please note, it's `char c;`, *not* `int c;`, since in C++, there is no need to code *end-of-file* in the character itself.

What is the problem with `cin >>c` as in `cio2.cpp`?

If you run `cio2.cpp` you will find that `cin >> c` loses *white-space*s, i.e. *newline*, *tab*, and *space* itself.

```
//----- cio2.cpp -----------------------------------
// copy input to output, version 2, wrong!
//  j.g.c. 1/2/97, 2006-09-06.
//--------------------------------------------------
#include <iostream>
using namespace std;
int main(){
  char c;

  while(cin >> c){
    cout<< c;
  }
  return 0;
}
```

2.7.2 Character counting

Program `ctch1.cpp` shows how to count the characters in the input stream, stopping at *end-of-file* and presenting the count.

```
//----- ctch1.c -------------------------------------
//counts input chars; version 1
// j.g.c. 1/2/97, 2006-09-06
//based on K&R p.18.
//---------------------------------------------------
#include <iostream>
using namespace std;
int main()
{
  char c;
  long nc=0; //long to make sure it is 32-bit int.

  while(cin >> c){
    ++nc;
  }
  cout<< nc;

  return 0;
}
```

Dissection

1. ++ operator – increment by one; -- is decrement by one. On their own, *post-increment* `++nc` and *pre-increment* `nc++` are equivalent, but give different values when used in expressions. E.g.

   ```
   int i,n=6;
   i=++n; /*POST-increment gives i==7, and n==7*/
   ```
   but
   ```
   int i,n=6;
   i=n++; /*POST-increment gives i==6, and n==7*/
   ```

2. `long` (int) is at least 32 bits long.

2.7.3 Line Counting

Program `clines.cpp` shows how to count the lines in the input stream, stopping at *end-of-file* and presenting the count.

```
//----- clines.cpp -----------------------------------
// count lines in input.
//  j.g.c. 1/2/97, 2006-09-06.
//----------------------------------------------------
#include <iostream>
using namespace std;
int main()
{
  char c;
  int nl = 0;

  while(cin.get(c)){
    if(c=='\n')++nl;
  }
  cout<< nl <<endl;

  return 0;
}
```

Dissection

1. `if` statement:
   - Tests the condition in `(..)`.
   - If true executes statement or group `{..}` following.
   - Otherwise (false) skips them.
   - Or, can add `else`, e.g. `else cout<< "char not a newline";`
2. == denotes comparison operator *is-equal-to*. *Caution*: = in place of == can be syntactically correct and so need not trapped by compiler – a nasty hard-to-find error results!
3. Character constants, e.g. `'\n'` represents a `char` value equal to the value of the character in the machine's character set; In ASCII, e.g.: `\n` == 10 decimal, `'A'` == 65 decimal. You should *never* use numeric values, they might change, and render your program non-portable.

2.7.4 Word Counting

For the purposes of this program, a *word* is any sequence of characters that does not contain a *white-space* character – i.e. *blank-space*, *tab*, or *new-line*.

Program `clwc.cpp` shows how to count lines, words, chars in the input stream, stopping at *end-of-file* and presenting the counts.

```
//----- clwc.cpp -----------------------------------
// count lines, words, and chars in input.
// as in K&R -- a word is simply some chars delimited by
// whitespaces
//  j.g.c. 1/2/97
//--------------------------------------------------
#include <iostream>
using namespace std;
int main()
{
  char c;
  int nl,nw,nc;
  bool out = true; //outside a word

  nl = nw = nc = 0;
  while(cin.get(c)){
    ++nc;
    if(c=='\n')
      ++nl;
    if(c==' '||c=='\n'||c=='\t')
      out=true;
    else if(out){
      out=false;
      ++nw;
    }
  }
  cout<< nl<<" "<< nw <<" "<< nc <<endl;

  return 0;
}
```

Dissection

1. `nl = nw = nc = 0;` In C++, an assignment has a value, i.e. `nc=0;` has the value `0`, which is, in turn, assigned to `nw` etc..
2. `||` denotes Boolean *or*
3. `&&` denotes Boolean *and*
4. Note form of `if...then...else` – you do not write `then`.

```
 if(expression)
   <statement1>
else
   <statement2>
```

As usual `<statement>` can be a compound statement `{..}`.

2.8 Arrays

Program `cdig.cpp` shows a program to count the occurrences of each numeric digit, of white-spaces and of all other characters together – without using 12 named counters!

```
//----- cdig.cpp -----------------------------------
// counts digits, white-spaces, others
// after K&R chapter 1
// j.g.c. 2/2/97, 2006-09-06.
//--------------------------------------------------
#include <iostream>
using namespace std;
int main(){
  char c;
  int i, nWhite = 0, nOther = 0;
  int nDigit[10];

  for(i = 0; i< 10; i++){
    nDigit[i] = 0; //variable need to be explicitly initialised
  }
  while(cin.get(c)){
    if(c>='0' && c<='9')  //is it a digit?
      ++nDigit[c-'0'];
    else if(c==' '||c=='\n'||c=='\t') // a white space?
      ++nWhite;
    else
      ++nOther; //otherwise!
  }
  cout<< "digit counts = ";
  for(i = 0;i < 10; ++i) cout<< nDigit[i]<< ", ";
  cout<<"\nWhite spaces "<< nWhite<<"\nothers "<<nOther<< endl;

  return 0;
}
```

Dissection

1. `int ndigit[10];` defines an array of 10 `int`s.
2. Unlike in Java, C++ array is **not** an object;
3. Notice no `int[] nDigit = new int[10];`
4. `for(...)` loops must go 0,1 ... 9, since, in C++, array subscripts must start at 0; same as Java.
5. The C++ code pattern to loop over the first `n` elements of an array is: `for(int i=0; i<n; i+)+`.

6. `&&` denotes logical *and*; same as Java.

7. `c-'0'` assumes that `0, 1, ... 9` have successive values. Actually, there is a function `isdigit(char c)` which is a better way. If you use `isdigit()`, you must have `#include <ctype.h>`.

8. `c-'0'` is an integer expression – so it's OK for a subscript.

9. The following is the model for a multi-way decision; you can have any number `else if`. Note the default.

10. Note: there is *no* `elsif`, it is just `else` followed by another statement, which may be `if`.

```
if(condition1)
   statement1;
else if(cond2)
   stmt2;
 .........
else
   stmtn; //default -- if none of above true
```

11. There is another `switch-case` multi-way construct that we will encounter later. But, anything you can do with it, you can do with an `if...else` *ladder*.

12. Note indenting style; all the `else`s are of equal status, hence they should be aligned; and, we don't want to run off the right-hand edge of the paper.

2.9 Functions in more detail

Program `Tr5a.cpp` shows a program based on `Tr5.cpp` from Appendix A, with a few modifications. We'll use it as a case-study to exemplify many aspects of the use of functions.

```
/*----- Tr5a.cpp --------------------------------
 j.g.c. 2003/11/29, 2006-09-06.
 based on Tr5.cpp, see that program for proper comments
 and analysis
-------------------------------------------------*/
#include <iostream>
using namespace std;

void nl(){
  cout<< '\n';
}

int stars(int n){
  int nc = 0;
  for(int i= 0; i< n; i++){
    cout<< '*'; nc++;
  }
  n = n + 10; //just to demonstrate pass by value
  return nc;
}

int spaces(int n){
  for(int i= 0; i< n; i++){
    cout<< ' ';
  }
  return n;
}

int main(){
  int h = 5;
  int nSp = 0;
  int nSt = 0;
  int nNl = 0;
  for(int j= 0; j< h; j++){
    nSp += spaces(h - 1 - j);
    nSt += stars(j + 1);
    nl(); ++nNl;
  }
  int nc =  nSp + nSt + nNl;
  cout<< "Number of characters = "<< nc<< endl;
  return 0;
}
```

Dissection

1. A function definition consists of:

```
return-type function-name(parameter list -- if any)
{
   definitions of local variables;
   statements;
}
```

2. If the function has no parameters, you may use either `(void)` or `()`

3. The functions may be all in the same file, along with `main`, or may be spread across many files.

4. The variable `nc` is local to `stars`; it is invisible elsewhere. It is quite distinct from the `int nc` in `main`.

5. Unless the function returns a value (some don't) return is not essential, but if the function declaration indicates that the function returns a value, then there must be an appropriate return statement.

6. The calling function may ignore the returned value.

7. The default for arguments in C++ is *pass-by-value*. Notice that

```
n = n + 10; // 'pass-by-value'+ has no effect on the variables in main
```

   Of course, this is the same as C++. When we get to objects, we will see that C++ passes by value too (the full object); that will be discussed in detail later.

2.9.1 Declaration of functions

Program `Tr5b.cpp` shows a version with altered layout; here we've kept `main` at the beginning, and functions at the end; consequently, we've had to *declare* functions `stars` etc. *before* they are called.

C++ demands that functions be declared before they are called.

```
/*----- Tr5b.cpp --------------------------------
 j.g.c. 2003/11/29, 2006-09-06.
 based on Tr5a.cpp
-----------------------------------------------*/
#include <iostream>
using namespace std;

void nl();
int stars(int n);
int spaces(int n);

int main(){
  int h = 5;
  int nSp = 0;
  int nSt = 0;
  int nNl = 0;
  for(int j= 0; j< h; j++){
    nSp += spaces(h - 1 - j);
    nSt += stars(j + 1);
    nl(); ++nNl;
  }
  int nc =  nSp + nSt + nNl;
  cout<< "Number of characters = "<< nc<< endl;
  return 0;
}

void nl(){
  cout<< '\n';
}

int stars(int n){
  int nc = 0;
  for(int i= 0; i< n; i++){
    cout<< '*'; nc++;
  }
  n = n + 10; //just to demonstrate pass by value
  return nc;
}

int spaces(int n){
  for(int i= 0; i< n; i++){
```

```
    cout<< ' ';
  }
  return n;
}
```

Dissection

1. `int stars(int n)` declares the *type* of `stars`; this is called the *prototype* for `stars`;
2. `int stars(int n)` is a *declaration*; declares the type of `stars`;
3. On the other hand,

   ```
   int stars(int n){
     int nc = 0;
     for(int i= 0; i< n; i++){
       cout<< '*'; nc++;
     }
     return nc;
   }
   ```

   is a *definition* of `stars`; defines what it does; like assigning a value to a variable;
4. `int i` declares the type of `i`; `i = 10` gives it a value (defines it).
5. `stars` has type: `int -> int`; it takes an int and returns an int;
6. `nl` has type: `void -> void`; takes an empty argument list and returns nothing;
7. Parameter names in prototypes are neither significant nor necessary. But, they can provide good documentation.
8. e.g. `int stars(int n)` and `int stars(int)` are equivalent.
9. Normally, it is a good idea to split a program into *modules* (different files), e.g. the stars and spaces functions we've written may be tested and stable, whilst the main program is subject to change; it's nonsensical to have to re-compile the functions each time, and while they are in a file that's subject to change, we have no guarantee that errors or other changes could be introduced to them; so we'll put them in a separate file `funs.cpp`.

2.9.2 Program Split into Modules

Program `Tr5c.cpp` shows the new main program; here, we `\#include` the function declarations in `funs.h`, but N.B. *not* their definitions / implementations).

Now, the program and functions are compiled and loaded using:

Notice that `.cpp` files are compiled, and not `#included`. `.h` files, on the contrary, are `#included`, but not separately compiled; (a compiler may choose to (pre-)compile `.h` header files, but that's an optimisation that we'll ignore here.

```
/*----- Tr5c.cpp --------------------------------
 j.g.c. 2003/11/29, 2006-09-06.
 based on Tr5b.cpp, split into modules / files.
-------------------------------------------------*/
#include "funs.h"
using namespace std;

int main(){
  int h = 5;
  int nSp = 0;
  int nSt = 0;
  int nNl = 0;
  for(int j= 0; j< h; j++){
    nSp += spaces(h - 1 - j);
    nSt += stars(j + 1);
    nl(); ++nNl;
  }
  int nc =  nSp + nSt + nNl;
  cout<< "Number of characters = "<< nc<< endl;
  return 0;
}
```

Files `funs.h` and `funs.cpp` show, respectively, the function declarations and the definitions (implementations).

```
/*------ funs.h ---------------------
  used by Tr5c.cpp
  j.g.c. 2006-09-06
-----------------------------------*/
#ifndef FUNS_H
#define FUNS_H
#include <iostream> //good idea to have this here, funs.cpp needs it

void nl();
int stars(int n);
int spaces(int n);

#endif
```

```
/*------ funs.cpp ---------------------
  used by Tr5c.cpp
  j.g.c. 2006-09-06
-----------------------------------*/
#include "funs.h"
using namespace std;

void nl(){
  cout<< '\n';
}

int stars(int n){
  int nc = 0;
  for(int i= 0; i< n; i++){
    cout<< '*'; nc++;
  }
  n = n + 10; //just to demonstrate pass by value
  return nc;
}

int spaces(int n){
  for(int i= 0; i< n; i++){
    cout<< ' ';
  }
  return n;
}
```

Dissection

1. When using a library or external *module*, you must get into the habit of producing a *header* `.h` file that contains declarations of the functions.
2. Not unreasonably, C++ demands that you declare functions before you *call* them.
3. The prototypes of standard library functions and standard classes are contained in header files, e.g. `iostream` is just an ordinary text file containing the *stream* classes and functions.
4. `.h` files get compiled as part of any file in which they are included. Recall, the C++ Preprocessor executes all # commands before the compiler proper sees the source code.

When you have separate *modules* or *compilation units*, obviously, the compilation & linking procedure must be modified.

More of this in section 2.10.4.

I'm going to use command line here; the same will happen in Visual Studio, but the details will be hidden.

2.10 Creation of Executables

2.10.1 Some Basics – Compiling and Linking

Source programs like `Tr5b.cpp`, `Tr5c.cpp` etc. are *not* directly executable. There are a number of stages in creation of the executable, and executing it.

Compilation The first stage of creating an executable is to *compile* `Tr5c.cpp` into *object* code.

```
g++ -c Tr5c.cpp
```

```
g++ -c funs.cpp
```

This compiles and puts the object code into a files `Tr5b.o` and `funs.o`. This object code is essentially *machine* code. It has most the *building blocks*, except the *system* code, which is in *libraries*.

Linking The second stage is to *link* the machine code in `Tr5c.o` with appropriate library code: the — put the building blocks together.

This produces the executable file `Tr5c`, or `Tr5c.exe` in Windows.

Execution Finally, to execute `Tr5c.exe`, you type

`Tr5c`.

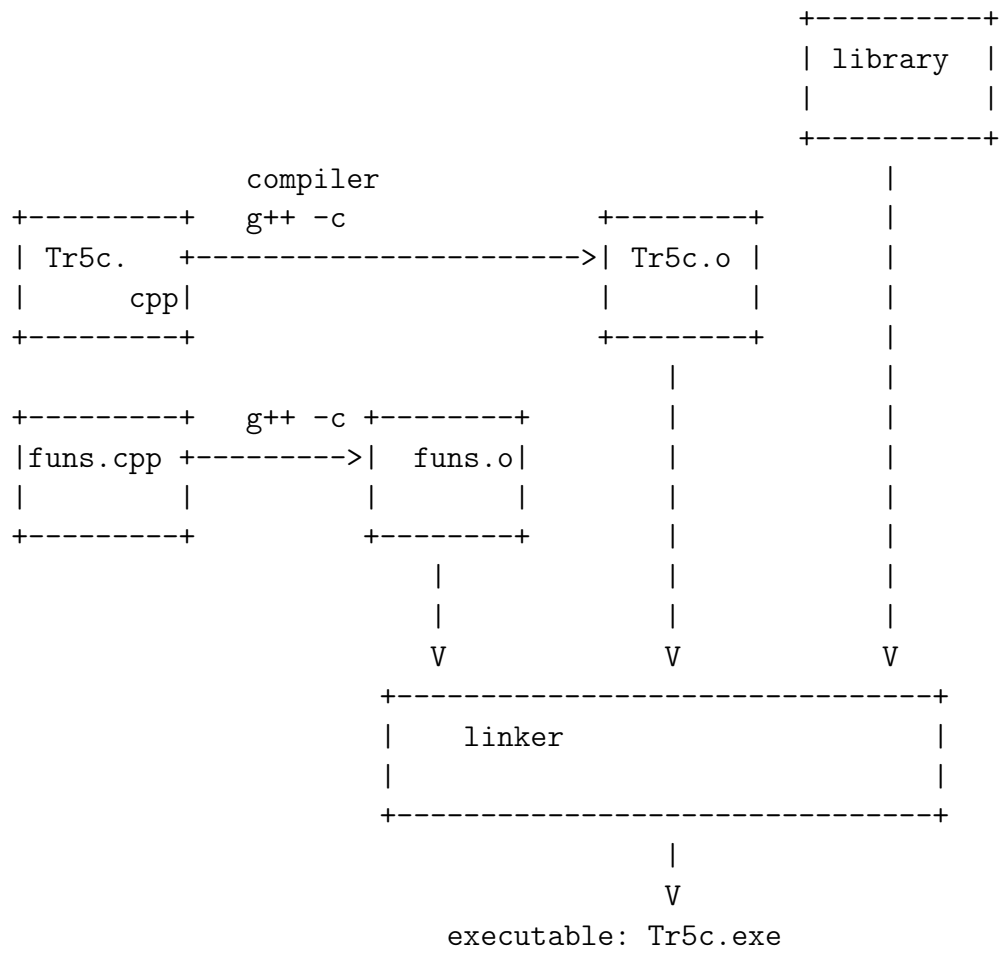
This reads the contents of `exe` into memory, and starts execution at an appropriate start address.

Compilation & Linking – Summary The situation may be made more clear-cut if we look at the two module program `Tr5c.cpp`, `funs.cpp`. Recall,

```
g++ -c Tr5c.cpp       # yields Tr5c.o
g++ -c funs.cpp          # -> funs.o
```

Now link `g++ -o Tr5c Tr5c.o funs.o  # yields executable Tr5c`

The following figure describes the process.



2.10.2 Other Libraries

Not all *system* functions are in the default libraries that are searched by `g++` and `ld`. For example, maths functions like `sqrt`. If you were to call `sqrt` in `Tr5c`, you would have to explicitly mention the mathematics library, using `-lm`:

```
g++ -o Tr5c.exe Tr5c.cpp funs.cpp -lm
```

You must also remember to `#include` the appropriate *header file*, e.g. `#include <math.h>`.

Likewise for libraries to do with OpenGL functions.

2.10.3 Static versus Shared Libraries

In section 2.9.2, the code for `stars`, `spaces`, and `nl` is linked *statically*, i.e. the object code for each of the functions is copied into the file `Tr5c.exe`. For common functions like `cin.get` this can become wasteful; hence *shared* libraries, in which the linker inserts just a *pointer* to shared code that is already in the operating system. Windows `DLL`s are a bit like this.

2.10.4 Make and Makefile

When you get to more complicated systems of multi-module programs, the utility `make` can become useful. In fact, IDEs like Visual Studio often use `make` to do a *build*.

`make` is like `Ant` that we used with Java projects.

`make` executes the file `Makefile`; i.e. the same as `Ant` executing `build.xml`; but maybe you never noticed that.

The following shows a `Makefile` for `Tr5c.cpp`.

```
CC = g++ -W -Wall -ansi -pedantic -Wformat-nonliteral \
-Wcast-align -Wpointer-arith -Winline -Wundef \
-Wcast-qual -Wshadow -Wconversion -Wwrite-strings \
-ffloat-store -O2

all: Tr5c

funs.o: funs.cpp funs.h

Tr5c.o: Tr5c.cpp funs.h

Tr5c: Tr5c.o funs.o

test: all
./Tr5c

clean:
rm -f *.o *~ *.t *.gch *.exe Tr5c
```

The standard practice is to name this file `Makefile`, then invoking

```
make
```

causes `exe` to be *built*: all programs compiled and linked. Make takes care of dependencies, e.g. if `funs.o` and `funs.o` already exist, make will not recompile them. On the other hand, if they exist, but if, say, `Tr5c.cpp` has changed since the last compilation to `Tr5c.o`, then make will recompile `Tr5c.cpp`, but *not* `funs.c`.

In the example above,

```
make clean
```

will delete the `.o` files, and other temporaries and executables.

Make is very much part of UNIX and Linux; it's handy to know that it exists, for if you ever download a program for Linux as source (need to be compiled and built), you will encounter a `Makefile`.

Another point, because of its complex syntax, people rarely create a `Makefile` from scratch; find a template `Makefile` (one that works!) that is close to what you require and change bits to suit. Be careful with the places where the TAB syntax matters.

2.10.5 Interpreted Languages

The chief difference between a **compiler** and an **interpreter** is as follows. A compiler translates from one language **source** to some other **object** language. At its simplest, an interpreter takes a program and executes it.

When you create an interpreted language program (lets call the language `Basic`, you get a choice: you can compile and create an `.exe` file much as described above.

If you run the Basic program in interpreted mode, then something quite different happens. There is an *interpreter* program (let's call it `basinterp`) which reads the *source* and executes it directly. `basinterp` has its own **fetch-decode-execute** cycle running.

We have something like the following, grossly simplified, assuming that we are dealing only with instructions of the form `result = num1 operation num2` .

```
int operand1, operand2, result;
String line;

f= open(prog1.bas);

while(NOT end-of-file(f))

 line = readLine(f); /*fetch next line*/

 decode 'line' to produce:
     - operation
     - operand1, operand 2

 /* execute */

 if(operation=='+')result= operand1 + operand2;
 else if(operation=='-')result= operand1 - operand2;
 else if(operation=='/')result= operand1 / operand2;
 else if(operation=='*')result= operand1 * operand2;
 else if(operation=='F')result= fred(operand1, operand2);
 else /* etc */

 assign 'res' to actual result variable.

 and go back for more ...
}
```

2.10.6 Java — Compiler AND Interpreter

Java is interpreted, but its interpretation is a little different from that described for Basic. Java programs go through a compiler **and** an interpreter. When you compile a Java program (e.g. `prog1.java`), you produce a file `prog1.class` which contains Java **byte-code**. Java **byte-code**

is sort-of like machine code, which is interpreted by a Java interpreter program called Java Virtual Machine(JVM).

So, we have a Java compiler, which produces the byte-code; then we have the Java interpreter program, the JVM, which applies its version of the *fetch-decode-execute cycle*; Of course, the Java Virtual Machine is software.

Since the JVM is software, any computer can execute Java byte-code as long as they have the JVM interpreter program.

From the point of view of Operating Systems, you could think of the JVM as another *layer* on top of the operating system, and below the applications. Indeed, the JVM provides *operating-system-like* facilities, like support for concurrency (multitasking).

Another *operating-system-like* facility provided by the JVM is its treatment of *applets*. Applets are Java byte-code (executable) programs meant primarily for downloading over the Internet by web-browsers and executed directly by the browser running the JVM. Now, this would normally be a recipe for disaster, and you would fear all sorts of viruses and malicious programs. However, rather like an operating system with its *privileged* and *user* modes, the JVM has a special mode for executing applets. For example, applets may not access disk files.

2.10.7 Java — Dynamic Linking

You will notice that a Java `.class` file may be a lot smaller than the equivalent C++ executable. This is because Java does not build a great big executable that contains everything — including local and system library code; Java keeps the class files separate and loads them into memory only as needed — *dynamic linking*.

2.10.8 Java Enterprise Edition (J2EE) and .NET

A JVMs runs on a single machine. In these days of the Internet, people and enterprises often want to do something called *distributed computing*, i.e. a set of cooperating processes, (see Operating Systems course), executing on separate machines connected via a network (such as the Internet). This is all the easier if the processes are all running on JVMs. Add a bit of Internet glue and you have J2EE.

Where are Microsoft in all this? When Java (developed and *owned* by Sun Microsystems) became successful, Microsoft did not like it. They attempted to create an extended Java of their own; but Sun forbade them. Then came the success of Java distributed computing. So Microsoft designed their own version of J2EE — called .NET. C# (pronounced "C-sharp") is Microsoft's attempt at Java. Like Java, C# is compiled to a sort of byte-code, and executed on a virtual machine. The is a .NET version of VisualBasic.

2.10.9 Ultimately, All Programs are Interpreted

If you think about it, in computing, interpreters (in general) crop up everywhere. In addition to the examples we have mentioned:

- The operating system shell (e.g. Windows `cmd`) fetches and decodes your commands, and then gets the kernel to do the executing;
- The hardware processor itself is an interpreter; in fact, if you look at the *microprogram* that runs Mac-1, you will clearly see that the program performs a *fetch-decode-execute* cycle. And even if the processor is controlled by hardware rather than a microprogram, then the hardware will, in its own way, be performing interpretation.

2.11 Summary

This chapter has given a quick tour of C++ syntax, certain basic facilities, and compilation and linking and `make`.

Chapter 3

Variables, Types, etc. in C++

3.1 Introduction

This chapter discusses the *elementary* types provided by C++ (`int`, `float` etc.), together with operators and expressions involving them. We call these *built-in* types to distinguish them from types formed by *classes*.

We mention also the *standard library* types (C++ classes) `string` (very like Java's `String`) and `vector` (very like Java's `ArrayList`).

We leave record (aggregate) types, i.e. `structs`), until a later chapter.

3.2 Variable names

A variable name is made up of letters and digits; it must start with a letter; underscore _ is also allowed. Note C++ keywords, you cannot use them for names, though they can be part of names.

The world of programming is sharply divided by people's preferences in variable naming, especially, multi-word names; some choices:

(i) Underscore separators, e.g. `multi_word_name`;

(ii) Uppercase for inner names, e.g. `multiWordName`;

(iii) Hungarian notation – favoured by Microsoft, which is a special case of (ii), with the first few letters used to code the *type* and any special *use* of the variable.

I now prefer (ii), with partial use of (iii) for pointers – each pointer name begins in `p`.

Note that style and consistency affect readability – and that is as important as syntactic correctness.

C++ is case sensitive. Although case has no syntactic significance, typical uses of case are:

- As above, lower-case for first word, upper-case for leading letter of remaining words – if any;

- Upper-case for *leading* letter of *class* names.
- Upper-case for all letters of `#define` symbolic-constant, e.g. `#define BIGNUM 1000`.

For class data members, I follow the FAQ (Cline, Lomow & Girou 1999b) and use a trailing underscore, e.g. `int sec_;`.

3.3 Elementary Data Types and Sizes

The C++ built-in type system is less rich than some other programming languages. If we ignore *pointers*, there are just two basic categories:

- Integer types, which represent whole numbers, both positive and negative.
- Floating point types, which represent numbers that have fractional parts.

Within these two categories, the different types differ only by the range of numbers represented — which depends on the *size* of the representation (number of bytes / bits used) and whether the representation is *signed* or *unsigned*. In allowing the *signed* qualifier, C++ departs from what you are accustomed to in Java.

3.3.1 Integer types

Integer types can be either `signed` (default) or `unsigned`. A `signed` type has the range [-min..max], e.g. `signed char` [-128..+127]; an `unsigned` has the range [0..max], `signed char` [0..+255].

`signed` is the *default*, so that `char c` has the range [-128..127].

- `char`. Typically represented by a byte (8-bits). Suitable for holding single text characters or small integers. Hence, there is nothing to stop you doing arithmetic on them.
- `int`. Typically represented by either 16-bits (range [-32,768..+32,767]) or 32-bits. Again, `signed` is the *default*, so that `signed int i;` is equivalent to `int i;`
- `long int`, or simply `long`. Typically represented by 32-bits. As usual, `signed` is the *default*.
- `bool`. Also an integer type. Although `bool` has values false, true, C++ does little to disguise that these are represented by values 0, 1, respectively. If you assign a `bool` to an `int`, you get either 0 or 1. If you assign an `int` to a `bool`, any value other than 0 will convert to `true`, while 0 will convert to `false`. This is demonstrated by program `bool.cpp`.

```
//----- bool.cpp ----------------------------------
// tests bool type.
// j.g.c. 9/2/97, 7/2/99, 2006-09-13
//-------------------------------------------------
#include <iostream>
using namespace std;

int main()
{
  bool b; int i;

  b = true;
  cout<< "bool b = true: "<< b<< endl;

  b = false;
  cout<< "bool b = false: "<< b<< endl;

  b += 22;
  cout<< "b+=22: "<< b<< endl;

  i = false + 22;
  cout<< "i=false+22: "<< i<< endl;

  i = true + 22;
  cout<< "i=true+22: "<< i<< endl;

  return 0;
}
```

Output.

```
bool b = true: 1
bool b = false: 0
b+=22: 1
i=false+22: 22
```

In general, in C++ you must be careful about overflowing integer types. In the example shown in program `char.cpp`, the variable `unsigned char u` increments satisfactorily between 0 and 255. But `char c` and `signed char s` go badly wrong at 127; if you are careless of problems like this, they can lead to very confusing errors.

```
//----- char.cpp ----------------------------------
// tests overflow of char, unsigned char and signed char.
// also shows some limits
// j.g.c. 2006-09-13
//-------------------------------------------------
#include <iostream>
```

```
#include <climits>
#include <cfloat>

using namespace std;
int main()  {
  char c,x; unsigned char u; signed char s;
  int i;

  cout<< "CHAR_MAX = "<< CHAR_MAX<< endl;
  cout<< "CHAR_MIN = "<< CHAR_MIN<< endl;

  cout<< "SCHAR_MAX = "<< SCHAR_MAX<< endl;
  cout<< "SCHAR_MIN = "<< SCHAR_MIN<< endl;

  cout<< "UCHAR_MAX = "<< UCHAR_MAX<< endl;

  cout<< "SHRT_MAX = "<< SHRT_MAX<< endl;
  cout<< "SHRT_MIN = "<< SHRT_MIN<< endl;

  cout<< "INT_MAX = "<< INT_MAX<< endl;
  cout<< "INT_MIN = "<< INT_MIN<< endl;

  cout<< "INT_MAX = "<< UINT_MAX<< endl;

  cout<< "FLT_MAX = "<< FLT_MAX<< endl;
  cout<< "FLT_MIN = "<< FLT_MIN<< endl;

  cout<< "DBL_MAX = "<< DBL_MAX<< endl;
  cout<< "DBL_MIN = "<< DBL_MIN<< endl;

  for(i=s=c=u=0; i<16;i++){
    for(int j=0;j<16;j++,u++,c++,s++){
      cout << "u = "<< int(u);
      cout << ", c = "<< int(c);
      cout << ", s = "<< int(s)<< endl;
    }
  cout<< "type any char + enter:"; // just to pause the loop
  cin>> x;
  }
  return 0;
}
```

At the beginning, everything goes fine:

```
u = 0, c = 0, s = 0
u = 1, c = 1, s = 1
u = 2, c = 2, s = 2
...
```

```
u = 14, c = 14, s = 14
u = 15, c = 15, s = 15
type any char + enter:
```

But when we get to 127 odd things happen:

```
u = 125, c = 125, s = 125
u = 126, c = 126, s = 126
u = 127, c = 127, s = 127
type any char + enter:c
u = 128, c = -128, s = -128
u = 129, c = -127, s = -127
u = 130, c = -126, s = -126
```

3.3.2 Floating-point types

- `float`. Typically 32-bit IEEE floating point.
- `double`. Typically 64-bit IEEE floating point.

These are always *signed*.

3.3.3 Implementation Dependencies

`<climits>`, `<cfloat>` contain symbolic constants that specify the limits and sizes of the integer and floating point types on the current implementation, see `char.cpp` above. If you really want to *bullet-proof* your software against differing representations — to make it portable — you may need to know about these.

3.4 References

A *reference*, serves as an alternative name for the object with which it has been initialised. The definition of a reference *must* specify an initialisation.

Example:

```
int x = 10; //'plain' int
int& r = x; //reference
```

`x` is defined as an *ordinary* `int`; subsequently, `r` is defined as a reference, and *initialised* with `x`.

Think back to Java.

```
String s = new String("Hello"); // s is a reference to an object
                                 // containing "Hello";
String t = s;  // t is a reference to the same object
```

Defining a reference without initialisation results in a compiler error:

```
int& r; //compiler error: reference not initialised
```

All operations applied to the reference act on the variable to which it refers, i.e. the same as with Java objects:

```
r = r+ 2;
```

adds 2 to `x`, now `x == 12`.

Thus, a reference is a true *alias*; since most computer science warns of the problems of aliases, why references? The only valid use for references that I have come across is in reference parameter passing in functions.

Recall that C++ defaults to *pass-by-value* / *pass-by-copy*, so that in the example below, the parameters `a, b` are *copies* of the actual arguments: any alteration of `a, b` will not be reflected in the calling program. However, `c` is different, it is a *reference* which is *initialised with the actual parameter*. Hence, `c` is an *alias* for the actual parameter, and any changes to `c`, *will* also be changes to the actual argument – in the calling program.

```
void addf(float& c, float a, float b)
{
  c = a + b;
}

float x= 1.3, y= 4.5, z= 10.2;
addf(z, x, y);
// now z== 5.8
```

Reference parameters have an additional advantage that they remove any performance penalty caused by passing (copying) large objects. In such a case, some safety can be ensured: if a reference parameter is not modified within the function, it can be declared with the `const` modifier.

3.5 Arrays

For *any* type `T`, the definition

```
T a[n];
```

defines an array containing `n` elements of type `T`.

e.g. `float x[10];//10 floats.`

Array elements are indexed as, e.g `x[5] = 5.0` places `5.0` in the *fifth* element of `x[]`; but *fifth* only if you start counting at *zeroth*: the indices of `float x[10]` go 0, 1, ..., 8, 9. The C++ idiom for traversing such an array is:

```
for(int i=0; i<10; i++){
  x[i] = float(i); //for example
}
```

Multi-dimensional arrays `int j[4][15];`

declares an array with 4 elements, and each of these is itself an array of 15 `int`s.

The *tenth* element of the *second* (again counting from *zero*!) is indexed as:

`j[2][10]`

N.B. *NOT* `j[2,10]`.

Unfortunately, the latter is legal, but a different thing altogether.

Arrays and functions Arrays passed to functions are *always* passed by *reference*, see above.

Pointers and arrays As noted in section 3.6.1, in C++ pointers and arrays are *very* closely related. In fact, the previous fragment can legally be replaced by

```
for(int i=0; i<10; i++){
  *(x+i) = float(i);
}
```

But try to avoid this style – it makes your program hard to read, and it does *not* make it run faster.

Arrays as function parameters When an array, e.g. `int a[10]` is passed to a function, e.g. `fred(a)`, it decays to a pointer; this is one way of viewing the fact that arrays are not passed by *value*.

3.6 Pointers

3.6.1 Introduction

In any high-level language, pointers are something of a necessary evil. In some languages, their role is quite restricted: they are used only as a method of accessing memory allocated dynamically off the heap, the pointer is used as a *reference* for the allocated memory — which is otherwise anonymous. Java has managed to avoid them altogether — because, in Java, all identifiers, except those which identify elementary variables (`int`, `float`, etc.) are references (rather than values).

In C++, a pointer can point to *any* sort of memory, they are the C++ embodiment of machine language *addresses*. It is common to use the term *address* as a synonym for *pointer*; nevertheless, taken literally, this is usually unhelpful.

For many reasons, including the syntax for declaring them, pointers in C++ and C are known to be difficult and error prone.

There are two primary uses for *pointers* in C++:

- As references for *anonymous* memory allocated dynamically off the heap – using operator `new`; in C, the equivalent is `malloc`.
- In C++, as in C, pointers are closely related to arrays. For example, as already mentioned in the previous section, when an array is passed to a function the array reference decays to a pointer. There are other cases where arrays and pointers are indistinguishable, but to dwell on this matter would introduce unnecessary detail and possibly recommend bad habits.

In C, pointers had to be used to provide a sort of *programmed pass-by-reference* for functions. Fortunately, C++ has *reference*s, see above, which are much more suitable for this purpose.

3.6.2 Declaration / Definition of Pointers

Pointers are defined as in the example below:

```
int* pi;   // pi is of type: pointer-to-int
float* pf; // pf is of type: pointer-to-float
char* s;   // s  is of type: pointer-to-char
```

It is most important to realise that there are two parts to a *pointer* type declaration:

- The fact that it is a pointer.
- Its base type, e.g. `pi` pointer-to-`int`.

3.6.3 Referencing and Dereferencing Operators

Reference Operator & When applied to a variable, & generates a *pointer-to* the variable. Some books say *'address-of'* – Okay so long as you don't take it too literally, e.g. attempt to use the *'address'* as a numeric address; this may be more natural, owing to the *a*ampersand sign.

Example:

```
int* p;
int c;

p = &c; // causes p to point-to c.
```

Dereference operator * `*` *de*references a pointer, i.e. it obtains the value of variable / object that a pointer points-to – it simply reverses the action of &; i.e. `d = *(&c);` is equivalent to `d = c;`.

Example:

```
int* p; int c, d;

p = &c;
d = *p; //d now has the same value as c.
```

Beware of Confusion The overloading of `*`, with two quite separate uses:

- Declare `pointer-to` – in a declaration, e.g. `int* p;`. `p` is a *pointer*.
- *Dereference* – in an expression, e.g. `int i = *p;`. `p` is a pointer, but `*p` is an `int`!

is most confusing to those new to C++ and C.

Uninitialised and dangling pointers Defining a pointer creates the pointer itself, but it does not create the object *pointed-to*. Moreover, defining a pointer does not initialise it, hence it can point anywhere. Thus,

```
int *p;
```

creates a pointer, which is initialised to some random value, i.e. it could point at a location in free memory, part of your program, or anywhere. Now,

```
*p=10;
```

causes the value `10` to be written to the location `p` points to. Unless this is free memory, this may cause your world as you know it may come to an abrupt end. Fortunately, the memory management of Linux will usually trap a severe violation, resulting in program halt and the message *segmentation error*.

A *dangling pointer* is similar in effect. Consider the function `fred`:

```
int* fred(int a)  /*fred returns a 'pointer to int' */
{
  int* p; int b;

  b=a+33;
  p=&b;  /*p 'points to' b */
  return p;
}
```

caller:

```
int*pa; int c,d;
...
pa=fred(c);
```

This looks fine, but, whenever control returns from `fred` to its calling point, `b` gets destroyed; consequently, `pa` points to something that doesn't exist.

3.6.4 Pointers and Arrays

As mentioned in section 3.6.1, pointers are sometimes used interchangeably with arrays; not the best thing, but it happens and you must be able to read code that does it.

Example.

```
int* pi;
int x, z[10];

pi = &z[0];  /*pi points to the first element of z*/
x  = *(pi+2); /*same as x=z[2]*/
pi = pi+4;  /*pi now points at z[4]*/
x  = *pi;   /* x gets z[4] */
```

It is crucial to understand that `pi=pi+4` doesn't add `4` to whatever address value of `pi`; if you must think in addresses, it adds `4 x sizeof(int)` to `pi`, where `sizeof(int)` gives the size of the memory cell used by `int`.

So, if you increment a pointer `p` that points to a `float` (say), or. more properly an array of `float`, then the compiler will increment `p` appropriately.

Note: `&z[4]` and `z+4`,

have exactly the same *pointer* values.

3.7 Strings in C++

3.7.1 Introduction

In C, there is no *native* 'text-string' type; and that was also the case in C++ until ten years ago. The *was* a convention to use a *null-terminated* (`'\0'`) array of `char`; these we call a 'C-strings'; C-strings present some problems, all of them to do with C's (and C++'s) low level view of arrays, see section 3.6.1.

The C++ *standard library* has improved the situation by including a `string` class, which provides a much safer and higher-level text-string implementation. I will introduce standard `string` in the subsection following.

I advise that standard `string` be used where possible, but for the sake of tradition and because there are situations where they may still be heavily used, I'd better cover C-strings in some detail.

3.7.2 C-strings

We note again that C-string, e.g. `"this is a string"`, is *not* a native type. There are two sets of support for this convention:

- The `strXXX` functions, whose prototypes are in `<cstring>`, that use null-terminated character arrays as strings, eg. strlen(s) returns the length of `s`, i.e. the number of characters up to the null terminator.
- Strings delimited by double-quotes, e.g. `"qwerty"`, are accepted as string constants.

  `char s[] = "qwerty";` is represented in memory as seven characters:

  `q w e r t y \0`; don't forget the `\0` – a very common error.

We have not yet covered *pointers*, but it is worth pointing out that `char *s;`

has a lot in common with

`char s[7];`

Example:

```
char s[10];
s[0]='a';  s[1]='b';    s[2]='\0';
cout<< s<< endl;
```

causes `ab` to be printed on the screen.

Note that,

`string = "ab";`

is *not* allowed, because whole arrays cannot be assigned, instead, you must use

`strcpy(string,"ab");`

The following example implementations of `strcpy`, show, not only how *null-terminated* `char` arrays as used as *strings*, but also how

`char s[];` and `char* s;`

are used interchangeable. `char*` == string is such an idiom in C++ and C that `char *` is synonymous with *string*. Nevertheless, we will have cause to develop a *String* class, with more secure properties.

Simple `char` array.

```
/*--- copies f(from) to t(to) ----*/
void strcpy(char t[], char f[])
{
  int i=0; char c;
```

```
  c=f[i];
  t[i]=c;
  while(c!='\0'){
    c=f[i];
    t[i]=c;
    i=i+1;
  }
}
```

Using a common C / C++ *idiom*.

```
void strcpy(char t[], char f[])
{
  int i=0;
  while((t[i]=f[i])!='\0')
     i++;
}
```

Using pointers.

```
void strcpy(char *t, char *f)
{
  while((*t=*f)!='\0'){
     t++; f++;
  }
}
```

Fancier still, and more cryptic.

```
void strcpy(char *t, char *f)
{
  while((*t++=*f++)!='\0')
     ;
}
```

Finally – making use of the fact that `'\0'` has the value `0` and that `0` is `false`.

```
void strcpy(char *s, char *t)
{
  while(*s++=*t++)
     ;
}
```

When using character arrays as strings, you must take the utmost care to ensure that you do not overflow the array. Indeed, it is a common error to treat an *uninitialised* `char` pointer (see above) as a character array / string – the problem is that, whilst it is acceptable as the equivalent to an array, it is of *zero* length!

3.8 Standard String Class

As mentioned in the previous subsection, standard `string` is now, with the advent of the Standard Library, becoming a more common way of handling text-strings.

Since `string` is a class, we again touch on classes before we have dealt with them in detail. But don't worry, except for a minor point, `string` can be treated largely like a native *type*. In chapter 13, we will mention the development of our own string class.

Program `hellostd.cpp` shows a version of `Hello, world!"` that uses the standard `string`.

```
//----- hellostd.cpp -----------------------------
//Hello, World using Std lib. 'string'
//j.g.c. 5/1/98
//------------------------------------------------
#include <iostream>
using namespace std;
int main()
{
  string s1="Hello";
  string s2="world";
  string s3;
  cout<< s1<< endl;
  cout<< s2<< endl;

  cout<< s3<< endl;

  s3= s1 + s2;
  cout<< s3<< endl;

  s3+= s1;
  cout<< s3<< endl;
  return 0;
}
```

Dissection of hellostd.cpp

1. `string s1="Hello";` defines an *object* `s1` that is an *instance* of standard class `string`. It initialises it with `"Hello"`. `"Hello"` happens to be a C-string, but C++ can handle the conversion – just as it can convert `int` to `float`.

2. `string s3;` defines a `string` `s3` and initialises it to an empty string.

3. `cout<< s1<< endl;` writes `s1` to the screen. We will see more details of this later, but suffice to say that `string` provides an `overloading` of the operator <<; similarly >>.

4. `s3= s1+ s2;` and `s3+= s1;` are further examples of operator overloading, in this case + is overloaded as a *concatenation* operator.

3.9 Vector

The name **vector** *here has almost nothing to do with the vectors that we use in graphics.*

In addition to `string` and a pile of other useful types, the C++ standard library includes an *array* class called `vector`. Java programmers should think of `ArrayList`. In general, `vector` is a lot safer to use than a built-in array.

When you declare a `vector` container you must declare the type of its contents, for example, `vector<double> v1;`, i.e. the declaration has a *type parameter*. The use of the type parameter is an example of something called *template* in C++ and *generic* in Java. The next section contains a brief introduction to templates.

The program below gives a comparison of built-in arrays with `vectors`; it show also how to use *iterator*s and the `sort` *algorithm*.

```
//------ VectorTest2.cpp --------------------------
// tests of std::vector
// j.g.c. 2003/02/20, 2006-09-13
//-------------------------------------------------
#include <iostream> // for cout<< etc.
#include <algorithm> // for sort
#include <vector>
#include <cstdlib> // for rand

using namespace std;

int main()
{
  const int n= 10; unsigned int rnseed= 139;
  double d[n];
  srand(rnseed); // initialise random number generator.
  //cout<< "Max. random number= "<< RAND_MAX<< endl;
  for(int i= 0; i< n; i++){
    d[i]= (double)rand()/(double)RAND_MAX; // numbers in 0.0 to 1.0
  }
  cout<< "\nArray of random numbers\n"<< endl;
  for(int i= 0; i< n; i++){
    cout<< d[i]<< " ";
  }
  cout<< endl;

  vector<double> v1;
  for(int i= 0; i< n; i++){
    v1.push_back(d[i]);
  }

  cout<< "\nIterating through vector, one way \n"<< endl;
  vector<double>::iterator iter;
```

```
for(iter = v1.begin(); iter!= v1.end(); iter++){
  cout<< *iter<< " ";
}
cout<< endl;

cout<< "\nIterating through vector, another way \n"<< endl;
for(int i = 0; i< n; i++){
  cout<< v1[i]<< " ";
}
cout<< endl;

sort(v1.begin(), v1.end());
cout<< "\nSorted vector\n"<< endl;
for(int i = 0; i< n; i++){
  cout<< v1[i]<< " ";
}
cout<< endl;

reverse(v1.begin(), v1.end());

cout<< "\nReverse sorted vector\n"<< endl;
for(int i = 0; i< n; i++){
  cout<< v1[i]<< " ";
}
cout<< endl;

cout<< "\nFront, back elements of vector, and size()\n"<< endl;
cout<< v1.front()<< " "<< v1.back()<< " "<< v1.size();
cout<< endl;

cout<< "\nYou can modify the vector (x 10.0)\n"<< endl;
for(int i = 0; i< n; i++){
  v1[i] = v1[i]*10.0;
}
for(unsigned int i = 0; i< v1.size(); i++){
  cout<< v1[i]<< " ";
}
cout<< endl;

cout<< "\nYou can have vectors of strings\n"<< endl;
vector <string> v2;
v2.push_back("The");  v2.push_back("quick");
v2.push_back("brown");  v2.push_back("fox");
v2.push_back("jumped"); v2.push_back("over");
v2.push_back("the");  v2.push_back("lazy");
v2.push_back("dog's"); v2.push_back("back");

for(unsigned int i = 0; i< v2.size(); i++){
  cout<< v2[i]<< " ";
```

```
}
cout<< endl;

sort(v2.begin(), v2.end());
cout<< "\nSorted\n"<< endl;
for(unsigned int i = 0; i< v2.size(); i++){
  cout<< v2[i]<< " ";
}
cout<< endl;

cout<< "\nYou can assign vectors \n"<< endl;
vector<string> v12 = v2;
for(unsigned int i = 0; i< v12.size(); i++){
  cout<< v2[i]<< " ";
}
cout<< endl;

cout<< "\nYou can remove elements from vectors \n"<< endl;
remove(v12.begin(), v12.end(), "lazy");
for(unsigned int i = 0; i< v12.size(); i++){
  cout<< v12[i]<< " ";
}
cout<< endl;

cout<< "\nYou can search vectors \n"<< endl;
vector<string>::iterator it;
it= find(v12.begin(), v12.end(), "fox");
cout<<*it<< endl;

cout<< "\n... and then erase that element \n"<< endl;
v12.erase(it);
for(unsigned int i = 0; i< v12.size(); i++){
  cout<< v12[i]<< " ";
}
cout<< endl;

vector<string> v22;
cout<< "\nYou can copy using copy, but use resize first \n"<< endl;
v22.resize(4);
copy(v12.begin(), v12.begin()+3, v22.begin() );
for(unsigned int i = 0; i< v22.size(); i++){
  cout<< v22[i]<< " ";
}
cout<< endl;

vector< vector<string> > vv;
cout<< "\nYou can have vectors of vectors, but use > > in decl. \n"<< endl;
vv.push_back(v2);
vv.push_back(v12);
```

```
  vv.push_back(v22);
  for(unsigned int i = 0; i< vv.size(); i++){
    cout<< "vector "<< i<<": ";
    for(unsigned int j = 0; j< vv[i].size(); j++){
      cout<< vv[i][j]<< " ";
    }
    cout<< endl;
  }
  cout<< endl;

  return 0;
}
```

3.10 Templates, brief introduction

When you declare a `vector` container you must declare the type of its contents, for example, `vector<double> v1;`, i.e. the declaration has a *type parameter*. The use of the type parameter is an example of something called *template* in C++ and *generic* in Java. This section contains a brief introduction to templates.

The major use of templates is in collection classes; we are not yet ready to develop our own collection class, so for the meanwhile we'll have to get by with the template examples here and in the section that introduces `vector` (section 3.9).

3.10.1 Template Functions

Overloaded Functions

The two *overloaded* `swap` functions shown below clearly invite use of a type parameter.

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

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

Overloading function names – Ad Hoc Polymorphism The overloading of function names is yet another example of *polymorphism*, in this case called *ad hoc polymorphism*. It is called *ad hoc* because, even though the name of the function (swap) is the same, there is nothing to ensure that they are similar in other than name; e.g. we could define `int swap(String s)` which gives the length of the String!

Template Function

A template version of `swap` is shown below:

```
template <class T> void swap(T& p, T& q)
{
  T temp = p;  p = q; q = temp;
}
```

The program `swapt.cpp` shows use of this `swap`. Here, we are able to swap `int`s, `float`s, and `String` objects. In the case of user defined classes, the only requirement is that the assignment operator = is defined.

```
//--- swapt.cpp ---------------------------
// j.g.c. 10/4/97, 2/5/98, 14/1/99
//-----------------------------------------
#include <iostream>
#include <string> // using std string

template <class T> void swap(T& p, T& q){
  T temp = p; p = q; q = temp;
}
int main()
{
  int a = 10, b = 12;
  float x = 1.25, y = 1.30;

  cout<< "a, b = "<< a<< " " << b<< endl;
  swap(a, b);
  cout<< "a, b (swapped) = "<< a<< " " << b<< endl;

  cout<< "x, y = "<< x<< " " << y<< endl;
  swap(x, y);
  cout<< "x, y (swapped) = "<< x<< " " << y<< endl;

  string s1("abcdef"), s2("1234");
  cout<< "s1 = "<< s1<< endl; cout<< "s2 = "<< s2<< endl;

  swap(s1, s2);

  cout<< "swapped: ";
  cout<< "s1 = "<< s1<< endl; cout<< "s2 = "<< s2<< endl;
  return 0;
}
```

3.11 Polymorphism – Parametric

Templates are another form of *polymorphism*; using templates one can define, for example a polymorphic `List` – a List of `int`, or `float`, or `string`, etc... Likewise we can define a polymorphic `swap`. Unlike the polymorphism encountered in chapter 11 (`Person` and `Cell` class hierarchies), the type of each instance of a generic unit is specified at compile time – *statically*.

Hence, the polymorphism offered by templates is called *parametric polymorphism*.

3.12 Constants and their Types

- `1234` is an `int`; `123456L` (or `l`) is long.
- `0x42` is *hexadecimal*; `037` is *octal*.
- `'x'` is `signed char`.
- "x" is a string constant, see above.
- Mixing up 'x' and "x" is a common error – and can be difficult to trace.
- `123.4` is `double`, i.e. *floating-point* constants default to `double`.

3.13 Declaration versus Definition

In C++ and C, it is common to make a distinction between *declaration* and *definition*.

- A *declaration* declares the properties of a variable or function.
- A *definition* both *declares* **and** allocates space for variables, or *declares* a function **and** defines it what does.

const qualifier Declaring a variable `const` means it cannot be the target of an assignment; eg. `const int limit = 999;`, any later statement of the form `limit = ...` will generate a compiler error.

3.14 Arithmetic Operators

`+, -, *, /, %+;`

% is *modulus* — remainder after division; e.g. `5%3 -> 2`.

/ is division; it means something different (or sort of different) whether we are using it on `int` or `float`; in integer, it truncates the result; e.g. `5/3 -> 1`.

3.15 Relational and Logical Operators

Relational operators between arithmetic types:

`>, >=, <, <=, ==, !=`

Logical operators on *logical* (`bool`):

- `||` – inclusive *or*.
- `&&` – *and*.
- `!` – unary complement. `!true -> false`.

3.16 Type Conversions and Casts

C, and to a lesser extent, C++, are quite liberal about *mixed-mode* expressions, there is automatic promotion from *narrower* to *wider*; the promotion ranking is something like: `char, int, long, float, double`.

In an expression, or a sub-expression delimited by `( ... )`, all components are implicitly *cast* to the *widest* type that is present in the expression.

It is often better, from the point of view of readability and for debugging, to use an explicit type conversion, e.g. `float(2)`.

Casts versus conversion functions Whilst C used *cast* operators, e.g `(float)2`, C++ has, in addition, conversion functions, e.g. `float(2)`. C++ has other casts that we will deal with later.

3.17 Assignment Operators and Expressions

For most binary operators, e.g. `+, -, *, /,` there is a corresponding *assignment operator*.

Thus, `i+=2;` is equivalent to `i=i+2;`.

The general form is `<op>=`,

where `<op>` is one of:

`+, -, *, /, %, <<, >>, &, ^, |`.

3.18 Increment and Decrement Operators

The *pre-increment* expression `++n` is equivalent to `n+=1` (see above), which in turn is equivalent to `n = n + 1`.

Recall: any assignment statement is also an expression, e.g. `a = b = 0;`

The *post-increment* expression `n++` is different, but only subtly. Compare:

```
n = 5;                          n = 5;
x = n++;                        x = ++n;
//here x==5 the old value       //here x==6 the new value of n
          //n==6 in both cases
```

There are also -- (decrement) operators.

3.19 Conditional Expressions

```
if(a>b)
  z = a;
else
  z = b;
```

can be replaced with:

```
z = (a > b) ? a : b;
```

The general form for a *conditional expression* is:

```
(<logical expression>) ? <expr1>:<expr2>
```

If the logical expression is *true*, the value of the expression is `expr1`, otherwise the value is `expr2`.

3.20 Bitwise Operators

- Bitwise *and*: `&`.
- Bitwise *or*: `|`.
- Bitwise *exclusive-or*: `^`.
- Bitwise *unary ones-complement*: `~`.
- Shift left by n bits: `<<n`.
- Shift right by n bits: `>>n`.

Bitwise operators can be used only on operands of the integer family. Be careful to distinguish from *logical - boolean* operators: `&&`, `||`.

3.21 Precedence and Order of Evaluation of Operators

My advice is, if in the slightest doubt, use brackets. If necessary, see (Stroustrup 1997b).

Note: when you *overload* an operator in a class, e.g. + (covered later), the operator retains its original precedence.

Chapter 4

Control Flow

4.1 Introduction

I think that this chapter has very little that is new for Java programmers.

As in Java and other *block structured* programming languages, we normally have the following flow of control constructs:

Sequence The normal stepping through from one statement to the next;

Selection To enable selection, for the next statement or block, from amongst a number of possibilities.

Repetition Repetition of a single statement or of a block. These may be deterministic, such as `for(...)`), or non-deterministic, such as `while(...)`, and `do...until`.

C++ also has `goto`, `break`, and `continue` which can be used for *unstructured* interruptions of control flow.

4.2 Statements and Blocks

Block A *block*, or *compound-statement* is a group of statements enclosed in `{...}`. Syntactically, a *block* may replace a single statement. Actually, in C++, as in C, a block has an associated *activation* and scope; in that sense it is like a function, but without parameters.

In what follows, we will tend to use the term *statement* to signify either *compound* or *single* statement – always understanding that a compound statement can take the place of a single statement.

In addition, *statement* includes selection and repetition statements – including the statements or blocks that they govern.

It is worth noting too that, syntactically, `;` is a statement – a *null* statement, that does nothing.

4.3 Selection

4.3.1 Two-way Selection – if else

```
if (<logical expression>)
     <statement1> //performed if expression true
else
     <statement2> //performed if false
```

Remark: recall that C++ doesn't really differ between numeric values and logical values: any *non-zero* (including negative) value is taken as *true*, a *zero* value (including a `NULL` pointer value) is taken to be *false*.

4.3.2 Multi-way Selection - else if

Actually, `else if` is *not* a separate construct, it is just an `else` that happens to be governing an `if` statement,

The following is a multi-way selection: if `<expr1>` is *true*, `<stmt1>` gets executed; then `<expr2>` is evaluated and if *true*, `<stmt1>` gets executed;

...

Finally, if none of the previous is *true*, `<stmt4>` gets executed.

```
if(<expr1>)
     <stmt1>
else if (<expr2>)
     <stmt2>
else if (<expr3>)
     <stmt3>
else
     <stmt4> //otherwise -- default
```

The previous construct – a so-called *if-else ladder* is well worth learning off – many programmers prefer it to the more obvious, but more difficult to use `switch`, see below.

Note the indentation style suggested; since each *rung* of the ladder is essentially *equal*, even though they are evaluated sequentially, there is no reason to further indent each `else`, indeed, to my mind, to do so would confuse the reader of the program.

4.3.3 Multi-way Selection – switch - case

```
switch (<expr>){

  case <const-exp1>: <stmnt(s)1> [break;]
  case <const-exp2>: <stmnt(s)2> [break;]
  case <const-exp3>: <stmnt(s)3> [break;]
  default: <stmnt(s)d>; [break;]

}
```

<expr> must evaluate to an integer; in addition, the case expressions must be a constant integer.

The switch is much the same as the *if-else ladder* in the previous section; there is, however, one big difference, and one that can be a big trap to the unwary: upon executing (say) <stmnt(s)1>, above, control will *fall-through* into <stmnt(s)2> and <stmnt(s)3>, etc. It will execute these *without* evaluating any condition: it is not the case expressions that alter the flow of control, but the switch.

If you don't want this to happen, you must use the break; statement, which causes control to jump to the end of the switch – to just outside the } at the end.

4.4 Repetition

4.4.1 while

```
while(<expr>)
      <statement>
```

First, <expr> is evaluated, if *true* <statement> is executed, then <expr> is evaluated again, and so on ....

while(true) is *do-forever* – normally with a selection involving some form of *exit*, e.g. break. This often appears as while(1).

The following program sums the first n integers,

```
#include <iostream>
int main()
{
  int i=1;sum=0;n=4;

  while(i<=n){
```

```
    sum+=i;
    ++i;
  }
  cout<< sum<< endl;
}
```

4.4.2 for

```
    for(<expr-init>;<expr-continue>;expr-iter)
            <statement>
```

is equivalent to

```
        <expr-init>
        while(<expr-continue>){
          <statement>
          <expr-iter>
        }
```

`for(;;){...}` is allowable and often used for *do-forever*, i.e. the same as `while(true)`.

```
int a[n],i;
for(i = 0; i < n; i++)a[i] = 0;
```

is the C++ idiom for traversing `n` elements of an array – since the array indices go `0, 1, 2, ..., n-1`.

4.4.3 do - while

```
     do
          <statement>
     while (<expr>);
```

`<statement>` is always executed at least once;

4.5 break and continue

`break` causes the innermost enclosing *repetition loop* or *switch* to be exited immediately — to just outside the `}` at the end of the block.

`continue` causes the remainder of an iteration block to be bypassed and the next iteration to be started immediately, i.e. avoiding the remainder of the current repetition.

4.6 goto and Labels

`goto` can cause a jump to any labelled statement, anywhere in the entire function. For example,

```
  goto label1;
...

label1: i = 0;
        ...
```

In case you haven't heard, `goto` is frowned upon, and in the vast majority of cases it can be avoided.

Chapter 5

C++ Program Structure

5.1 Introduction

As we know already, a C++ program can be constructed entirely of functions, i.e. unlike Java, which requires all functions/methods to be members of classes. Java also requires execution to proceed via object creation and object method calls; in C++ we can call functions on their own.

This chapter covers the definition, declaration and calling of functions.

5.2 Basics

We have covered most of this in Chapter 2, but some of it is worth repeating.

The syntax of the *definition* of a function is:

```
<return-type> <function-name>(<parameter declarations>)
{
   <declarations of local variables>
   <statements>
}
```

Example.

```
int add(int a, int b) // a and b are also local variables
{
        int value; //local variable

        value = a + b;
        return;
}
```

`return <expression>;`

returns the value of `<expression>` to the calling point, with `<expression>` being converted to `<return-type>` as necessary.

The caller can ignore the returned value – it is not a compilation error. Hence,

```
int x= 10,y= 20;
p = add(x, y);
add(x, y); //syntax okay, but not much use!
```

are both legal, i.e. *syntactically correct*, but the second isn't very sensible — *semantically*.

In a definition, you must explicitly state `void` for `<return type>` if no value is returned; in this case, `return;` is optional.

If there are no parameters, the parameter list can be replaced by and empty parameter list – `()` – or by explicit `(void)`. In C, the `(void)` is essential.

5.3 Declarations of Functions – Prototypes

In C++, all functions must be *declared* before they are called – analogous to variables.

Syntax:

`<return-type> <function-name>(<parameter declarations>);`

Example.

```
int add(int a, int b); //note the ;  in the declaration
```

i.e. it has no *implementation / body*.

Function prototypes are a a bit of a nuisance, i.e. having to declare *and* define; as we know Java eliminates the separate declaration. But it's part of C++ and it won't change.

Header Files for Libraries and Classes It's a nuisance to have to declare many functions at the top of a program, so it's common practice to include all prototypes for a library or class in a header file, and `#include` the appropriate header file where any of the functions is used.

In Chapter 2, we have already encountered a small library of functions contained in a source file `funs.cpp`. In that case we put our prototypes in `funs.h`.

Then, we `#include "funs.h"` at the declaration part of all source files that use any of the functions in `funs`.

Thus, (a) you are saved a lot of typing, but more importantly, (b) you need maintain only one set of prototypes.

5.4 Function parameters

5.4.1 Parameters

You can pass to functions values of the following types:

- All elementary types `int, char, float,` etc. ...
- Pointers — to any type.
- References
- Records — structs
- Objects
- Arrays

There is no limit to the number of parameters that a function can have.

5.4.2 Pass-by-value parameters

In C++, the default parameter passing method is *pass-by-value* or sometimes called *pass-by-copy*.

The following function adds two floats and returns the result in a float:

```
float addf(float a, float b)
{
  float c=a+b;
  a = a + 10; // to demonstrate 'pass-by-value'
  return c;
}
```

I have deliberately inserted the nonsense instruction

`a = a + 10;`

for demonstration purposes; whilst this has an effect on the *local* variable `a` it has *no* effect on the argument in the caller. Thus:

```
float x=10, y=20, z=0;

z = addf(x, y);
cout<< x<<", "<< y<<", "<< z << endl;
```

will result in `10, 20, 30`

5.4.3 Pass-by-reference parameters

The following function `swap` uses reference parameters:

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

`swap` has an effect on the variables `x, y` in the caller as it must do if it is to be of any use. Thus:

```
int x = 10, y = 20, z = 0;

swap(x, y);
cout<< x<<", "<< y<< endl;
```

will result in `20, 10`

Notice that the caller need not make any special indication of the *reference* nature of the arguments — that is all handled by the definition of the function: `int& a, int& b`.

There is no equivalent in Java; in Java all primitive types are *pass-by-value*.

Just to drive the point home, let us examine a *wrong* `swap` which uses *pass-by-value*.

```
void swapSilly(int a, int b) //wrong, pass by value
{
     int temp=a;
     a=b;
     b=temp;
}
```

Now, `swapSilly` does **not** have an effect on the arguments `x, y` in the caller, and the function has no effect. Thus:

```
int x = 10, y = 20, z = 0;

swapSilly(x, y);
cout<< x<<", "<< y<< endl;
```

will result in `10, 20`

5.4.4 Programmed pass-by-reference via pointers

Just for completeness, we will include a version of `swap` which demonstrates how pass-by-reference can be *programmed* via *pointers*. This is how it must be done on C, which does *not* have a reference type qualifier.

The following function `pswap` uses pointer parameters:

```
void pswap(int* pa, int* pb)
{
     int temp = *pa;
     *pa = *pb;
     *pb = temp;
}
```

Notice that it is what `pa, pb` *point-to* that are swapped, *not* `pa, pb` themselves, i.e. `*pa, *pb` — *dereferenced*.

`pswap` has an effect on the variables pointed-to by the caller arguments. Thus:

```
int x = 10, y = 20;
    pswap(&x, &y);
    cout<< x<<", "<< y<< endl;
```

will result in `20, 10`

Note that in this case the caller must pass pointers to the variables, i.e. `&x, &y` are passed. In addition to the extra complexity of the function, this *programmed pass-by-reference* error-prone and generally less satisfactory than proper *pass-by-reference*.

It is worth noting that in the case of `pswap` the pointer values are still *passed-by-value* — it's just that the values are pointers, and so can be dereferenced to access the variables that they reference — in the caller!

5.4.5 Semantics of parameter passing

The distinctions between the previous examples can be clearly defined by considering the detailed semantics of parameter passing by value and by reference.

Pass-by-value Recall the example of pass by value. Here `x, y` are local variables in the caller; when they are created, they are initialised with the values `10, 20`.

```
int x = 10, y = 20;
```

When `swapSilly` is called, `swapSilly(x, y);` the following happens: variables local to `swapSilly` are created `(int a, int b)` and these are initialised with the values of `x, y` — almost as if we had the definition: `int a = x, int b = y`. Hence, computations involving `a, b` are on these entirely separate and local variables.

Pass-by-reference Using the same example, `x, y` are local variables in the caller; when they are created, they are initialised with the values `10, 20`. Now we can define a *reference* `int& rx` and initialise it with `x` – *not the value of* `x`, `rx` is an *alias* for `x` – whatever is assigned to `rx` is assigned to `x`: `x` and `rx` refer to the same object in memory.

```
int x = 10, y = 20; int& rx = x;
```

Likewise when `void swap(int& a, int& b)` is called, `swap(x, y);` the following happens: reference variables are created `(int& a, int& b)` and these are initialised with variables `x, y` — almost as if we had the definition: `int& a = x, int& b = y`. Hence, computations involving `a, b` also involve `x, y`.

Pass-by-reference via pointer Again using the same example, `x, y` are local variables in the caller; when they are created, they are initialised with the values `10, 20`.

Likewise when `void pswap(int* pa, int* pb)` is called, `pswap(&x, &y);` the following happens: temporary pointer variables are created, and these are initialised with pointers to variables `x, y`, i.e. as if `(int *px = &x, int* py = &y)`.

Pointer variables local to `pswap` are created `(int* pa, int* pb)` and these are initialised with the values of temporaries `px, py` — almost as if we had the definition: `int* pa=px, int* pb=py`.

Computations involving `pa, pb` are on these entirely separate and local pointer variables, but, being pointer variables, they can be *dereferenced* to access the actual variables `x, y`. Hence, computations involving `*pa, *pb` also involve `x, y`.

Java pass by value only Well, sort-of. All *elementary* types, e.g. `int, float, double` are passed by value only. Thus, a function like `swap` cannot be done in Java.

On the other hand, just like arrays in the next section, in Java all non-elementary objects are *always* references.

5.4.6 Arrays as Parameters

Arrays are always *pass-by-reference* — though implicitly so.

When passing arrays as parameters, you need declare only that it is an array – you need not give its length. For example, a function `getline` which reads a *line* of characters:

```
int getline(char s[],int lim)
{
...
s[i]=c;// or *(s+i)= c;
...
}
```

As we have already noted in chapter 6, C++ arrays are closely related to pointers, hence the following is *entirely* equivalent:

```
int getline(char *s, int lim)
{
...
*(s+i)=c; // or, s[i]=c;
...
}
```

Function `getline` can be called either as:

```
int nchars;
const int n=100;
char s[101];

nchars = getline(s, n); // or
nchars = getline(&s[0], n);
```

5.4.7 Default parameters

In a function definition a parameter can be given a default value.

Example.

```
void init(int count=0, int start=1, int end=100)
{
// body  ...
}
```

Call:

```
init(); //is equivalent to
init(0, 1, 100)
//or
init(22, 23); //is equivalent to
init(22, 23, 100);
```

Only trailing parameters can be defaulted:

```
void init(int count=0, int start, int end); //ILLEGAL
```

In normal programming you should use default parameters only if you have a compelling reason to do so. However, they can be very useful in some class member functions, see later chapters.

5.5 Function return values

Functions can return the following values:

- All elementary types: int, char, float, etc. ....
- Pointers – to anything, but beware of *dangling pointers*, see Chapter 3.
- References – but beware of *dangling references*, see Chapter 3.
- Structs.
- Objects.
- They cannot return arrays, but, of course, they can return a pointer to an array.
- Only *one* value can be returned.

5.6 Overloaded function names

Consider the following function `addf`, which adds `floats`.

```
float addf(float a, float b)
{
  return a + b;
}
```

If we want to provide an integer version of `addf`, we might define:

```
int addi(int a, int b)
{
  return a + b;
}
```

However, in C++, through *overloading* of function names, we can use the more natural name `add` for both, and the linker will *bind* the appropriate version — according to the types of the arguments:

```
float add(float a, float b)
{
  return a + b;
}

int add(int a, int b)
{
  return a + b;
}
```

Caller:

```
int i,j,k;
float x,y,z;
...
z = add(x, y); // float add(float a, float b) is called
k = add(i, j); // float add(int a, int b) is called
```

Function name overloading is made possible by allowing the parameter types to become part of function's identity. Note: the return type is *not* used in this *disambiguation*.

Function name overloading finds extensive use in *class*es:

- A class may have a number of *constructors*, all with the same name, but each having a different parameter list.
- To enable classes, especially within a class *hierarchy*, to exhibit uniformity of behaviour; e.g. trivially, many classes can have an *overloaded* `print` function.

5.7 Inline functions

When function `add` is called, a certain processing overhead is incurred: local variables `a, b` must be created and initialised, and, likewise, the return value must be copied to the point of call. In the case of `add` this overhead may amount to more than the simple `a + b`. With larger objects, the overhead may be more severe.

```
int add(int a, int b)
{
  return a + b;
}
```

C++ was developed with one eye on efficiency and performance, and allows the specifier `inline` as a recommendation to the compiler. If we have

```
inline int add(int a, int b)
{
  return a + b;
}
```

a call to `add(.,.)` may cause `add` to be expanded at the point of call; i.e. `a + b` is inserted at the point of call, instead of a *call*.

Programmers are always warned about overly liberal use of `inline`; apparently small functions can involve large amounts of code, which, if `inline` is used, would be replicated at each call, leading to a large executable program.

Notice: `inline` allows trade-off of space – code replicated – for efficiency – lack of call / return overhead.

5.8 External variables

An external variable is *defined* once, *and once only*, outside any function, e.g. `int globalx;`

Then, *declared* anywhere it must be used, it is brought into *scope* by *declaring* it, e.g. `extern int globalx;`

Example. A program in two files `prog.cpp, funs.cpp`:

file prog.cpp:

```
int globalx; /*defined OUTSIDE any function*/
#include "funs.h"
int main()
{
  extern int globalx;    /* can declare here but not
                            essential, as 'globalx' is in
                            SCOPE from its DEFINITION -- above,
                            until the end of the file*/
  f1();
  return 0;
}
```

File `funs.cpp`, separate from prog.cpp:

```
void f1(void)
{
  extern int globalx; //bring globalx into SCOPE
  globalx=1;
}
```

External / global variables have *static* lifetime, see section 5.12, i.e. they are created before the program starts executing, and are not destroyed until the execution is completed.

5.9 Scope of variables

The *scope* of a variable (or function) is those parts of a program where the name can be used to access the variable (or function); simply, *scope* is the range of instructions over which the name is *visible*.

Lifetime is a related but distinct concept, see section 5.12.

The scope of *automatic / local* variables defined in a function or block is from the definition beginning until the end of the function or block: they have *local* scope, they are *private* to that function; thus, functions have a form of *encapsulation*.

Local names that are reused, in the same or different source files, or in different functions or blocks, are unrelated.

Example.

```
int add(int, int);
int main()
{                               |
  int x,y,z,a,b; /* (1)*/       |scope of x, y ..a, b
                                |
  x=1;y=1;a=32;b=33;            |
  z=add(x,y);                   |
  return 0;                     |
}
int add(int a, int b)
{                          |
  int value;               | scope of a, b, value
                           |
  value=a+b;               | these a,b NOT related to above a,b
  return value;            |
}
```

Blocks are scope units Like functions, blocks are scope units. Thus, within a block, you can declare a variable and its scope will be from the point of declaration to the end brace (}) of the block.

Thus:

```
int fred(int a)
{
  int b;
  { //c is in scope only in this little block
   int c; c = 22;
  }
  b= a + 10;
  return b;
}
```

Scope Hiding Consider:

```
     void fred(int n, float b)
     {
```

```
  int c = 33, d = 16,i;

  for(i = 0; i <= n; i++){
    float d; // note these d, c are different from outer
    d = 22.5;
    int c = 49;
  }
  // here c==33, d==16.
}
```

The outer `d` — the one declared first — is *hidden* in the block governed by the `for`, by the inner declaration.

Scope of functions All functions, *everywhere* (at least in libraries that are included in the linking process and which are *declared* in the program file (e.g. in a '.h' file)), are *in scope* in *every* part of all functions. This is called *external* (or global) scope. So, functions have external / global scope by default, whereas variables are local by default.

Note: functions have global scope *independent* of declarations by `#include`.

Java has a much more uniform and safe approach to scope: no *externals* are in scope, unless they are `imported` from their package/class;

This *global scope* can cause name clash problems where you are using libraries from multiple vendors; it has been addressed by the recent introduction of `namespace`, see section 5.10.

Class scope C++ classes offer their own form of *encapsulation* and scope, see later chapters, that is more or less the same as Java.

Class members that are declared `private` are *in scope* only in member functions of the class, or in functions or classes which have been declared `friends` of the class — see later sections.

On the other hand, classes *themselves* have the same external scope as functions.

5.10 Namespaces

As we have mentioned in the previous section, the global scope of functions and classes may cause problems where you are using library software from multiple sources and in which the name name has been used.

For example, we have already used class `string` which is declared using `#include <string>`. In a later chapter, we develop our own class `String`. Now the fact that one is `string` and the other is `String` is sufficient to avoid a name clash.

But let us assume that our string is also called `string` and, moreover, that we want to use them together — e.g. to test relative performances. In that case, the declarations:

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

will cause the compiler to complain about multiple declarations of `string`.

The way around it is to declare our own `string` within its own `namespace`:

```
//---- string.h ------------------------------------
// string class.
// j.g.c. ... 23/3/97
//--------------------------------------------------
#ifndef STRINGH
#define STRINGH
#include <iostream>

namespace bscgp2{

class string {
public:
  String(const int len=0);
//etc...
}
} // end of namespace
```

In addition, all standard library stuff will have been declared under `namespace std`.

Now, we can use the two `string`s together and use the scope resolution operator `::` to discriminate between them:

```
#include <string>
#include "string.h"
std::string s1;   // standard lib string
bscgp2::string s2; // our own string
```

The use of the scope resolution operator `::` may appear clumsy, so, in cases where we are not troubled with name conflicts, we can use the declaration `using namespace` in the user program.

```
using namespace bscgp2;

// now no need for bscgp2::
```

5.11 Heap memory management

5.11.1 Introduction

Up to now, we have become used to memory management being done automatically. Thus:

```
int fred(int a)
{
  int b; //here b created automatically
  // auto int b; -- is equivalent declaration; auto = automatic
  b = a*10 + 2;
  return b;
  //here the value of b is returned, and b is deleted (destroyed)
}
```

Sometimes, but only in very special cases, we may need to take direct control of the *creation* and *deletion*. In the example, `b` – and `a` – is allocated on *the stack*.

Two significant and related factors differentiate so-called *heap* or *free* memory, and the more familiar so-called *stack* memory used by local variables.

Here, also, we find the primary use of pointers in C++ – as references for (initially *anonymous*) memory created on the heap.

- Unlike *stack* variables, which are created and destroyed automatically, *heap* the memory management (creation/deletion) of heap variables must be programmed.
- Once *created*, heap memory continues to exist until it is *deleted* by an explicit delete command.

Heap variables are created using operator `new`, and are destroyed using operator `delete`.

5.11.2 Operator new

`new` is similar to Java's `new`; Java's new returns a *reference* to the object that was created (on the heap); C++'s new returns a pointer.

Operator `new` creates a variable on the heap and returns a pointer to it. Here is a somewhat toy example:

```
int* readAndCreate()
{
  int n;
  cout<< "Give size:"<< endl;
  cin>> n;
  int* p = new int[n];
  // prompt read n ints
  for(int i = 0; i<n; i++){
    cout<< "Enter an int:";
    cin >> *(p+i); //or p[i]
  }
  return p;
}

int* pa = readAndCreate();
```

Function `readAndCreate()` creates an `int` array variable of size `n`, and returns a pointer to the caller.

Although `p` is destroyed upon return from `readAndCreate()`, what it points to is not, and `pa` in the caller, can legitimately continue to reference the array.

In fact, once created, this array variable continues to exist until explicitly de-allocated, using `delete`. Consequently, the *lifetime* of a heap variable is from its explicit creation, until its explicit deletion.

If `new` cannot accomplish the creation, e.g. if a larger block of memory is needed, than is available, then `new` will return the NULL pointer `0`. Or, if `#include <new.h>` is present, it will call an error function specified in that – typically resulting in an error message followed by termination of the program. Generally, programmers must be careful to consider the consequences of running out of heap memory.

Dangling Pointers At this point you should very carefully note the complete inadequacy of the following version of `readAndCreate()`:

```
int* readAndCreateDangling()
{
  int n;
  cout<< "Give size:"<< endl; cin>> n;
  int ar[1000]; // we assume 1000 is always greater than n
  // prompt and read n ints
  ...
  // this is partially okay; 1. a pointer to the first element of ar
  // will be returned
  // but 2. ar[] will be deleted as soon as that happens!
  return ar; //or return &ar[0]

}
```

5.11.3 Operator delete

Operator `delete` de-allocates heap memory – destroys heap variables. Thus:

```
int* readAndCreate()
{
  int* p = new ...
  // etc...
  return p;
}

int* pa = readAndCreate();
// do something useful with the array
delete [] pa;
```

The [] is necessary to signify to `delete` that the target is an array. In the case that the variable is not an array, the [] must be avoided.

```
int* p = new float;
delete p;
```

5.11.4 Anonymous variables

In

```
int* p = new float;
```

the variable initially created by `new` is *anonymous* — it has no name — however `new` returns a pointer by which it may be referenced, and, of course, this is assigned to the pointer `p`, which subsequently may be used to reference the variable.

5.11.5 Garbage

Garbage refers to the situation of *anonymous* heap memory, see section refsec:anon, whose *reference* has been deleted before the heap variable itself. Once this reference is lost, the heap memory may never again be accessed, *even to de-allocate it*!

Java has automatic garbage collection — Java can detect when an object can no longer be referenced and it then deletes the object. C++ does not have garbage collection.

Thus, garbage is in some ways the opposite to *dangling* or uninitialised references — in which the reference exists, but what it references does not.

Again in comparison to dangling references, garbage *may* be more benign – it can cause program failure only by repeated *memory leak* leading eventually to the supply *free* memory becoming exhausted.

Note: do not be confused by the English connotation of the word, *garbage* does *not* refer to uninitialised variables or incorrect data.

Java In Java, all non-elementary variables (all objects and arrays — arrays are considered to be objects) are allocated on the heap. A consequence of this is that objects *obey reference semantics* rather than value semantics; beware, this is more subtle than may appear at first – for the references are passed-by-value to functions!

In addition, Java has *garbage collection*. Thus, whilst you need to create objects with `new` you do not delete them. When the connection between a Java reference and its object is eventually broken – by the reference going out of scope, or by the reference being linked to another object – the object is subjected to *garbage collection*.

5.12 Lifetime of variables

The *lifetime* of a variable is the interval of time for which the variable *exists*; i.e. the time from when it is created to when it is destroyed; *duration, span*, or *extent* are equivalent terms for the same thing.

It is common to find confusion between *scope* and *lifetime* – though they are in cases related, they are entirely different notions: *lifetime* is to do with a period of time during the execution of a program, *scope* is to do with which parts of a program text. In C and C++, lifetime is *dynamic* – you must execute the program (or do so in a thought experiment) in order to determine it. Scope is *static* – determinable at compile time, or by reading the program text.

In the case of local variables (local to blocks or functions), and where there are no *scope-holes*, lifetime and scope correspond: scope is the remainder of the block / function after the variable definition; lifetime is the whole time that control is in that part of the program from the definition to the end of the block / function.

Example, local / automatic variables.

```
int fred(int a, int b)
  {
  int c;  /*when control passes into 'fred': local (AUTOMATIC)
          variables a,b, c are created at this time -- their
          LIFETIME starts then*/

  c= 22;
  ...
  }       /*when control reaches here, locals are destroyed*/
```

.

Thus, `c`, and `a, b` exist only for the duration of the call to function `fred`. For each call, entirely new variables are created and destroyed.

If you wanted to retain the value of `c` from call to call, you would have to use the *static* type modifier.

Static lifetime Example, static variable.

```
  int fred(int a, int b)
  {
    static int c;     /*when control passes into 'fred': local
                    variables a,b, are created at this
                    time -- their LIFETIME starts then; however,
                    since c is static, it was already created at
```

```
                    compile / link time, and it lives on until the
                    program stops*/

    c = 22;
    ...
}       /*when control reaches here, locals are destroyed but
        statics live on*/
```

Many meanings of static Most unfortunately, there are *three* overloadings of the `static` qualifier:

1 *Internal* static. In a function, a `static` variable provides *permanent* storage within the function; i.e. as in the example above. This is by far the most common.

2 *External* static. Applied to an otherwise global / external variable or a function in a source file, `static` limit the `scope` of that variable or function to the remainder of that source. Thus, this use of `{static` signifies *private*.

3 Static *class member*. In a *class* definition, a data member (variable) can be declared `static`, in which case this variable is shared amongst *all* instances of the class! Normally avoid!

Of course, all global /external variables have *static* lifetime – they exist for the full life of the program; they are *created* before the program starts executing, and *destroyed* only when it halts.

5.12.1 Lifetime of variables – summary

It is possible to summarise the various lifetime classes by classifying them according to increasing degrees of persistence, from *transient* – very short lifetime – to *persistent* – very long lifetime:

Temporary variables Used in evaluation of an expression, e.g.

`y = x * (x + 1.0) + 2.0 * x;`

will almost certainly involve temporary variables, e.g. `a, b` and `c`, which exist only during the evaluation of the expression:

```
a = x + 1.0; b = x * a; c = 2 * x;

y = a + b + c;
```

Local variables Their lifetime is from entry to their definition to exit from their block / function. These are called `automatic` in C / C++.

Heap variables Their lifetime is from *allocation* to *de-allocation*. Sometimes called *dynamic* variables.

Static variables – including global. Their lifetime is from the start of execution of the complete program, until it stops.

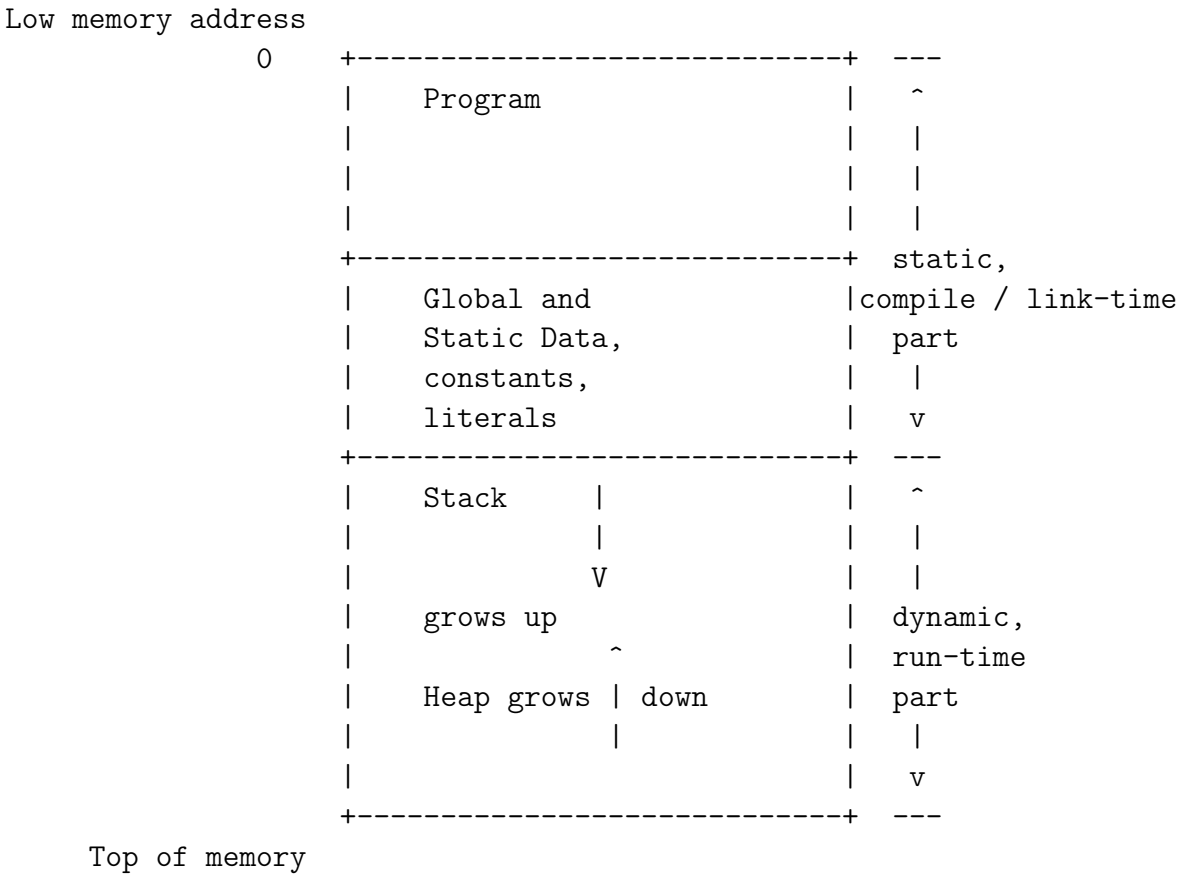
Variables held in files Their lifetime is over many program executions; they are *persistent*. Persistence of objects is of significant interest for database applications – *object-oriented databases*.

Some readers may gain a better understanding of *lifetime* by reading section 5.13 which gives a model of the run-time structure of a C++ program.

5.13 Memory layout of a C++ program – a model

See (Sethi 1996), (Louden 1993). This is a *model*; exact implementation detail may differ depending on compiler, machine architecture and operating system.

The diagram below shows the overall structure.



Memory Layout of a C++ Program

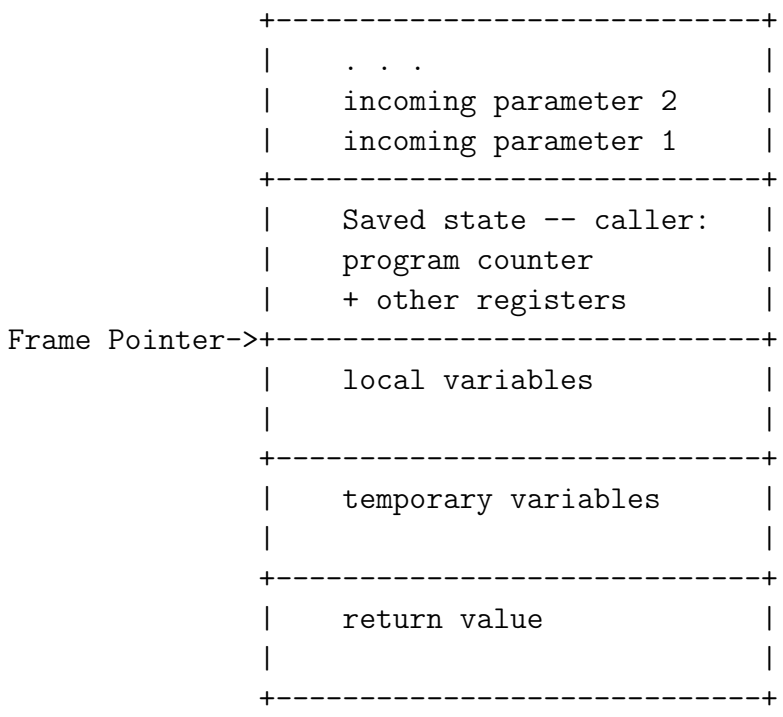
Program Executable code.

Global data etc. See sections 5.8, 5.12 and 5.9. These variables have `static` lifetime, that is their lifetime extends from when (or before) the program starts executing, until it stops executing. Hence, their lifetime is the same as that of the program code.

Stack Local variables are created on the stack. The stack grows as functions and blocks create local *environments*, and diminishes when these functions / blocks are exited. An *Environment* is typically implemented using an *activation record*, probably via a *stack-frame*, see the stack-frame diagram below.

Heap Heap memory is created by *allocate* command (`new` in C++), and is not destroyed until an appropriate *de-allocate* command (`delete` in C++) is executed. Insufficient care to de-allocate heap variables can lead to *garbage* and *memory leak*.

Next, we show a typical function *environment* – as implemented using an *activation record*, probably via a *stack-frame*.



A Stack Frame -- Activation Record

It is a matter of opinion whether a stack-frame model of an environment is helpful, useful discussions of *environments* that *avoid* any notion of implementation detail, and given in (Bornat 1987).

5.14 Initialisation

Only `extern` and `static` are guaranteed to be initialised implicitly — to some base value, e.g. for numbers, zero; Automatic or register will contain arbitrary data unless they are explicitly initialised.

If `extern` or `static` are initialised in the definition statement, the initialiser must be a constant expression, e.g.

```
int x=1; char c='a';
```

For `auto` or `register` *storage classes* there is no restriction to constant..

In array initialisation, the compiler will fill `[]` (array size) according to size of the list, if no array length specified, e.g.

```
int dayPerMonth[]={31,28, ...,30,31};
```

will define an array of `12 int`s.

5.15 Register Variables

It is possible to advise the compiler that a variable, e.g. `x` will be heavily used and should, if possible, be placed in *scratch-pad register* storage.

Example.

```
register int x;
```

This may be ignored by the compiler; without register declaration the compiler will make up its own mind; in my (JC) opinion it's best to leave it that way! On such matters, most compilers are cleverer than most programmers.

5.16 Recursion

C++ functions may be called recursively. File `fac.cpp` shows set of `factorial` functions in two styles:

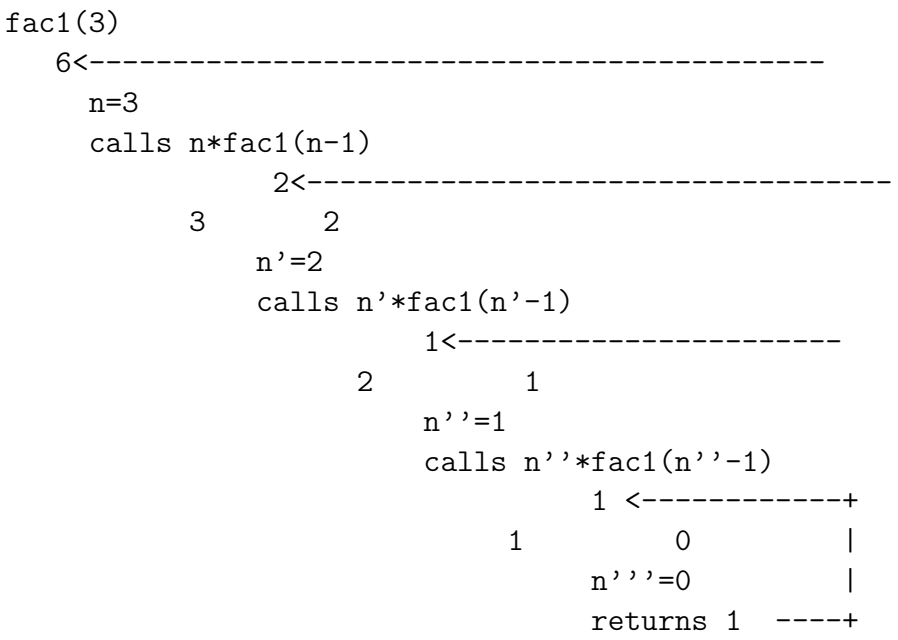
- Non-recursive.
- Recursive.

```
/*----- fac.cpp --------------------------------
  j.g.c. 6/3/95 j.g.c. 14/2/97
  comparison of functional and procedural styles
  and erroneous use of static variable in a recursive function
-----------------------------------------------*/
#include <iostream>
int facp(int n) //imperative style
{
  int i,f= 1;

  for(i=1;i<=n;i++)f=f*i;
  return f;
}


int fac1(int n) // functional style
{
  if(n<=0)return 1;
  else return n*fac1(n-1);
}


int main()
{
  int n,ff;
```

```
  cout<< "\nGive n:";   cin>> n;

  ff=facp(n);  cout<< ff<< endl;
  ff=fac1(n);  cout<< ff<< endl;
  ff=fac2(n);  cout<< ff<< endl;
  return 0;
}
                  Factorial -- comparison of styles
```

You should note that a function creates a new `environment` every time it is called; thus, in `fac1()`, we have a new `int n` for every call – indeed you should consider that there is a completely new copy of `fac1()` for each call.

Consider `fac1(3)`: `3x2x1 = 6`; the execution of it proceeds as:



By a similar tracing, you will find that `fac2` always gives the result 0 – because the `static i` *N.B. only one copy* gets overwritten by `0` in the last call; it is then `0` when it is multiplied in *all* of the statements `return i*fac2(.)`.

5.17 The C++ Preprocessor

The C pre-processor (cpp) used to be very much part of C, though reliance on it is greatly diminished in C++.

The pre-processor is quite distinct from the compiler; all files are run through the pre-processor before they are submitted to the compiler. There are four main features offered by the pre-processor.

5.17.1 File inclusion

`#include <stdio.h>` replaces that line with the contents of `stdio.h`.

`<..>` tells the compiler to look in some standard `include` directory,

`#include "mylib.h"` would search the current logged directory.

5.17.2 Symbolic constants and text substitution

e.g.

```
#define true 1
#define false 0
```

In fact you can define any replacement text:

```
#define BEGIN {
#define END }
```

would allow you to partially pretend you were using Modula-2.

5.17.3 Macro substitution

Recall the conditional expression:

`e = a>b ? a:b; // e gets max of a, b`

A `MAX` *macro* can be defined:

`#define MAX(A,B) ((A) > (B) ? (A) : (B))`

then

`e=MAX(a,b);`

behaves like a `inline` function call – the macro is inserted at the point of call. However, macros may carry certain safety penalties, and *inline* should normally be used instead.

C provides a lot of `functions` this way, e.g.

`putchar(c);` is actually a macro of

`putc(stdout,c);`

where `stdout` is a file pointer to the user standard output stream.

5.17.4 Conditional compilation / inclusion

You can effectively `program` the preprocessor to skip sections of code. You will become very familiar with this in *header* files.

Example.

```
#ifndef HDR
#define HDR
//declarations here.
#endif
```

The scheme above could be used to ensure that a header file, if invoked more than once, will have effect only on the first invocation.

Conditional compilation is also very useful for building in portability across a number of compilers or target computers. E.g.

```
#ifdef WINDOWS
  // WINDOWS code
#elif LINUX
  //Linux code
#endif
```

Chapter 6

Struct, class, union

6.1 Introduction

Long before we had object-oriented programming and *classes*, we had records or aggregate types; i.e. type which allowed you to group together more than one value; records contained only data values. Object-oriented programming extended records to contain also functions/methods.

In C++, records (aggregate types) are called `struct` or `class`.

Essentially, *in C++*, `class` and `struct` are equivalent except for an insignificant detail: in a `struct` members default to `public`, whilst, in a `class`, members default to `private`; this will not matter to most programmers, for the normal practice is to explicitly include `private` / `public` as required. In this usage, `class`, `struct` are *entirely* equivalent.

In this chapter we describe the use of `struct` as *record*, i.e. without member functions and encapsulation.

6.2 A Point struct

In two-dimensional computer graphics, a *point* is defined by an x-coordinate (horizontal), and y-coordinate (vertical).

We can define a `Point` record, and define instances `p1, p2`:

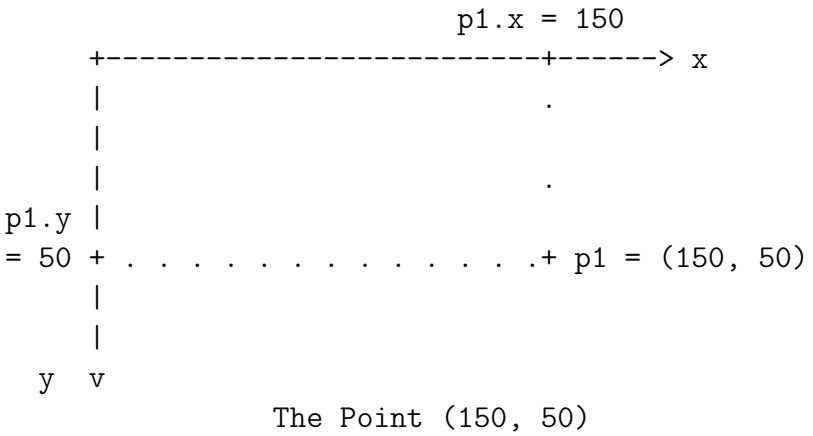
```
struct Point {int x;
              int y;
};

Point p1, p2;
```

defines two 'points' (pairs of int coordinates).

We can then assign values to the members:

```
p1.x = 150; p1.y = 50;
```

and the result is as follows:



If pp is a pointer to a Point:

```
Point *pp, p1;  int a;

p1.x = 150; p1.y = 50;
pp = &p1;
```

then a = pp->x; //access the x member

and is equivalent to,

a= (*pp).x;

the -> is provided for shorthand.

6.3 Operations on Structs

Assignment A struct can be assigned as a complete unit:

```
Point p1, p2;
p1.x = 3;
p1.y = 2;

p2 = p1; /*copies p1 values into p2*/
```

is equivalent to:

```
p2.x = p1.x;
p2.y = p1.y;
```

Copy A `struct` can be copied as a complete unit, i.e. in pass by value to functions and returning values by return.

Pointer You can take a pointer-to a `struct` with the `&` operator.

Access members Access members with `.`, or `->`, for a pointer.

File `points.cpp` shows a program which uses `Point`.

```
//---- points.cpp --------------------------------
// j.g.c. 20/3/95, 17/2/97, 14/11/98
// exercises 'Point' struct
//------------------------------------------------
struct Point{ int x;
              int y;
            };
#include <iostream>
#include <math.h> //for sqrt()

Point readPoint();
void printPoint(Point p);
float distPoint(Point p, Point q);

int main()
{
  Point p1, p2;
  float d;

  p1=readPoint();  cout<< "p1 = "; printPoint(p1); cout<< endl;
  p2=readPoint();  cout<< "p2 = "; printPoint(p2); cout<< endl;
  cout<< "Point that has largest x-value: ";
  if(p1.x > p2.x)printPoint(p1);
  else printPoint(p2);

  cout<< endl;
  d= distPoint(p1,p2);
  cout<< "distance p1 to p2 = "<< d<< endl;
  d= distPoint(p2,p1);
  cout<< "distance p2 to p1 = "<< d<< endl<< endl;

  p1.x = 3;
  cout<< "After 'p1.x = 3;' p1 = "; printPoint(p1);
  d= distPoint(p2,p1);
  cout<< "Now distance p2 to p1 = "<< d<< endl<< endl;

  Point *pp;
  pp= &p1;  //pp points at p1 -- no problems with unitialised ptr*/

  cout<< "pp is a pointer to a Point, *pp= ";  printPoint(*pp);
```

```
  d=distPoint(p2, *pp);
  cout<< "distance p2 to *pp (actually p1) = "<< d<< endl;

  pp->y = 5;
  cout<< "\npointer->member is one way of dereferencing\n";
  cout<<"  a pointer to a struct & accessing a member\n";
  cout<< "After 'pp->y = 5;' *pp = "; printPoint(*pp);

  cout<< "\n(*pointer).member is another way of dereferencing\n";
  cout<< "a pointer to a struct & accessing a member\n";

  (*pp).x = 6;
  cout<< "After '(*pp).x = 6;' *pp = "; printPoint(*pp);
  cout<< endl;

  return 0;
}

Point readPoint()
{
  Point p;

  cout<< "enter point  x,y (both ints 'space' separating) : ";
  cin>> p.x >> p.y;
  return p;
}

void printPoint(Point p)
{
  cout<< "(x = "<< p.x<< ", y = "<< p.y<<")"<< endl;
}

float distPoint(Point p, Point q)
{
  float dx = p.x - q.x;
  float dy = p.y - q.y;
  float dsquared = dx*dx + dy*dy;

  return sqrt(dsquared);
}
```

6.4 Unions

A `union` is declared using the same syntax as `struct`. Although they may look similar, they are quite different in meaning.

Whilst a `struct` contains at all times each and every component declared within it, i.e. it is a *record / tuple*, a `union` may contain but one of its components at a time; which component it is

depends what was last assigned.

Consider the example:

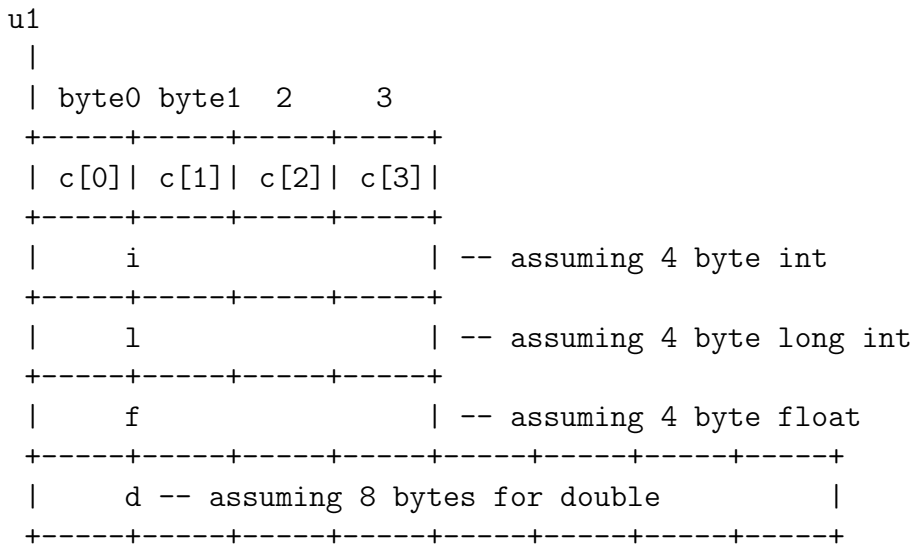
```
union Manytype{
  char c[4]; int i; long int l; float f; double d;
};
Manytype u1,u2;
```

Here we have declared two variables of union type, to contain `char[4]`, or `int`, or `long int`, or `float`, or `double`, but only one of these at any time.

Here we have declared a struct, that contains a tuple: char[4], and int, and long int, and float, and double.

```
struct Manyparts{
  char c[4]; int i; long int l; float f; double d;
};
Manyparts s1,s1;
```

From the point of view of memory layout, a typical run-time memory layout for the `union` could be:



So, if you assign something to `i`, `c` will change, and all the others too.

Now if you assign to `f`, and examine `c[0], c[1]` etc, you will get nonsense; i.e. between assignments, what a `union` stands for must not change.

On the other hand, a `union` *can* be used to `break-into` the type system: in the case mentioned above, we can assign a `float` and examine its contents as (four) `chars`.

In the case of `struct`, all the components co-exist. Hence, of course the `struct` uses more memory, and the `union` can conserve memory where you want to use only one of the components at a time.

The example program below (`strun.cpp`) compares and contrasts `struct` and `union`.

```
//---- strun.cpp -------------------------------
// j.g.c. 7/1/96, 17/2/97
// experiments with unions & structs
//----------------------------------------------
#include <iostream>

 union Manytypes{ char c[4]; int i; long int l; float f; double d;
  };
 struct Manyparts{ char c[4]; int i; long int l; float f; double d;
  };

int main() {
  int j;
  Manytypes u1, u2;
  Manyparts s1, s2;

  cout<< "\n\t\tunion:\n";

  for(j=0;j<=3;j++)u1.c[j]=j;

  cout<< "\nu.c[0..3] == \t";
  for(j=0;j<=3;j++)cout<< " "<<(int)u1.c[j]; cout<< endl;

  u1.i = 0x12345678;
  cout<< "\nu.i == "<< u1.i;

  cout<< "\nNotice, the c array has been overwritten\n";

  cout<< "\nu.c[0..3] == \t";
  for(j=0;j<=3;j++)cout<< " "<< (int)u1.c[j]; cout<< endl<< endl;

  cout<< "\t\tstruct:\n\n";

  for(j=0;j<=3;j++)s1.c[j]=j+2;

  cout<< "\nu.c[0..3] == \t";
  for(j=0;j<=3;j++)cout<< " "<< (int)s1.c[j]; cout<< endl<< endl;

  s1.i = 0x12345678; cout<< "\ns.i == Hex"<< s1.i<< endl;

  cout<< "\nNotice, the c array is unchanged\n";

  cout<< "\ns.c[0..3] == \t";
  for(j=0;j<=3;j++)cout<< " "<< (int)s1.c[j]; cout<< endl<< endl;

  return 0;
}
```

The big problem with unions is that they are not *discriminated*, that is there is no mechanism

within the type system to *remember* what was last assigned.

Hence, it may be safer to bolt a *discriminant* onto a `union` by enclosing both within a `struct`.

Here we define a *discriminant type* for the `union Manytypes`:

```
enum Utype {CHAR4,BYTE,INT,FLOAT,DBL};
```

Next, the `union` itself:

```
union Manytypes{ char c[4]; int i; long int l; float f; double d;
};
```

Then the *discriminant* and `union` aggregated together in a `struct`:

```
struct Smany{
              Utype d;
              Manytypes data;
            };
```

Finally, we show how to assign a `float`, and also an appropriate discriminant. Obviously, to be of real effect, `struct Smany` would have to be encapsulated in a proper `class`, with the data `private`, see the next chapter.

```
Smany s;

s.data.f = 1.25; // assign data
s.d = FLOAT; // assign discriminant
```

Chapter 7

Introduction to Classes and Objects

7.1 Introduction

This chapter introduces the basics of C++ *classes* and *objects*. For the moment, we will avoid *inheritance*, *polymorphism* and *dynamic / run-time binding*. We start with a very simple class, `Cell`, which reduces the class/object concept its bare essentials. At the end of the chapter we give the code for vectors and transformations in 3D.

Recall that a data *type*, such as the native data types `int`, `float`, etc., is characterized by:

1. A set of values – that can be assumed by objects of the type; for example in the C++ type **char**, the set of values is $-128, -127, \ldots, -1, 0, 1, \ldots 126, 127$.
2. A set of operations (functions) that can be legitimately performed on objects of the type; for example, for `int`, some of the functions are: `+, -, *, /`.

Users of the type do not concern themselves with the representation of the values, and, certainly, they are not encouraged to fiddle with the representation — they interact with the variables *only* through the legitimate operations. Hence `private` data and `public` operations (methods).

You probably reckon that you already know all this; fine, but it will do no harm to revise it and while we are at it we can point out the significant differences between C++ and Java.

7.2 A Memory Cell Class

7.2.1 Informal Specification of the Class

This class does nothing more than represent a single `int` value. As with *types* we are interested in: (a) *values* — the set of states an object may take on, and (b) *operations* — behaviour, what an object can do.

State We would expect a `Cell` object to be able to store the current *state*, i.e. its integer value.

Behaviour How would we like a `Cell` object to behave? It should have:

- Constructors. Since `Cell` is to be used in a computer program we need to be able to declare, define and create object of the class. We will define a single constructor, the *default constructor*, which initializes objects to have a state 0. In addition, we provide an initializing constructor – which *overload* the default constructor name.
- Inspector method. We should be able to obtain the state — but only via an interface function.
- Mutator method. We should be able to modify the state — but again only via an interface function.
- Input-output methods. We require – more for the purposes of demonstration than anything else – facilities to convert a `Cell` object into a humanly readable format; for this we provide a `print` function.

7.2.2 Class Cell

Class `Cell` class is shown below, and below that a test program. Note: in these examples, we try to keep white space to a minimum — so that it will be possible to get programs on a single OHP slide. Note also that we are (i) stuffing everything in one file — the class and the user program, (ii) we are including the implementation of the methods in the `class Cell` declaration; normally, for example, the *declaration* of the class would be in `Cell.h` and the implementation of the methods would be given in `Cell.cpp`.

```
//----- Cell0.cpp -------------------------------------
// j.g.c. 12/2/98, 8/1/99, 2003/11/29
//-----------------------------------------------------
#include <iostream> using namespace std;

class Cell{
public:
  Cell(){val_= 0;}
  void set(int val){val_= val;}
  int get(){return val_;}
  void print(){cout<<"value= " << val_<< endl;}
private:
  int val_;
};

int main(){
  Cell c;

  c.set(123);  cout<< "Cell c: "<< endl;          c.print();

  Cell* pc= &c;  cout<< "Cell* pc = &c: "<< endl;   pc->print();

  //c.val_= 345;  // remove the first "//" and see if the program compiles
  return 0;   }
```

Dissection

1. Public interface. First, we have the *interface-functions* or *methods* – which provide the *behaviour*; these are declared `public`.

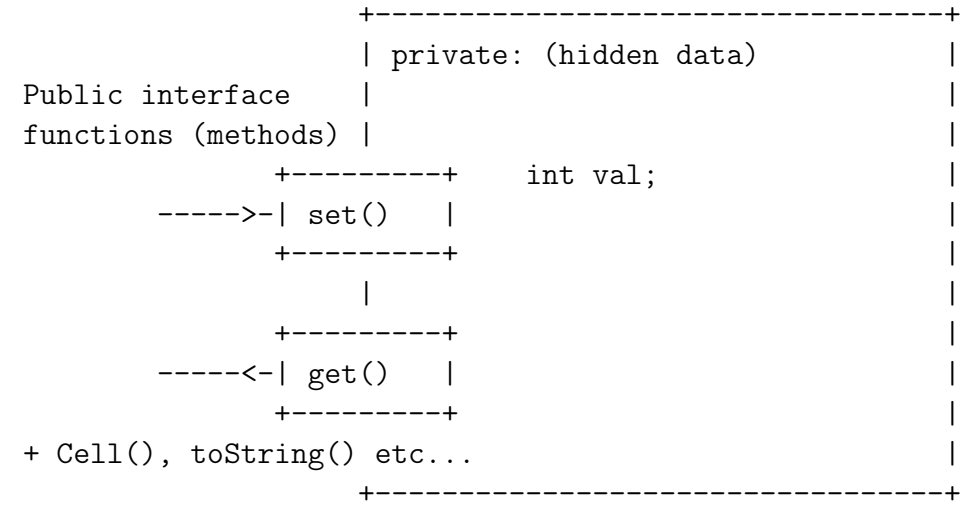
2. `public` means that the members can be directly accessed by client programs as: `instance.member` e.g. `c.set(123);` where `c` is an instance of `Cell`.

3. Constructors *must* have the same name as the `class`; often, we will have multiple constructors, all with the same name; the name sharing is allowable due to function name *overloading* – functions may share the same name, as long as they are resolvable by their parameter list (their *signature*).

4. **Private by default**. Had we left out the keyword `public`, the interface functions would have been *inaccessible* by user programs.

5. After the interface functions, we have the *representation* of the state; this is *private*; though the *private* representation is visible in the text, it is still *encapsulated* and invisible to client programs.

   Encapsulation As a consequence of *encapsulation* we can view objects, e.g. of class as capsules, containing the representation data, in this case a single datum `int val`, but these data may be accessed only through the interface functions (methods). This is shown diagrammatically:



6. After previously specifying `public`, it is necessary to revoke this directive using `private`.

7. `private` means that the member `v` *cannot* be directly accessed by client programs. I.e.

   ```
   c.val = 22;// illegal -- compiler error
   ```

   This is called *encapsulation* and provides *information hiding*.

8. Generally, the syntax for calling a method (member function), i.e. sending a *message* to an object – is: `object.method(argument)`, e.g.

   ```
   c.set(123);
   ```

Message to `c`: *set your state to* 123.

9. Notice that member data of the object *itself* can be accessed without any `.` operator.
10. In the client program, notice how `Cell c` defines an object – same as defining a variable.

    Class $\leftrightarrow$ **type, object** $\leftrightarrow$ **variable**.

7.3 Using Separate Files

Normally, we will want to write classes so that they can be used by a great many programs — all without needing to have a copy of the class in each of them. To enable this we need *two* separate class files: (a) the declaration of the class: `Cell.h`; this *declares* how to use the class; (b) the definition of the class: `Cell.cpp`; this *defines* the workings of the class. `Cell.cpp` can be compiled separately and the result stored in a library (as an object file). These files, and the new (separate) test program, are shown below.

Notice how the declaration `Cell.h` must be `#included` in the `.cpp` files. As an exercise, remove `#include "Cell.h"` from one of the `.cpp` and see what the compiler has to say.

Why is this necessary? Answer. If `Cell.h` is not `#included` in a `.cpp` file, when the compiler sees `Cell`, it has no idea what you are talking about.

```
//----- Cell.h -------------------------------------
// j.g.c. 12/2/98, 8/1/99, 2003/11/29
// minimal class, encapsulating an int value
// .h file contains just declarations
//--------------------------------------------------

class Cell{
public:
  Cell();
  void set(int val);
  int get() const;
  void print() const;
private:
  int val_;
};


//----- Cell.cpp -----------------------------------
// j.g.c. 12/2/98, 8/1/99, 2003/11/29, 2006-11-01
// minimal class, encapsulating an int value
//--------------------------------------------------
#include <iostream>
using namespace std;
#include "Cell.h"

  Cell::Cell() : val_(0){ }

  /* The above *constructor initialiser* has the same effect as
  Cell::Cell(){
    val_= 0;
  }
   it is commonly used because it is more efficient; in the above
   previous constructor, val_ gets allocated and initialised; in the
   code above, val_ get allocated and initialised to some default
   and then 0 is assigned to it.*/

  void Cell::set(int val){
    val_= val;
  }

  int Cell::get() const{
    return val_;
  }

  void Cell::print() const{
    cout<<"value= " << val_<< endl;
  }
```

```
//----- CellT.cpp -------------------------------------
// j.g.c. 12/2/98, 8/1/99, 2003/11/29
// tests Cell class
//-----------------------------------------------------
#include <iostream>
using namespace std;
#include "Cell.h"

int main()
{
  Cell c;
  c.set(123);
  cout<< "Cell c: "<< endl;          c.print();

  Cell* pc= &c;
  cout<< "Cell* pc = &c: "<< endl;   pc->print();

  return 0;
}
```

7.4 Exercises

You will find the programs introduced earlier in the chapter in my public folder `cpp4jp\progs\cell\`.

1. Write a little program `CellT4.cpp` which uses `Cell` (`Cell.h, Cell.cpp`):

   (i) Declare two `Cells`, `ca` initialised by the default constructor, `cb` initialised to state (value) of 4094;

   (ii) Print out the values of `ca, cb`;

   (iii) Next, use `set` to change the value of `ca`'s state to 3056; print out it's value to confirm.

2. Add a statement to `CellT4.cpp` as follows:

   ```
   ca.val_ = 39;
   ```

   See what the compiler says about that. Explain how to legally set the state of `ca` to 39 and correct the statement above and add a statement to print the result.

3. Add statements to `CellT4.cpp` as follows:

   ```
   ca = cb;
   cout<< "after ca = cb;" << endl;
   ca.print(); cout<< endl;
   ```

   Notice how you can assign objects.

1. Based on `Cell`, implement a class `Pair` which holds *two* state values — one we will call `key` and has type `char` and the other `value` which has type `double`. `Pair` should have the following methods: (i) two constructors — one default, the other initialising; (ii) `void set(char k, double v)`; (iii) `double get(char k)`; (iv) a `print` method; (v) I may have forgotten something — add as necessary.

   Give separate `Pair.h` and `Pair.cpp`

2. Write a small test program for `Pair` — `PairT.cpp`.

3. Develop a class `Marks` which will be capable of holding student marks (coursework and examination) along the following lines:

```
//---------------------------------------------------------
// Marks.h - StudentInfo project
// j.g.c. 2003/02/20
//---------------------------------------------------------
#ifndef MARKSH
#define MARKSH

#include <string>
#include <iostream>
#include <cstdio> // for toString/sprintf
using namespace std;

class Marks{

public:
  Marks(int ex, int ca);
  Marks();
  int overall() const;
  void print() const;

private:
  int ex_;
  int ca_;
};
#endif
```

   Method `overall` should compute the overall mark as follows:

```
  double x = ex_*0.7 + ca_*0.3;
  return (int)x;
```

4. Write a small test program for `Marks` (`MarksT.cpp`) which declares two `Marks` objects and initialises them with (exam 40, ca 60), (exam 80, ca 50); print them, and compute and print the overall marks for each.

5. (i) Add a method to `Marks` (`bool isPass()`) which tests whether a `Marks` object is pass or not (pass is overall mark `>= 40.0`);

   (ii) Include another `Marks` object in `MarksT.cpp` initialised to (exam 25, ca 40); print out all three objects along with pass/fail indication.

7.5 3D Affine Transformations

This section is added because we may use these classes for some of our graphics course. You will note that minor optimisations are in place (e.g. the data is public, so that we can *set* or *get* without a method; this because I know exactly what it will be used for, and that it will not need to be modified or extended.

You will note that we have not yet covered (i) *operators*, e.g. the *put-to* and *get-from* operators << and >> and the `friend` qualifier. Operators are like *ordinary* methods except you call them in *infix* mode. The `friend` qualifier gives access, by non members, to the `private` area of a class. For example, below << is an `ostream` operator, not a `Vector4D` operator, but it needs to acces the private data of `Vector4D`. When you have a << operator you can output an object using the normal `cout<< object` style. Likewise for output to files and for `cin>> object`.

```
// friends are not members, but have access to private data
friend std::ostream& operator<<(std::ostream& os, const Vector4D& v);
friend std::istream& operator>>(std::istream& os, Vector4D& v);
```

7.5.1 Homogeneous 3D Vector — Vector4D.h

```
//-------------------------------------------------------
// Vector4D.h
// j.g.c. 2005-03-28, 2006-10-31
//-------------------------------------------------------
#ifndef Vector4DH
#define Vector4DH

#include <string> #include <iostream>
#include <cstdio> // for toString/sprintf
#include <cmath>

#define N 4

class Vector4D{

  // friends are not members, but have access to private data
  friend std::ostream& operator<<(std::ostream& os, const Vector4D& v);
  friend std::istream& operator>>(std::istream& os, Vector4D& v);

public:
  Vector4D();
  Vector4D(double d[]);
  Vector4D(double x, double y, double z, double w);
  Vector4D(std::istream& is);

  void setAll(double val);
  Vector4D add(Vector4D v);
  Vector4D sub(Vector4D v);
  Vector4D scale(double s);
  double length();

  std::string toString() const;

  //note below public
  double d_[N];
  size_t n_;
};

#undef N
#endif
```

7.5.2 Homogeneous 3D Vector — Vector4D.cpp

```
//-------------------------------------------------------
// Vector4D.cpp
// j.g.c. 2005-03-28, 2006-10-31
//-------------------------------------------------------
#include "Vector4D.h"

#define N 4

using namespace std;

Vector4D::Vector4D(): n_(N){
  setAll(0.0);
}

Vector4D::Vector4D(double d[]) : n_(N){
  for(size_t i= 0; i< n_; i++)d_[i]= d[i];
}

Vector4D::Vector4D(double x, double y, double z, double w) : n_(N){
  d_[0]= x; d_[1]= y; d_[2]= z; d_[3]= w;
}

Vector4D::Vector4D(istream& is) : n_(N) {
  for(size_t i= 0; i< n_; i++)is>> d_[i];
}

void Vector4D::setAll(double val){
  for(size_t i= 0; i< n_; i++)d_[i]= val;
}

Vector4D Vector4D::add(Vector4D v){
  Vector4D v1= Vector4D();
  for(size_t i= 0; i< n_; i++)v1.d_[i]= d_[i] + v.d_[i];
  return v1;
}

Vector4D Vector4D::sub(Vector4D v){
  Vector4D v1= Vector4D();
  for(size_t i= 0; i< n_; i++)v1.d_[i]= d_[i] - v.d_[i];
  return v1;
}

Vector4D Vector4D::scale(double s){
  Vector4D v1= Vector4D();
  for(size_t i= 0; i< n_; i++)v1.d_[i] = s*d_[i];
  return v1;
}
```

```
double Vector4D::length(){
  double len= 0.0;
  for(size_t i= 0; i< n_; i++)len+= d_[i]*d_[i];
  return sqrt(len);
}

ostream& operator<< (ostream& os, const Vector4D& v){
  os<< v.toString();
  return os;
}

string Vector4D::toString() const{
  //ideally, I should be using stringstream here, but there
  // appears to be a problem under g++ 2.96 j.c. 2003/02/22
  char cc[80];
  sprintf(cc, "(%8.3f, %8.3f, %8.3f, %8.3f)", d_[0], d_[1], d_[2], d_[3]);
  return string(cc);
}


istream& operator>>(istream& is, Vector4D& v){
  // assumes that fields are separated by whitespace
  for(size_t i= 0; i< v.n_; i++)is>>v.d_[i];

  return is;
}
```

7.5.3 Test for Vector4D.cpp

```
/**
 * Vector4DT1.cpp
 * tests Vector4D
 * author j.g.c.
 * version 1.1; 2005-03-28, 2006-10-31
 */

#include <iostream> // for cout<< etc.
#include <string>
#include <fstream>  // for files - ofstream, ifstream
#include "Vector4D.h"
using namespace std;

int main(){

    double d[4]= {1., 2., 3., 4.};

    Vector4D x1= Vector4D(d);

    cout<<"Vector4DT1"<< endl;

    cout<< "x1 = " << x1.toString()<< endl;

    Vector4D x2= Vector4D();
    x2.setAll(10.0);
    cout<< "x2 = "<< x2.toString()<< endl;

    Vector4D x3 = x2.add(x1);
    cout<< "x3 = x2.add(x1): "<< x3.toString()<< endl;

    Vector4D x4 = x3.scale(0.25);
    cout<< "x4 = x3.scale(0.25): "<< x4.toString()<< endl;

    double len = x1.length();
    cout<< "len = x1.length(): "<< len<< endl;
}
```

7.5.4 Homogeneous 3D Vector Transformations — Transform4D.h

```
//-------------------------------------------------------
// Transform4D.h
// j.g.c. 2005-03-28, 2006-10-31
//-------------------------------------------------------
#ifndef Transform4DH
#define Transform4DH

#include <string>
#include <iostream>
#include <cstdio> // for toString/sprintf
#include "Vector4D.h"

#define N 4

class Transform4D{

  // friends are not members, but have access to private data
  friend std::ostream& operator<<(std::ostream& os,
                                  const Transform4D& v);
  friend std::istream& operator>>(std::istream& os, Transform4D& v);

public:
  Transform4D();
  Transform4D(double d[N][N]);
  Transform4D(std::istream& is);

  void setAll(double val);
  void setIdentity();
  void setTrans(Vector4D t);
  void setRotZ(double theta);
  void setRotX(double theta);
  void setRotY(double theta);
  void setScale(Vector4D s);

  Vector4D mpy(Vector4D v);

  //b = a*this
  Transform4D mpy(Transform4D s);

  std::string toString() const;

private:
  double m_[N][N];
  size_t n_;
};
#undef N
#endif
```

7.5.5 Homogeneous 3D Vector Transformations — Transform4D.cpp

```
//-----------------------------------------------------------
// Transform4D.cpp
// j.g.c. 2005-03-28, 2006-10-31
//-----------------------------------------------------------
#include "Transform4D.h"

#define N 4

using namespace std;

Transform4D::Transform4D() : n_(N){
  setAll(0.0);
}

Transform4D::Transform4D(double m[N][N]) : n_(N){
  for(size_t r = 0; r< n_; r++){
    for(size_t c = 0; c< n_; c++){
      m_[r][c]= m[r][c];
    }
  }
}

Transform4D::Transform4D(istream& is) : n_(N){
  for(size_t r= 0; r< n_; r++){
    for(size_t c= 0; c< n_; c++){
      is>> m_[r][c]; // undoubtedly wrong; test
    }
  }
}

void Transform4D::setAll(double val){
  for(size_t r = 0; r< n_; r++){
    for(size_t c = 0; c< n_; c++){
      m_[r][c] = val ;
    }
  }
}

void Transform4D::setIdentity(){
    setAll(0.0);
    for(size_t i = 0; i< n_; i++)m_[i][i] = 1.0;
}

void Transform4D::setTrans(Vector4D t){
    setIdentity();
    size_t n= n_-1;
    for(size_t c = 0; c< n; c++)m_[c][n] = t.d_[c]; //x, y, z
```

```
}

void Transform4D::setRotZ(double theta){
    setIdentity();
    double cs = cos(theta);
    double sn = sin(theta);
    m_[0][0] = cs;
    m_[0][1] = -sn;
    m_[1][0] = sn;
    m_[1][1] = cs;
}

void Transform4D::setRotX(double theta){
    setIdentity();
    double cs = cos(theta);
    double sn = sin(theta);
    m_[1][1] = cs;
    m_[1][2] = -sn;
    m_[2][1] = sn;
    m_[2][2] = cs;
}

void Transform4D::setRotY(double theta){
    setIdentity();
    double cs = cos(theta);
    double sn = sin(theta);
    m_[0][0] = cs;
    m_[0][2] = -sn;
    m_[2][0] = sn;
    m_[2][2] = cs;
}

void Transform4D::setScale(Vector4D s){
    setIdentity();
    size_t n= n_-1;
    for(size_t i = 0; i< n; i++)m_[i][i] = s.d_[i]; //x, y, z
}

/**
   * returns pre-multiplication of this*parameter
   * @param v a vector to be multiplied by this
   * @return product b = this*v
*/
Vector4D Transform4D::mpy(Vector4D a){
    Vector4D b = Vector4D();
    double dot;
    for (size_t r = 0; r < N; r++) {
      dot = 0;
      for (size_t c = 0; c < N; c++) {
```

```
        dot += m_[r][c]*a.d_[c];
      }
      b.d_[r] = dot;
    }
    return b;
}

/**
   * returns pre-multiplication of parameter*this
   * @param a transform to be multiplied by this
   * @return product b = a*this
*/
Transform4D Transform4D::mpy(Transform4D a){
    Transform4D b = Transform4D();
    double dot;
    for (size_t r = 0; r < N; r++) {
      for (size_t c = 0; c < N; c++) {
        dot = 0.0;
        for (size_t i = 0; i < N; i++) {
          dot += a.m_[r][i]*m_[i][c];
        }
        b.m_[r][c] = dot;
      }
    }
    return b;
}

ostream& operator<< (ostream& os, const Transform4D& m){
  os<< m.toString();
  return os;
}

string Transform4D::toString() const{
  //ideally, I should be using stringstream here, but there
  // appears to be a problem under g++ 2.96 j.c. 2003/02/22
  char cc[80];
  string s= string("[");
  for(size_t r= 0; r< n_; r++){
    sprintf(cc, "(%8.3f, %8.3f, %8.3f, %8.3f)\n",
          m_[r][0], m_[r][1], m_[r][2], m_[r][3]);
    s+= string(cc);
  }
  s+= "]";

  return s;
}


istream& operator>>(istream& is, Transform4D& m){
```

```
  // assumes that fields are separated by whitespace
  for(size_t r= 0; r< m.n_; r++){
    for(size_t c= 0; c< m.n_; c++){
      is>> m.m_[r][c];
    }
  }
  return is;
}

#undef N
```

7.5.6 Test for Transform4D.cpp

```
/**
 * Transform4DT1.cpp
 * tests Transform4D
 * @author j.g.c.
 * version 1.0; 2005-03-28
 */

#include <iostream> // for cout<< etc.
#include <string>
#include <fstream>  // for files - ofstream, ifstream
#include "Transform4D.h"
#include "Vector4D.h"
using namespace std;

class Transform4DT1 {

public:
  Transform4DT1();
};

Transform4DT1::Transform4DT1(){

    cout<< "Transform4DT1"<< endl;

    double d[4]= {1.0, 2.0, 3.0, 4.0};
    Vector4D xa= Vector4D(d);
    cout<< "xa = "+ xa.toString()<< endl;

    Transform4D f = Transform4D();
    f.setIdentity();
    cout<< "f.setIdentity(): \n"+ f.toString()<< endl;

    Transform4D g = Transform4D();
    g.setIdentity();
    cout<< "g.setIdentity(): \n"<< g.toString()<< endl;
```

```
Transform4D fg= f.mpy(g);
cout<< "fg = f.mpy(g) (g identity): \n"<< fg.toString()<< endl;

Vector4D xb = f.mpy(xa);
cout<< "xb = f.mpy(xa): "<< xb.toString()<< endl;

double d1[4]= {0.0, 0.0, 0.0, 1.0};
Vector4D x0= Vector4D(d1);
cout<< "x0 = "<< x0.toString()<< endl;

double dt[4]= {1.0, 2.0, 3.0, 0.0};
Vector4D t= Vector4D(dt);
cout<< "t = "+ t.toString()<< endl;

Transform4D h = Transform4D();
h.setTrans(t);
cout<< "h.setTrans(t): \n"<< h.toString()<< endl;

Vector4D x1 = h.mpy(x0);
cout<< "x1 = h.mpy(x0): \n"<< x1.toString()<< endl;

double pi= 3.1415926;
//pi/2 rotation of (1,0,0) should make (0,1,0)
h.setRotZ(pi/2.0);
cout<< "h.setRotZ(pi/2): \n"<< h.toString()<< endl;
Vector4D x1001 = Vector4D(1.0, 0.0, 0.0, 1.0);
cout<< "x1001 = "<< x1001.toString()<< endl;
x1=  h.mpy(x1001);
cout<< "x1 = h.mpy(x1001): \n"<< x1.toString()<< endl;

cout<< "\nRotation followed by translation ..."<< endl;
h.setRotZ(pi/4.0);
cout<< "h.setRotZ(pi/4.0): \n"<< h.toString()<< endl;
Vector4D x2101 = Vector4D(2.0, 1.0, 0.0, 1.0);
cout<< "x2101 = "<< x2101.toString()<< endl;
x1=  h.mpy(x2101);
cout<< "x1 = h.mpy(x2101): \n"<< x1.toString()<< endl;
f.setTrans(Vector4D(1.0, 2.0, 0.0, 0.0));
cout<< "f.setTrans(Vector4D(1.0, 2.0, 0.0, 0.0)): "<<
        f.toString()<< endl;

Vector4D x2 = f.mpy(x1);
cout<< "x2 = f.mpy(x1): "<< x2.toString()<< endl;

// incorrect order
/*
System.out.println("\nConcatenating rotation and translation ...");
g = f.mpy(h); // this is h.f
```

```
    System.out.println("g = f.mpy(h): \n"+ g.toString());
    x2 = g.mpy(x2101);
    System.out.println("x2 = g.mpy(x2101): "+ x2.toString());
    */

    cout<< "\nConcatenating rotation and translation ..."<< endl;
    g = h.mpy(f); // this is f.h
    cout<< "g = h.mpy(f): \n"<< g.toString()<< endl;
    x2 = g.mpy(x2101);
    cout<< "x2 = g.mpy(x2101): "<< x2.toString()<< endl;
}

  /** The main method constructs the test
    */
int main(){
  Transform4DT1();
}
```

Chapter 8

Inheritance

8.1 Introduction

Using *inheritance*, a *derived* class can be defined as an extension of a *base* class.

The inheriting (derived) class inherits all the behaviour and state memory of the *base class*. In addition, it is free to add its own member data and functions — so-called *extensions* or *specialisations*.

Note. The use of *specialisation* instead of *extension* has potential for confusion. The same goes for the use of the terms *superclass* and *subclass* for base class, and derived class, respectively. Thus, we will attempt to standardise on the terminology: *base class* and *derived class*, where the derived class is the one that inherits (is extended); the derived class *inherits* from the base class.

As well as *adding* member functions and data, a derived class may *overload* base functions and operators — to make them appropriate for itself. An example that we will see is the overloading of a `print` function, that is capable of printing any extra fields introduced in the derived class.

However, the real power of inheritance comes from *polymorphism* and *dynamic binding or run-time binding* via `virtual` functions.

In this chapter we cover nearly everything *twice* — once from the point of view of the `Cell` class hierarchy, and once from the point of view of `Person` class hierarchy. The repetition is intentional: the concepts here are important, and crucial to an understanding of the theory and practice of OOP. Later we will cover the same ground again for a third time, this time in terms of computer game objects.

Advanced. Ease of program extension. Run-time binding allows a class (hierarchy) to be extended *after* the program which uses it has been written — leading to the comment that: *old code can 'call' new code*; this refers to the reversal of the traditional roles, when, generally new *calling* programs are based on (calling of) old library code.

Another view of the same property is expressed in the *open-closed principle* of object-oriented programming: to be useful to client programmers, a class must be *closed to modification*; yet, using inheritance together with run-time binding, the class is also *open to extension*. Make sure that I explain the distinction between *compile-time binding* (the obvious and mots usual one) and

dynamic binding. The point of being *closed to modification* is that you don't have to modify and retest working software; the point of *open to extension* is that software can be extended — as is *always* necessary.

Liskov Substitution Principle It is important that inheriting classes respect the *Liskov Substitution Principle*; this states that it must be possible to substitute a derived object (of an inheriting class) instead of a base object without breaking the software. For example, see below, we want to be able to substitute a `Student` object (or a `Lecturer` object) for a `Person` object and the software should still work. One everyday language way of expressing this is that a `Student` **is a** `Person`. Is a is always a good thought test to run when you are thinking of using inheritance.

8.2 Example 1: Class Cell extended via inheritance

We will approach the solution in three stages; first, we show inheritance, but *without* `virtual` functions. (If you want your code to objects as in Java, you need virtual functions.) Next we demonstrate the use of `protected` as `private` but with `public` to deriving classes; Java programmers should be comfortable with the latter. Finally, we introduce `virtual` functions and the full effect of *dynamic binding* and *polymorphism*.

8.2.1 First Attempt

Program `CellR1.cpp` presents class `ReCell`, an extension of class `Cell`. It extends `Cell` to have a *backup* state that stores the last state when `set()` is used.

The extension comprises the following:

Constructor This initialises the backup value, and, via the base (Cell) constructor, the Cell part, i.e. `val_`;

Overloaded set() This first backs-up the current value. Then is uses the `Cell` version of `set()`;

Overloaded print() `print()` is also overloaded;

The :: qualifier `Cell::set(val)` insists that `set` from the base class `Cell` is to be used; in Java you would write `super.set(val)`;

Additional behaviour A function `restore` is provided;

Additional state `int backup_` stores the backup state.

That is *nearly* all. A `ReCell` object has whatever a `Cell` has, plus:

- *additional* behaviour;
- *revised* behaviour;
- *additional* state memory (data).

But there is a little more, as we see below.

```
//----- CellR1.cpp -----------------------------------
// inheritance etc. -- first version;
// see CellR2 etc for modifications
//----------------------------------------------------
#include <iostream>
using namespace std;

class Cell{
public:
  Cell() : val_(0){}
  void set(int val){val_= val;}
  int get() const {return val_;}
  void print() const {cout<<"value= " << val_<< endl;}
private:
  int val_;
};

class ReCell: public Cell{
public:
  ReCell() : Cell(), backup_(0){};
  void set(int val){backup_= get(); Cell::set(val);}
  void restore(){set(backup_);}
  void print() const
  {cout<< "value= " << get()<< " backup= " <<backup_ << endl;}
private:
  int backup_;
};

int main(){
  Cell c;
  c.set(123); cout<< "Cell c: "<< endl;  c.print();

  Cell* pc= &c;  cout<< "Cell* pc = &c: "<< endl; pc->print();

  ReCell rc;  rc.set(456);  rc.set(789);
  cout<< "ReCell rc: "<< endl; rc.print();

  rc.restore();
  cout<< "rc.restore() "<< endl; rc.print(); cout<< endl;

  cout<< "slicing: c= rc "<< endl;
  c= rc; c.print();

  cout<< "pc= & rc "<< endl;
  pc= &rc; pc->print(); cout<< endl;

  return 0;    }
```

Dissection of CellR1.cpp

1. Firstly, note that class `Cell` is unchanged.
2. `ReCell` contains an additional field `backup_`. This means that a `ReCell` object has two fields: `val_` and `backup_`.
3. `c= rc;` is a legal assignment, even though they are defined as different *types*.

   This is because, owing to the inheritance `rc` *is-a* `Cell` — in addition to being a `ReCell`.
4. *Slicing*. In the assignment `c= rc;`, slicing occurs, and only the `Cell` part of `rc` is copied. If you like, think of it like this: `c` would have nowhere to put the additional `backup_` part.
5. `pc= &rc;` is a also a legal assignment; however, this time, slicing does not occur — although we note that, in any case, no member data are copied (`pc` is a pointer) — and `pc` takes on the *dynamic* type `ReCell*` (pointer-to- `ReCell`). Java programmers will say *big deal*, Java does this all the time.
6. However, `pc->print();` acts as if *slicing* has taken place; this is because the *static* type of `pc` is `Cell*` and, consequently, it is `Cell::print()` that is *bound at compile-time*.

   In the third version of `ReCell` below, we will see how `virtual` functions can rectify the matter. `virtual` in the declaration of a method means that it is *capable of dynamic or run-time binding*. Java uses *dynamic binding* by default.

   In C++, the default is compile time binding (linking); note: *compile-time = static*. To get dynamic (run-time) binding, you must use the keyword `virtual` when declaring a method.
7. Inheritance is announced by `class ReCell: public Cell`. This is `public` inheritance; `private` inheritance *is* possible, but is so uncommon that we are never going to mention it again.
8. Notice that, in `ReCell`, we have been able to refrain from referencing the `private` members of `Cell`. Later, we will see that `protected`, which allows a reduced privacy, can be used if derived classes must access `private` members of base classes.

8.2.2 Protected

Program `CellR2.cpp` is the same as `CellR1.cpp` with the simple modification that `Cell`'s `val_` field is now `protected` instead of `private`. `protected` gives limited external visibility — to inheriting classes only. Hence, `protected` is *half-way* privacy: visible to classes related by inheritance, but otherwise hidden.

Thus, `ReCell` functions are at liberty to reference `val_`.

Whether `protected` is good or bad is a moot point. Some people take the view that it dilutes the main point of object-orientation — *encapsulation and information hiding*.

As we have seen, in `Cell, ReCell` it is possible to avoid. In more complex classes, efficiency and other concerns may make it unavoidable.

```
//----- CellR2.cpp -----------------------------------
// from CellR1 -- using protected
// j.g.c. 12/2/98, 8/1/99, 2003/12/02
//----------------------------------------------------
#include <iostream>
using namespace std;

class Cell{
public:
  Cell() : val_(0) {}
  void set(int val){val_= val;}
  int get() const {return val_;}
  void print() const {cout<<"value= " << val_<< endl;}
protected:
  int val_; // using protected, derived classes have access
};

class ReCell: public Cell{
public:
  ReCell() : Cell(), backup_(0) {}
  void set(int val){backup_= val_; val_= val;}
  void restore(){val_= backup_;}
  void print() const
  {cout<< "value= " << val_<< " backup= " <<backup_ << endl;}
private:
  int backup_;
};

int main(){
  Cell c;
  c.set(123);
  cout<< "Cell c: "<< endl; c.print();

  ReCell rc;
  rc.set(456);  rc.set(789);
  cout<< "ReCell rc: "<< endl; rc.print();

  return 0;
}
```

8.2.3 Virtual Functions and Dynamic Binding

Finally, in `CellR3.cpp` we get to the complete `ReCell`. Already, in the introduction, we have previewed the need to make `print()` a `virtual` function. In fact, for the purposes of demonstration, we have made two `print()` functions, one non-virtual as before, the other named `printV()` `virtual`.

```
//----- CellR3.cpp -------------------------------------
// inheritance etc. j.g.c. 2003/12/03 from CellR1.cpp
//------------------------------------------------------
#include <iostream> using namespace std;

class Cell{
public:
  Cell() : val_(0) {}
  virtual void set(int val){val_= val;}
  int get() const {return val_;}
  void print() const {cout<<"value= " << val_<< endl;}
  virtual void printV() const {cout<<"V: value= " << val_<< endl;}
private:
  int val_;
};
class ReCell: public Cell{
public:
  ReCell() : Cell(), backup_(0) {}
  virtual void set(int val){backup_= get(); Cell::set(val);}
  void restore(){set(backup_);}
  void print() const
  {cout<< "value= " << get()<< " backup= " <<backup_ << endl;}
  virtual void printV() const
  {cout<< "V: value= " << get()<< " backup= " <<backup_ << endl;}
private:
  int backup_;
};
int main(){
  Cell c;  c.set(123);
  cout<< "Cell c, c.print() "<< endl;  c.print();
  cout<< "Cell c, c.printV() "<< endl;  c.printV(); cout<< endl;

  Cell* pc= &c;
  cout<< "Cell* pc = &c, pc->print() "<< endl; pc->print();
  cout<< "Cell* pc = &c, pc->printV() "<< endl; pc->printV(); cout<< endl;

  ReCell rc;  rc.set(456);  rc.set(789);
  cout<< "ReCell rc:, rc.print() "<< endl; rc.print();
  cout<< "ReCell rc:, rc.printV() "<< endl; rc.printV(); cout<< endl;

  c= rc;  cout<< "c= rc, c.print() "<< endl; c.print();
  cout<< "c= rc, c.printV() "<< endl; c.printV(); cout<< endl;
  pc= &rc;  cout<< "pc = &rc, pc->print() "<< endl; pc->print();

  cout<< endl<< "...finally, we see the effect of *virtual* ..."<< endl;
  cout<< "pc = &rc, pc->printV() "<< endl; pc->printV();   cout<< endl;

  cout<< "Cell& cc= rc; cc.printV() ..."<< endl;
  Cell& cc= rc;   cc.printV();                return 0; }
```

Dissection

1. First, notice the use of the `virtual` qualifier in the declaration of `set` and `print`. Note again: **virtual** means **capable of dynamic binding**.

2. Now, let us jump to `main()`. In the following,

   ```
   c= rc;  cout<< "c= rc, c.print() "<< endl; c.print();
   cout<< "c= rc, c.printV() "<< endl; c.printV(); cout<< endl;
   ```

   Slicing again takes place and the following is output:

   ```
   c= rc, c.print()
   value= 789
   c= rc, c.printV()
   V: value= 789
   ```

   `c` is of type `Cell` and nothing can make it otherwise.

3. However, as already indicated in 8.2.1, `pc` takes on the *dynamic* type `pointer-to-ReCell`. Still, in the following,

   ```
   pc= &rc;  cout<< "pc = &rc, pc->print() "<< endl; pc->print();
   ```

   `Cell::print()` is *statically bound* (compile-time) to `pc`, whose *static* type is `pointer-to-Cell`, and the following is output:

   ```
   pc = &rc, pc->print()
   value= 789
   ```

4. But, in the following, we finally see the effect of `virtual`,

   ```
   cout<< endl<< "...finally, we see the effect of *virtual* ..."<< endl;
   cout<< "pc = &rc, pc->printV() "<< endl; pc->printV();
   ```

   The output is:

   ```
   ...finally, we see the effect of *virtual* ...
   pc = &rc, pc->printV()
   V: value= 789 backup= 456
   ```

   Here, the *dynamic* type of `pc` (`pointer-to-ReCell`) is respected and `ReCell::printV()` is called.

More on Dynamic Binding In function calls like `c.print()` etc., and even `c.printV()` conventional *static-binding* is used. I.e. the appropriate function is chosen & bound at compile-time — using the best choice of (overloaded) function the compiler can make: the one indicated by the static (declared) type of the calling object.

However, in `pc->printV()` quite different code is generated: extra *run-time* selection & dispatcher code is generated. At run-time, this code detects the *run-time* (or *dynamic*) type of the pointer, and calls (dispatches) the appropriate function.

In summary, four points are worth making:

- Dynamic binding is possible only for *pointers* or *references*;
- We must be dealing with objects related via inheritance;
- The static type of the pointer must be *pointer-to-base* or *reference-to-base*;
- The method must have been declared `virtual`.

Why does C++ not simply allow dynamic binding by default (like Java) and so get rid of any need for the `virtual` distinction? The answer is the desire to optimise for run-time *efficiency* by default. The *run-time selection & dispatcher code* mentioned above takes extra time and extra memory.

It used to be that games programmers frowned upon inheritance and virtual functions because their potential performance penalty. But that is no longer the case.

8.3 Overriding (virtual functions) versus Overloading

When we specialise a `virtual` function is provided, we call this *overriding*. On the other hand, when we specialise a non-virtual function, we call this *overloading*.

The difference between overriding and overloading is that:

- The binding (selection) of an *overridden* function is done at *run-time*;
- An *overloaded* function is bound at compile time.

8.4 Strong Typing

Many discussions on *strong typing* implicitly or explicitly equate strong typing to:

- Every variable is statically (at compile time) bound to a single type.
- Calls to operations on variables — including operations that combine two or more variables — can be statically checked for type compatibility.

To these we can add, for languages with dynamic typing:

- The type of value held in a dynamically typed variable can be determined at run time, i.e. dynamically, thus, it is possible to select *at run-time* an appropriate *polymorphic* function / operator.

Even where dynamic binding is employed, it is still possible to determine type compatibility at compile time. This is because dynamic binding is restricted to the *family* of classes related by inheritance, and, since *derived* / 'inheriting' classes are merely extensions of their *base(s)*, a *derived* class *is* type-compatible with a base class.

For the above situation to be semantically safe, we must ensure that the type system provided by the class hierarchy respects the *Liskov Substitution Principle*: if function `f()` is defined for a parameter of class B, and given a class D derived from B, then calls to `f()`, with an argument of class D, should work properly.

8.5 Inheritance & Virtual Functions — Summary

If we declare a function, say `VF()`, as `virtual` in a base class, say `B`. Then, in a derived class, say `D`,`VF()` can be *overridden*. Of course, `D::VF()` must have matching signature/prototype.

Now, a pointer to an object of class `D`, `pd`, can be assigned to a pointer of class `B`, `pb`, without explicit type conversion.

Then, when invoked with a pointer to the base class, the selection from the *overridden* virtual functions `VF()` is done *dynamically*, i.e. delayed until run-time, when the *dynamic* type of the pointer will be known.

Thus:

```
B b;
B* pb = &b;
D d;

pb = &d;
pb->VF(); //D's VF selected - dynamically
          //if VF is *not*  virtual - B's VF would have been
          //selected --- i.e. statically bound, at compile-time.
```

In the absence of a derived class VF(), of course, the base class virtual function will be used.

8.6 Inheritance versus Inclusion, is-a versus has-a

Without care it is easy for beginners to abuse inheritance through confusion of the class relationships: **is-a**, versus **has-a**. Inheritance is a mechanism by which a base class may be extended to yield a derived class; *inclusion* (composition) is the normal manner in which we can build a class using objects of other classes (composition, inclusion).

Since inheritance provides *extension*, it is crucial to understand that the relationship between a derived class and a base class is *is-a*: a derived class object *is-a* base class object.

In the `Person` class hierarchy, described below, in which `Student` and `Lecturer` are derived from (inherit from) `Person`, it is clear that an `Student` object certainly *does* qualify for *is-a* `Person`.

But please note that the *is-a* relationship is not *transitive*; a `Person` need not necessarily be a `Student` or `Lecturer`. Careful design of classes will mean that they will obey normal everyday semantics of relationships between categories, e.g. Persons, and sub-categories, e.g. Students.

On the other hand, the relationship between class `Person` and `Address` is *has-a*: a `Person` object *has-a* `Name` and *has-a*n `Address`. Given our strict interpretation of *has-a* and *is-a*, on no account could we say that a `Person` *is-a*n `Address`.

8.7 Inheritance & Virtual Functions and Ease of Software Extension

Suppose you sell, or give, precompiled versions of the `Person` class hierarchy, to someone else, together with a `main()` calling program — without giving them the source code. You envisage them extending the program by building other classes based on your hierarchy.

Without virtual functions and overriding, you would have to, in the `main()` program, predict all the different operations that would ever be required (impossible) and provide them in a giant `stitch` statement.

With inheritance and virtual functions, anyone can extend the class hierarchy, so long a they override appropriate virtual functions. Moreover, the `main()` program can call code that is developed long after it.

This leads to the aphorisms mentioned earlier:

Old code can call new code — which contrasts with the non-object-oriented solution of new code being written to call old code provided in libraries.

Upside-down libraries — i.e. the calling *framework* remains fixed, while the called functions may be modified. Virtual functions are C++'s object-oriented way of avoiding *callback* functions — such as those use in OpenGL.

And, we recall the *open-closed principle*: software modules (here classes) must be *open* to extension, but must be *closed* to modification.

8.8 Example 2: a Person class hierarchy

It is always difficult to explain the software design process in notes. Normally, the notes contain the finished article without any discussion of the design and implementation steps.

We started off requiring classes to represent a *course* (now module) in the institute. Our first attempt was the `Student` class below.

First, we thought the a single string for Address was inadequate — so we designed an `Address` class. Then it became clear that we needed a `Lecturer` and possibly other people classes. Consequently, we designed a `Person` class into which we could factor all the data and behaviour common to `Lecturer` and `Student`, and, presumably, other people classes that may arise.

```
class Student{
public:
  Student() : fn_(""), ln_(""), id_(""), ad_(""){}
  Student(string fn, string ln, string id, string ad, int m) :
        fn_(fn), ln_(ln), id_(id), ad_(ad){}
  Student(string);
  string first(){ return fn_; };
  string last(){ return ln_; };
  string ident(){ return id_;};
  string address(){ return ad_;};
  string toString() const;
private:
  string fn_;
  string ln_;
  string id_;
  string ad_;
};
```

File `Person.h` now shows a `Person` class. It has been deliberately kept simple, and thus, other than as a demonstration of inheritance, it is rather limited and unremarkable.

```
// ---------------- Person.h --------------------------
// j.g.c version 1.0; 2007-08-27
// ----------------------------------------------------
#ifndef PERSON_H
#define PERSON_H

#include <string>
#include "StringTokenizer.h"
#include "Address.h"

class Person{
private:
  std::string firstName_;
  std::string lastName_;
  Address address_;

public:
  Person(std::string& firstName, std::string& lastName,
         Address& address);
  // see Address.h for comment on = "..."
  Person(std::string str = "blank,blank;blank,blank,blank,blank");
  virtual std::string toString() const;
};
#endif


// ---------------- Person.cpp --------------------------
// j.g.c version 1.0; 2007-08-27
// ------------------------------------------------------
#include "Person.h"
#include <iostream>

Person::Person(std::string& firstName, std::string& lastName,
               Address& address)
          : firstName_(firstName), lastName_(lastName),
            address_(address){}

Person::Person(std::string str){
  // first split into name ; address
  StringTokenizer st1 = StringTokenizer(str, ";");
  std::string sName = st1.token(0);
  std::string sAddress = st1.token(1);

  address_ = Address(sAddress);
  // now split name
  StringTokenizer st2 = StringTokenizer(sName, ",");
```

```
    firstName_ = st2.token(0);  lastName_ = st2.token(1);
  }

std::string Person::toString() const{
    return firstName_ + "," +  lastName_ + ";" + address_.toString();
}
```

Address

```
// ---------------- Address.h ---------------------------
// j.g.c version 1.0; 2007-08-26
// ------------------------------------------------------
#ifndef ADDRESS_H
#define ADDRESS_H

#include <string>
#include "StringTokenizer.h"

class Address{
private:
  std::string street_; // street = first line of address
  std::string city_;
  std::string region_; // county or state
  std::string country_;

public:
  Address(std::string& street, std::string& city,
          std::string& region, std::string& country);
  // = "" does for default constructor
  // which C++ wants but will never be used
  Address(std::string str = "blank,blank,blank,blank");
  std::string toString() const;
};
#endif
```

Dissection of Person.h

1. There are two constructors; the first is the standard initialising constructor; the second constructs from a string. The latter is nice if we want to read from a file.

2. We avoid needing a default constructor (`Person()` with the default argument in

   ```
   Person(std::string str = "blank,blank;blank,blank,blank,blank");
   ```

   If we ever have `Person()`, the `Person(std::string)` constructor will be called with those default arguments. This is fine; in fact we never deliberately use a default constructor but C++ implicitly inserts one in certain situations and the compiler will complain if it doesn't exist.

3. Person *is composed of* two `strings` and an `Address`.

   Why this design? We could have avoided `Address` altogether and put the `Address` string straight into `Person`. But that causes two problems:

   (a) `Person` starts to get big and complicated; as it is we can design and test `Address` and after that we can more or less forget about the complexities of the `Address` part;

   (b) What happens if we ever need to design a class, for example Company, that has an an Address but yet does not inherit from Person? The answer is that we would have to duplicate the address part of Person — bad, bad, bad. If you ever find yourself duplicating code, go off and have a cup of tea and rethink your design.

4. We copy Java in giving classes `toString` methods; this makes life easy when we want to write an object to the screen or to a file. Think of this as the output equivalent of the `Person(std::string)` constructor that is used for input from keyboard and file.

5. We also copy Java in that we developed a `StringTokenizer` class for ripping apart strings of data separated by delimiters.

6. In the declaration of `toString` we have used the keyword `virtual`; this is highly significant since it releases the possibility of dynamic binding of `toString`;

We refrain from describing the implementation of `Person`; we can discuss that in class.

8.9 A Student Class

File `Student.h` shows an `Student` class which is *derived* from `Person` — it inherits from `Person`.

```
// ---------------- Student.h ---------------------------
// j.g.c version 1.0; 2007-08-27
// ------------------------------------------------------
#ifndef STUDENT_H
#define STUDENT_H

#include <string>
#include "Person.h"

class Student: public Person{
private:
  std::string id_;

public:
  Student(std::string& firstName, std::string& lastName,
          Address& address, std::string& id);
  // see Address.h for comment on = "..."
  Student(std::string str = "blank,blank;blank,blank,blank,blank;blank");
  virtual std::string toString() const;
};
#endif
```

```
// ---------------- Student.cpp --------------------------
// j.g.c version 1.0; 2007-08-27
// -------------------------------------------------------
#include "Student.h"
#include <iostream>

Student::Student(std::string& firstName, std::string& lastName,
                 Address& address, std::string& id)
          : Person(firstName, lastName, address), id_(id){}

Student::Student(std::string str) : Person(str){
  // first split into name ; address ; id
  // note that Person(str) will ignore id
  StringTokenizer st1 = StringTokenizer(str, ";");
  id_ =  st1.token(2); // (0) is name, (1) is address
}

std::string Student::toString() const{
  return Person::toString() + ";" + id_;
}
```

Dissection of Student

1. `class Student : public Person{...` signifies that `Student` inherits from `Person`.
2. `Student` exhibits the same behaviour as `Person`; well, the only common behaviour is the `toString` method!
3. As well as `id_` `Student`, via inheritance, contains all the data members of `Person`;
4. As a consequence of `public` inheritance, only the `public` parts of `Person` are visible within `Student`.

8.10 Use of the Person Hierarchy

First of all we show a simple test program for `Person` alone; then we show a test that uses al three classes in the hierarchy (`Person, Student, Lecturer` and demonstrates *polymorphism*.

File `PersonT1.cpp` shows a program which exercises `Person`. You will have access to similar programs for `Student` and `Lecturer`; the latter two are extremely similar to `PersonT1.cpp` and not worth printing here.

8.10.1 PersonT1.cpp

`PersonT1.cpp` demonstrates the use of `std::vector`, `std::list`, and `std::iterator` and the *random number generator*. These may be worth some discussion in class.

A further example of `std::vector` is given in section 8.11 at the end of this Chapter.

```
// ----------------- PersonT1.cpp ------------------
// j.g.c. 2007-08-27
// -------------------------------------------------
#include "Person.h"
#include <vector>
#include <list>
#include <iostream>
#include <iterator>
#include <cstdlib> // for rand

using namespace std;

class PersonT1 {
public:
  PersonT1(){

    vector<string> firstNames = vector<string>();
    firstNames.push_back(string("Seamus"));
    firstNames.push_back(string("Sean"));
    firstNames.push_back(string("Sharon"));
    firstNames.push_back(string("Sinead"));
    firstNames.push_back(string("Sarah"));
    firstNames.push_back(string("Simon"));

    vector<string> lastNames = vector<string>();
    lastNames.push_back(string("Bloggs"));
    lastNames.push_back(string("Burke"));
    lastNames.push_back(string("Blaney"));
    lastNames.push_back(string("Bradley"));
    lastNames.push_back(string("Brown"));
    lastNames.push_back(string("Black"));

    vector<string> streets = vector<string>();
    streets.push_back(string("Port Road"));
    streets.push_back(string("Pearse Road"));
    streets.push_back(string("College Terrace"));
    streets.push_back(string("Kingsbury"));
    streets.push_back(string("New Street"));

    vector<string> cities = vector<string>();
    cities.push_back(string("Newtown"));
    cities.push_back(string("Oldtown"));
    cities.push_back(string("Johnstown"));
    cities.push_back(string("Hightown"));
    cities.push_back(string("Lowtown"));

    vector<string> regions = vector<string>();
    regions.push_back(string("Oldshire"));
    regions.push_back(string("Newshire"));
```

```
    regions.push_back(string("Green County"));
    regions.push_back(string("Old County"));

    vector<string> countries = vector<string>();
    countries.push_back(string("Ireland"));
    countries.push_back(string("Wales"));
    countries.push_back(string("England"));
    countries.push_back(string("Scotland"));

    list<Person> personList = list<Person>();
    string street, city, region, country, lastName, firstName;
    unsigned int rnseed= 139;
    srand(rnseed); // initialise random number generator.
    int n = 10, j;
    for(int i = 0; i< n; ++i){
      cout<< "i = "<< i<< endl;
      j = rand()%streets.size();
      street = streets.at(j);
      j = rand()%cities.size();
      city = cities.at(j);
      j = rand()%regions.size();
      region = regions.at(j);
      j = rand()%countries.size();
      country = countries.at(j);

      j = rand()%firstNames.size();
      firstName = firstNames.at(j);
      j = rand()%lastNames.size();
      lastName = lastNames.at(j);

      Address a = Address(street, city, region, country);
      cout<< a.toString()<< endl;
      Person p = Person(firstName, lastName, a);
      cout<< p.toString()<< endl;
      personList.push_back(p);
    }
    string s = string("James,Mulligan;Lough Salt,Kilmacrennan,Donegal,Ireland");
    Person p(s);
    personList.push_back(p);
    p = Person();
    personList.push_back(p);
    cout<<"The Person list size is: "<< personList.size()<< endl;
    cout<< "The Person list is:"<< endl;
    list<Person>::iterator itr;
    for(itr = personList.begin(); itr!= personList.end(); ++itr){
      cout<< itr->toString()<< endl;
    }
  }
};
```

```
//main constructs the test

int main(){
  PersonT1();
}
```

8.10.2 PersonT2.cpp

Now we test the full Person hierarchy, and demonstrate *polymorphism*.

```
// ----------------- PersonT2.cpp ------------------
// j.g.c. 2007-08-27
// demonstrating polymorphism in the Person hierarchy
// -------------------------------------------------
#include "Person.h"
#include "Lecturer.h"
#include "Student.h"

#include <vector>
#include <list>
#include <iostream>
#include <iterator>
#include <cstdlib> // for rand
#include <sstream>

using namespace std;

class PersonT2 {
public:
  PersonT2(){

    // etc., see PersonT1.cpp

    vector<string> offices = vector<string>();
    offices.push_back(string("2220"));
    offices.push_back(string("3301"));
    offices.push_back(string("3302"));
    offices.push_back(string("1401"));

    list<Person*> personList = list<Person*>();
    string street, city, region, country, lastName, firstName;
    unsigned int rnseed= 1111353;
    srand(rnseed); // initialise random number generator.
    // and spin it a few times
    for(int i = 0; i< 1000; ++i)rand();
    int n = 10, j;
    string id1("L200702");
    for(int i = 0; i< n; ++i){
```

```
      j = rand()%streets.size();
      street = streets.at(j);
      j = rand()%cities.size();
      city = cities.at(j);
      j = rand()%regions.size();
      region = regions.at(j);
      j = rand()%countries.size();
      country = countries.at(j);

      j = rand()%firstNames.size();
      firstName = firstNames.at(j);
      j = rand()%lastNames.size();
      lastName = lastNames.at(j);

      Address a = Address(street, city, region, country);
      //cout<< a.toString()<< endl;
      int type = rand()%4; //randomly choose type of Person
      if(type < 2){
        Person *p = new Person(firstName, lastName, a);
        personList.push_back(p);
      }
      else if(type == 2){
        int k = i + 10;
        ostringstream ossid;
        ossid<< id1<< k;
        string id = ossid.str();
        Student *st = new Student(firstName, lastName, a, id);
        personList.push_back(st);
      }
      else {
        j = rand()%offices.size();
        string office = offices.at(j);
        Lecturer *l = new Lecturer(firstName, lastName, a, office);
        personList.push_back(l);
      }
    }
    string s = string("James,Mulligan;Lough Salt,Kilmacrennan,Donegal,Ireland");
    Person p2(s);
    personList.push_back(&p2);
    Person p3 = Person();
    personList.push_back(&p3);
    cout<<"The Person list size is: "<< personList.size()<< endl;
    cout<< "The Person list is:"<< endl;
    list<Person*>::iterator itr;
    for(itr = personList.begin(); itr!= personList.end(); ++itr){
      cout<< (*itr)->toString()<< endl;
    }
  }
};
```

```
// main constructs the test
int main(){
  PersonT2();
}
```

Dissection of PersonT2.cpp

1. In order to use *polymorphism*, we now use a `list<Person*>` — a list of *pointer to the base class*.

2. This means that `(*itr)->toString()` will act *polymorphically* and bind the appropriate `toString` (Person, Student, or Lecturer) according to the *dynamic* type of what `itr` has extracted from the list; for this to happen, `toString` must be `virtual`, see below.

3. If we had used `list<Person>` (plus obvious changes), all objects inserted into the list would have been *sliced* to produce a Person object and the additional parts in Student and Lecturer would have been discarded.

4. If we had used `list<Person*>` as above, but had left the `virtual` qualifier off `toString`, then *compile time* (static) binding would have been used to and in `(*itr)->toString()` `Person::toString()` would have been linked and all the objects would have been displayed as if they were (plain) `Person`.

   That is, *slicing* would not have taken place; the extended `id_` and `office_` are there alright, but `Person::toString()` knows nothing about them.

   Make sure you understand the difference between slicing and static binding in the context above.

8.11 Standard Library Vector

The program in `VectorTest1.cpp` demonstrates the usefulness of the standard library `vector` class and the associated standard library *algorithms*.

```
//------ VectorTest1.cpp --------------------------
// tests for std::vector
// j.g.c. 2003/02/20
//-------------------------------------------------
#include <iostream> // for cout<< etc.
#include <string>
#include <fstream>  // for files - ofstream, ifstream
#include <algorithm>
#include <vector>
#include <cstdlib> // for rand
#include "Marks.h"
using namespace std;
```

```
int main(){
  const int n= 10; unsigned int rnseed= 139;
  double d[n];
  srand(rnseed); // initialise random number generator.
  //cout<< "Max. random number= "<< RAND_MAX<< endl;
  for(int i= 0; i< n; i++){
    d[i]= (double)rand()/(double)RAND_MAX; // numbers in 0.0 to 1.0
  }
  cout<< "\nArray of random numbers\n"<< endl;
  for(int i= 0; i< n; i++){
    cout<< d[i]<< " ";
  }
  cout<< endl;

  vector<double> v1;
  for(int i= 0; i< n; i++){
    v1.push_back(d[i]);
  }

  cout<< "\nIterating through vector, one way \n"<< endl;
  vector<double>::iterator iter;
  for(iter = v1.begin(); iter!= v1.end(); iter++){
    cout<< *iter<< " ";
  }
  cout<< endl;

  cout<< "\nIterating through vector, another way \n"<< endl;
  for(int i = 0; i< n; i++){
    cout<< v1[i]<< " ";
  }
  cout<< endl;

  sort(v1.begin(), v1.end());
  cout<< "\nSorted vector\n"<< endl;
  for(int i = 0; i< n; i++){
    cout<< v1[i]<< " ";
  }
  cout<< endl;

  reverse(v1.begin(), v1.end());

  cout<< "\nReverse sorted vector\n"<< endl;
  for(int i = 0; i< n; i++){
    cout<< v1[i]<< " ";
  }
  cout<< endl;

  cout<< "\nFront, back elements of vector, and size()\n"<< endl;
  cout<< v1.front()<< " "<< v1.back()<< " "<< v1.size();
```

```
cout<< endl;

cout<< "\nYou can modify the vector (x 10.0)\n"<< endl;
for(int i = 0; i< n; i++){
  v1[i] = v1[i]*10.0;
}
for(unsigned int i = 0; i< v1.size(); i++){
  cout<< v1[i]<< " ";
}
cout<< endl;

cout<< "\nYou can have vectors of strings\n"<< endl;
vector <string> v2;
v2.push_back("The");  v2.push_back("quick");
v2.push_back("brown");  v2.push_back("fox");
v2.push_back("jumped"); v2.push_back("over");
v2.push_back("the");  v2.push_back("lazy");
v2.push_back("dog's"); v2.push_back("back");

for(unsigned int i = 0; i< v2.size(); i++){
  cout<< v2[i]<< " ";
}
cout<< endl;

sort(v2.begin(), v2.end());
cout<< "\nSorted\n"<< endl;
for(unsigned int i = 0; i< v2.size(); i++){
  cout<< v2[i]<< " ";
}
cout<< endl;

cout<< "\nYou can assign vectors \n"<< endl;
vector<string> v12 = v2;
for(unsigned int i = 0; i< v12.size(); i++){
  cout<< v2[i]<< " ";
}
cout<< endl;

cout<< "\nYou can remove elements from vectors \n"<< endl;
remove(v12.begin(), v12.end(), "lazy");
for(unsigned int i = 0; i< v12.size(); i++){
  cout<< v12[i]<< " ";
}
cout<< endl;

cout<< "\nYou can search vectors \n"<< endl;
vector<string>::iterator it;
it= find(v12.begin(), v12.end(), "fox");
cout<<*it<< endl;
```

```
cout<< "\n... and then erase that element \n"<< endl;
v12.erase(it);
for(unsigned int i = 0; i< v12.size(); i++){
  cout<< v12[i]<< " ";
}
cout<< endl;

vector<string> v22;
cout<< "\nYou can copy using copy, but use resize first \n"<< endl;
v22.resize(4);
copy(v12.begin(), v12.begin()+3, v22.begin() );
for(unsigned int i = 0; i< v22.size(); i++){
  cout<< v22[i]<< " ";
}
cout<< endl;

vector< vector<string> > vv;
cout<< "\nYou can have vectors of vectors, but use > > in decl. \n"<< endl;
vv.push_back(v2);
vv.push_back(v12);
vv.push_back(v22);
for(unsigned int i = 0; i< vv.size(); i++){
  cout<< "vector "<< i<<": ";
  for(unsigned int j = 0; j< vv[i].size(); j++){
    cout<< vv[i][j]<< " ";
  }
  cout<< endl;
}
cout<< endl;

cout<< "\nYou can have vectors of your own objects\n"<< endl;
vector <Marks> v3;
int ex, ca;
for(unsigned int i = 0; i< 8; i++){
  ex= rand()%100; ca= rand()%100;
  Marks m= Marks(ex, ca);
  v3.push_back(m);
}

for(unsigned int i = 0; i< v3.size(); i++){
  cout<< v3[i]<< " ";
}
cout<< endl;

sort(v3.begin(), v3.end());
cout<< "\nSorted\n"<< endl;
for(unsigned int i = 0; i< v3.size(); i++){
  cout<< v3[i]<< " ";
```

```
  }
  cout<< endl;

  cout<< "\nYou can then save that vector to a file\n"<< endl;
  ofstream fileout("xyz.dat");
  if(!fileout){
    cerr<< "cannot open file <xyz.dat>\n";
    return -1;
  }
  for(unsigned int i=0; i< v3.size(); i++){
    fileout<< v3[i];
    fileout<< endl;
  }
  fileout.close();

  cout<< "\n... and read some of them back\n"<< endl;
  vector<Marks> v4;

  ifstream filein("xyz.dat");
  if(!filein){
    cerr<< "cannot open file <xyz.dat>\n";
    return -1;
  }
  Marks tmp;
  for(unsigned int i=0; i< 5; i++){
    filein>> tmp;
    v4.push_back(tmp);
  }
  filein.close();

  for(unsigned int i = 0; i< v4.size(); i++){
    cout<< v4[i]<< " ";
  }
  cout<< endl;

  return 0;
}
```

8.12 Marks

Notice that if you ever want to apply the `sort` *algorithm* to a collection of objects, then the class must have a < (ordering) operator:

```
bool operator<(const Marks& lhs, const Marks& rhs){
  return lhs.overall() < rhs.overall();
}
```

```
//---------------------------------------------------------
// Marks.h - StudentInfo project
// j.g.c. 2003/02/20, 2007-01-05
//---------------------------------------------------------
#ifndef MARKSH
#define MARKSH

#include <string>
#include <iostream>
#include <sstream>

class Marks{
  // friends are not members, but have access to private data
  friend std::ostream& operator<<(std::ostream& os, const Marks& m);
  friend std::istream& operator>>(std::istream& is, Marks& m);
  friend bool operator<(const Marks& lhs, const Marks& rhs);

public:
  Marks() : ex_(0.0), ca_(0.0){} //defining in header is same as 'inline'
  Marks(double ex, double ca);
  Marks(std::istream& is);
  double overall() const;
  std::string toString() const;
  static const int examWeight = 70;

private:
  double ex_;
  double ca_;
};
#endif
```

```
//---------------------------------------------------------
// Marks.cpp - StudentInfo project
// j.g.c. 2003/02/20, 2007-01-12
//---------------------------------------------------------
#include "Marks.h"
using namespace std;
```

```
Marks::Marks(double ex, double ca){
  ex_ = ex;
  ca_ = ca;
}

Marks::Marks(istream& is){
  // assumes that fields are separated by whitespace
  string tmp;
  is>> tmp; // throw away decoration ("Marks:")

  is>> ex_;
  is>> ca_;
}

double Marks::overall() const{
  double exw= double(examWeight)/100.0;
  return exw*ex_ + (1.0 - exw)*ca_;
}

ostream& operator<< (ostream& os, const Marks& m){
  // ideally, this would not output any decoration; in that
  // case, >> could be the same as <<
  os<< "Marks:"<< " "<< m.ex_<< " "<< m.ca_;
  return os;
}

string Marks::toString() const{
  stringstream ss;
  ss<< "Marks:"<< " "<< ex_<< " "<< ca_;
  return ss.str();;
}


istream& operator>>(istream& is, Marks& m){
  // assumes that fields are separated by whitespace
  string tmp;
  is>> tmp; // throw away decoration ("Marks:")

  is>> m.ex_;
  is>> m.ca_;

  return is;
}

bool operator<(const Marks& lhs, const Marks& rhs){
  return lhs.overall() < rhs.overall();
}
```

Chapter 9

Classes that manage memory

9.1 Introduction

The classes that we have used so far — `Cell, ReCell`, `Person` and the graphics `Vector` classes have avoided any use of *heap* memory. In each case, member variables are *stack* memory based and memory is managed automatically, i.e. object creation and deletion is done automatically by the language.

On the other hand, many practical classes/objects require, for one reason or another, to employ heap memory; the sort of objects that require the flexibility of heap storage are *containers*: lists, stacks, queues, trees, and dynamically sizable arrays and strings.

Fortunately, the C++ *standard library* now makes available an extensive range of containers, so that the development of heap based classes is no longer as necessary as it used to be. That is fortunate, because, as we repeat below, development of classes that need to manage memory is difficult and error prone.

Make sure you recall heap and stack variables. In summary: *normally* declared/defined variables are constructed (created) on the **stack**; they are constructed when control reaches the point of declaration and they are destroyed when control reaches the end of the scope unit in which they were declared. Thus, programmers need only declare them and the language takes care of the rest — the *lifetime* of the variables is handled *automatic*ally. Variables that are constructed using `new` reside on the **heap**; as well as requiring to be explicitly constructed (using `new`), they must be destroyed explicitly (using `delete`).

The important distinction between heap-based objects and stack-based objects is the same for objects as for variables. That is, the compiler will automatically and implicitly generate memory management and similar functions for the simpler stack objects, but for *heap* based objects — those objects whose state is stored in heap memory and accessed through pointers, the class developer must explicitly program memory management, see section 5.11, i.e. constructors and destructors, and associated functions such as *copy*.

Memory management is never easy. However, we can develop patterns — or templates — (taking the general meanings of these terms) that show how to provide the memory management facilities generally needed for heap-based classes. We have briefly covered heap memory management in section 5.11; maybe you should review that now.

We will discuss in some detail some critical aspects of heap-based classes:

Destructors A *destructor* is called, automatically, when control reaches the end of the block in which the object was declared, i.e. when the object goes out of scope. The compiler will always provide adequate destruction of stack-based objects, but, for heap-based objects, proper destructor memory management must be provided if `garbage` and memory-leaks (or worse, *dangling pointers*) are to be avoided.

Copy constructor A *copy constructor* is called when the object is passed (*by value*) to and from functions. Again, for classes which use stack memory, the compiler will always provide an adequate copy constructor. For heap-based objects the case is quite analogous to that of the destructor: proper constructor memory management must be provided.

Assignment *Assignment operator* (the '=' operator) needs treatment similar to the copy constructor.

The Big-Three The C++ FAQs (Cline et al. 1999b) uses the term *the Big-Three* for these three functions: copy constructor, assignment operator, and destructor.

This is because, for classes that use heap storage, it is almost certainly necessary to explicitly program all three. If they are not programmed, the compiler will provide default versions which will probably not do what you would wish; moreover the inadequacies of these defaults may be most subtle, and may require quite determined and skilled testing to detect.

In the remainder of this chapter we are going to develop heap-based dynamic array class (rather like `std::vector`).

We will spend some time on this, but a much better example is given in in Chapter 12. The details of a linked list class are in Chapter 13.

There is some repetition between the remainder of this chapter and Chapter 12.

9.2 A Heap-based Dynamic Array Class

9.2.1 Class Declaration

Dissection of Array1.h

Copy constructor A copy constructor always has the signature `T(const Type& source);`, where `Type` is the class name.

```
Array(const Array& source);
```

Destructor There is only *ever* one destructor, and it must have the name of the class preceded by '∼' (tilde).

```
~Array();
```

```
//----- Array1.h ------------------------------------
// j.g.c. 6/1/99, 8/1/99, 2007-01-06
// heap memory, and full complement of
// copy constructor, destructor, assignment operator.
//---------------------------------------------------
#ifndef ARRAYH
#define ARRAYH
#include <iostream>
#include <cassert>

class Array{
public:
  Array(unsigned int len= 0); // defaults to zero if no arguments
  Array(unsigned int len, int val);
  Array(const Array& source);           // copy constructor
  ~Array();                             // destructor
  Array& operator=(const Array& source);// assignment operator
  void set(int val, unsigned int i);
  int get(unsigned int i) const;
  unsigned int length() const;
  void print() const;
private:
  unsigned int len_;
  int* dat_;
  void copy(const Array& source);       // used only internally
};
#endif
```

Figure 9.1: Array1.h class declaration.

Destructors *cannot* be invoked explicitly — they are implicitly invoked when an object must be destroyed, i.e. when exiting the block or function in which it was defined, i.e. when execution leaves the portion of the program in which it is *in scope*.

Assignment operator `Array& operator=(const Array& source);`

Private function — copy Note that `copy` is made `private` because it is used only internally — it is not part of the class interface.

```
void copy(const Array& source);       // used only internally
```

9.2.2 Class implementation code

The implementation code for the `Array` class is shown in Figures 9.2 and 9.3.

Dissection of Array1.cpp

1. We have added print statements to some of the significant functions; this is to enable tracing of calls to them. As we shall see in the next subsection, you may be surprised what is going on during, for example a call to a function which takes an object as a parameter.

   However, in the interests of clarity, these prints statements have been removed in the following discussions.

2. Default constructor.

```
Array::Array(unsigned int len) : len_(len) {
  if(len_==0){dat_ = 0; return;}
  dat_ = new int[len_];
  assert(dat_!= 0);
  for(unsigned int i=0; i< len_; i++)set(0, i);
}
```

   (a) If length is zero, no need to allocate, set `dat_` pointer to null and return.

   (b) Otherwise allocate an array of `len_` ints on heap. `new` returns a pointer to this data block, and this is assigned to `dat_`.

   (c) Use `assert` to ensure that the allocation was successful; `new` returns a null pointer if it is unsuccessful, e.g. due to resources of free memory having become exhausted.

   (d) Then initialise the memory to 0.

3. Initialising constructor.

```
Array::Array(unsigned int len, int val) : len_(len) {
  if(len_==0){dat_ = 0; return;}
  dat_ = new int[len_];
  assert(dat_!= 0);
  for(unsigned int i=0; i< len_; i++)set(val, i);
}
```

```
//----- Array1.cpp ----------------------------------
// j.g.c. 6/1/99, 8/1/99, 2007-01-06
//---------------------------------------------------
#include "Array1.h"

using namespace std;

  Array::Array(unsigned int len) : len_(len) {
    cout<< "*1.Array(unsigned int)*"<< endl;
    if(len_==0){dat_ = 0; return;}
    dat_ = new int[len_];
    assert(dat_!= 0);
    for(unsigned int i=0; i< len_; i++)set(0, i);
  }

  Array::Array(unsigned int len, int val) : len_(len) {
    cout<< "*2.Array(unsigned int, int)*"<< endl;
    if(len_==0){dat_ = 0; return;}
    dat_ = new int[len_];
    assert(dat_!= 0);
    for(unsigned int i=0; i< len_; i++)set(val, i);
  }

  Array::Array(const Array& source){
    cout<< "3.*Array(const Array&) -- copy constructor*"<< endl;
    copy(source);
  }

  Array::~Array(){
   cout<< "*~Array()*"<< endl;
   delete [] dat_;
  }

  Array& Array::operator=(const Array& source){
    cout<< "*operator=*"<< endl;
    if(this!= &source){ // beware a= a;
      delete [] dat_;
      copy(source);
    }
    return *this;
  }

  void Array::copy(const Array& source){
    len_= source.length();
    if(len_== 0){dat_ = 0; return;}
    dat_ = new int[len_];
    assert(dat_!= 0);
    for(unsigned int i= 0; i< len_; i++)dat_[i] = source.get(i);
  }
... continued ...
```


Figure 9.2: Array1.cpp class implementation, part 1

```
... Array1.cpp ...
  void Array::set(int val, unsigned int i){
    assert(i<len_);
    dat_[i]= val;
  }

  int Array::get(unsigned int i) const {
    assert(i<len_);
    return dat_[i];
  }

  unsigned int Array::length() const {
    return len_;
  }

  void Array::print() const {
    unsigned int len= length();
    cout<<"length= " << len<< ": ";
    for(unsigned int i=0; i<len; i++){
       cout<< get(i)<< " ";
    }
    cout<< endl;
  }
```

Figure 9.3: Array1.cpp class implementation, part 2.

This is the same as the default constructor — except for the initialising value.

4. Function copy.

```
void Array::copy(const Array& source){
  len_= source.length();
  if(len_== 0){dat_ = 0; return;}
  dat_ = new int[len_];
  assert(dat_!= 0);
  for(unsigned int i= 0; i< len_; i++)dat_[i] = source.get(i);
}
```

   (a) This is very similar to the default constructor — except that this time we have passed another object to be copied.

   (b) Notice that we have passed a reference (`Array& source`). If we don't we'll end up constructing many multiple copies, each of which must also be destroyed. And when the object becomes large, as may be the case for an `Array` the performance drain can be considerable.

   (c) Of course, we guarantee the safety of the referenced object (in the caller) by making the reference `const`.

5. Copy constructor.

```
Array::Array(const Array& source){
  copy(source);
}
```

   Here, all the work is done by `copy`. Again notice the use of reference and `const`.

6. Destructor.

```
Array::~Array(){
 delete [] dat_;
}
```

   (a) Here we use `delete` to release the allocated memory.

   (b) Owing to the fact that `dat_` points to an array, we must use `delete []`.

7. Assignment operator '='.

```
Array& Array::operator=(const Array& source){
  if(this!= &source){ // beware a= a;
    delete [] dat_;
    copy(source);
  }
  return *this;
}
```

   (a) Let us say we have two `Array` objects, `x, y` and `x = y`. Then this assignment is *exactly* equivalent to

```
x.operator=(y);
```

(b) Again notice the use of a reference and `const`.

(c) Since we can envisage `x = x`, however improbable, we have to be careful to check whether this is the case — and if it is, do nothing.

(d) **this** is an implicit pointer variable which points at the object itself. Thus `if(this!= &source)` checks if the calling object and the source object share the same memory location — and so are the same object!

(e) `return *this;` returns the object (actually a reference, see next comment).

(f) Why does `operator=` return `Array&`? (a) In C and C++ is is standard for an assignment to have the value of the object assigned, e.g.

```
int x = y = 10;
```

we want the same for objects.

And, as usual, we want the efficiency of reference passing.

The next chapter has a more extensive discussion of operator overloading.

9.2.3 Assertions

C++ provides a macro function, `assert`, which evaluates a Boolean expression and if it evaluates to *false* generates a run-time error message and halts the program. `assert` is defined in `cassert`.

The following shows an example use of `assert` to check the legality of the array index operator; `len_` is the *length* of the array.

```
void Array::set(int val, unsigned int i){
  assert(i<len_);
  dat_[i]= val;
}
```

When the following code is executed:

```
c2.set(200, 10); //demonstrates assert
```

the array index 10 will *not* satisfy the condition `<len_ (len- == 3)` and an error will be reported and the program halted:

```
Array1T1: Array1.cpp:52: void Array::set(int, unsigned int): Assertion
<len_' failed.
```

For performance or other reasons, the assertion can be disabled either by including the compiler switch `-DNDEBUG`; I'm not sure how Visual Studio handles this. Alternatively insert the following pre-processor directive in the source file:

```
#define NDEBUG
```

Note also the assertion which tests whether memory allocation has been successful:

```
dat_ = new int[len_];
assert(dat_!= 0);
```

Use of *assertions* like this is an example of *defensive programming* and can be very useful in testing and can greatly simplify debugging.

9.2.4 A simple client program

The program in Figure 9.4 demonstrates the use of the `Array` class.

Dissection The most interesting activities are those involving the constructors, the assignment operator and the destructor. To examine these, it is useful to examine the output generated by `Array1T1.cpp`.

1. This code

   ```
   Array c1;
   cout<< "Array c1: "; c1.print();
   ```

   outputs:

   ```
   *1.Array(unsigned int)*
   Array c1: length= 0:
   ```

   i.e. the default constructor is called, and, when printed, the array is seen to have length 0.

2. This code

   ```
   Array c2(3);
   cout<< "Array c2(3): "; c2.print();
   ```

   outputs:

   ```
   *1.Array(unsigned int)*
   Array c2(3): length= 3: 0 0 0
   ```

   i.e. the constructor is called with argument 3, and, when printed, the array is seen to have length 3 and initial values 0.

```
//----- Array1T1.cpp ----------------------------------
// j.g.c. 6/1/99, 8/1/99, 13/1/99, 2007-01-06
//-----------------------------------------------------
#include "Array1.h"
using namespace std;
Array fred(Array a){
  cout<< "Array a in fred: "; a.print(); cout<< endl;
  Array b(a);
  int len= b.length();  int x;
  for(int i= 0; i< len; i++){
    x= b.get(i);
    b.set(x+100, i);
  }
  return b;  }

int main(){
  cout<< "heap based array ..."<< endl;
  Array c1; cout<< "Array c1: "; c1.print();

  Array c2(3); cout<< "Array c2(3): "; c2.print();
  //c2.set(200, 10); demonstrates assert

  Array c3(5, 7); cout<< "Array c3(5, 7): "; c3.print(); cout<< endl;

  cout<< "c3.length() "<< c3.length()<< endl;

  int len= c3.length();
  for(int i= 0; i< len; i++){
    c3.set(i+20, i);
  }
  cout<< "c3 updated: "; c3.print(); cout<< endl<< endl;

  cout<< "c3.get: ";
  for(int i= 0; i< len; i++){
    cout<< c3.get(i)<< " ";
  }
  cout<< endl<< endl;

  Array c4= c3;
  cout<< "Array c4= c3: "; c4.print(); cout<< endl;

  Array c5(c3); cout<< "Array c5(c3): "; c5.print(); cout<< endl;

  c2= c3;
  cout<< "c2= c3: "; c2.print(); cout<< endl;

  c2= fred(c3);
  cout<< "c2= fred(c3): "; c2.print(); cout<< endl;
  cout<< "finishing ..."<< endl;
  return 0;}
```


Figure 9.4: Array1T1.cpp

3.
```
Array c3(5, 7);
cout<< "Array c3(5, 7): "; c3.print(); cout<< endl;
```
outputs:

```
2.Array(unsigned int, int)*
Array c3(5, 7): length= 5: 7 7 7 7 7
```

i.e. the initialising constructor is called with arguments 5, 7, and, when printed, the array is seen to have length 5 and initial values 7.

4.
```
Array c4= c3;
cout<< "Array c4= c3: "; c4.print(); cout<< endl;
```
outputs:

```
3.*Array(const Array&) -- copy constructor*
Array c4= c3: length= 5: 20 21 22 23 24
```

i.e. the copy constructor is called with argument c3; this may be a little surprising — in this circumstance, the compiler always knows to optimise out an unnecessary assignment. When printed, the array c4 is seen to have been appropriately copied from c3.

5.
```
Array c5(c3);
cout<< "Array 5(c3): "; c5.print(); cout<< endl;
```
outputs:

```
3.*Array(const Array&) -- copy constructor*
Array c5(c3): length= 5: 20 21 22 23 24
```

i.e. the copy constructor is called with argument c3; in this case this is more expected.

6. Now we get to an true assignment statement.

```
c2= c3;
cout<< "Array c2= c3: "; c2.print(); cout<< endl;
```

outputs:

```
*operator=*
Array c2= c3: length= 5: 20 21 22 23 24
```

i.e. the assignment operator is called with argument c3.

7. Now we get to a function call with parameter passing; this generates a lot more activity than you would expect.

```
c2= fred(c3);
```

First, a copy of c3 is made to create local object a, and the print statement executed.

```
3.*Array(const Array&) -- copy constructor*
Array a in fred: length= 5: 20 21 22 23 24
```

Next local `Array b` is created from `a`. (Actually, this is a little silly, since `a` is already a local copy; nevertheless, the code as given is a good general demonstration of the activity that surrounds function calls and parameter passing).

```
3.*Array(const Array&) -- copy constructor*
```

Then, after `b` is modified (no output), `return b` causes the copy constructor to make a copy to an anonymous temporary object in the main program:

```
3.*Array(const Array&) -- copy constructor*
```

Then both local objects, `a, b` are destroyed:

```
*~Array()*
*~Array()*
```

Then the temporary is assigned to `c2`:

```
*operator=*
```

and the temporary is destroyed:

```
*~Array()*
```

Then `c2` is printed,

```
c2= fred(c3): length= 5: 120 121 122 123 124
```

8. Finally, we get to the end of main, but it's not all over yet:

```
finishing ...
*~Array()*
*~Array()*
*~Array()*
*~Array()*
*~Array()*
```

Here we see objects `c1, c2 ..., c5` being destroyed.

9.3 The 'this' pointer

If we have to reference the object *itself* in a member function, we can do it through an implicit pointer `this`; e.g. below in `operator=`:

```
Array& Array::operator=(const Array& source){
  if(this!= &source){ // beware a= a;
    delete [] dat_;
    copy(source);
  }
  return *this;
}
```

9.4 Copy Constructors and Parameter Passing and Value Return

Usually, copy constructors must be provided for classes which use *heap* memory. As already described, if the developer of the class does *not* provide one, the compiler will do so.

9.4.1 Naïve member-wise constructor, shallow copy

The problem is that the compiler provided *default copy constructor* will merely perform a naïve *member-wise* or *shallow copy* constructor. In the case of `Array` this means that only members `len` and `dat_` will be copied.

The deficiency of the default copy constructor is seen in the following example:

```
Array a(5,7), b(a);
Array c(a); //OK so far, but b and a share the same
            //data array -- referenced by pdat_
b.set(125, 2); // 2nd element of b becomes 125
int x= a.get(2); // what value of x? 7?
                 // *no* it is 125!
                // because a.pdat_ and b.pdat_ are aliases for one
                                          //another
```

A worse problem is caused if `a` gets destroyed before `b`, or vice versa; in this case `b.dat_` will become a *dangling reference*.

9.4.2 Proper 'deep' copy constructor

In the case of `Array`, we provide

```
Array::Array(const Array& source){
  cout<< "3.*Array(const Array&) -- copy constructor*"<< endl;
  copy(source);
}
void Array::copy(const Array& source){
  len_= source.length();
  if(len_== 0){dat_ = 0; return;}
  dat_ = new int[len_];
  assert(dat_!= 0);
  for(unsigned int i= 0; i< len_; i++)dat_[i] = source.get(i);
}
```

This provides a so-called *deep copy*, i.e. copy not just the explicit members, but also what they point to.

9.4.3 Copy constructor and parameter passing and return

Already, we have given a detailed account of constructor activity during the following function call:

```
call:       c2= fred(c3);

Array fred(Array a){
  Array b(a);
  // ...
  return b;
}
```

1. Local object `a` is created — using the copy constructor.
2. Then object `b` is created — again the copy constructor.
3. On return, `b` must be copied to the caller; this is to a an anonymous temporary object, which, momentarily, replaces `fred(c3)`.
4. Next, the assignment operator = is invoked to copy the temporary object to `c2`.

I suppose the main point to me made here is that a faulty copy constructor is a great liability — but, because of the implicitness of most of its use, it can cause problems that are very hard to trace.

9.5 Assignment

Assignment is subject to the same requirements as *copy constructors*, and the discussion of section 9.4; i.e. a non-trivial assignment operator must usually be provided for classes which use *heap* memory. If such is not provided, again the compiler will provide a default one, which uses *shallow copy*.

Copy constructors and assignment operators perform very similar tasks, and so their code often closely mirrors that of the other. In the following example, the only difference is (a) the check against mistaken deletion of an objects memory if it is assigned to itself, and (b) the use of `delete [] dat_;` to free whatever memory the object is currently using.

```
Array& Array::operator=(const Array& source){
  cout<< "*operator=*"<< endl;
  if(this!= &source){ // beware a= a;
    delete [] dat_;
    copy(source);
  }
  return *this;
}
```

A more detailed account of `operators` is given in the next chapter.

As mentioned earlier, you should note carefully that `operator=` returns a reference (`Array&`). This allows constructs of the form `a = b = c;`, which, if it is to be valid for built-in types like `int`, had better be valid for objects. Indeed, many *container* classes, e.g. those in the C++ standard library, depend on and demand this property.

9.6 Destructor

As repeatedly mentioned, destructors are called, automatically, when program control reaches the end of the block in which the object was defined (and hence constructed).

The destructor for an object which references heap memory is subject to the same memory management demands as *copy constructor* and *assignment operator*. The compiler will always provide adequate destruction of stack-based objects, but, for heap-based objects, proper destructor memory management must be provided if `garbage` and memory-leaks (or worse, *dangling pointers*) are to be avoided.

Thus, the `Array` destructor:

```
Array::~Array(){
 delete [] dat_;
}
```

9.7 The Big-Three

The C++ FAQs (Cline et al. 1999b) uses the term *the Big-Three* for the previous three functions: copy constructor, assignment operator, and destructor.

This is because, for any class for which you must provide one, it is almost certainly necessary to provide all three.

9.8 Reference parameters and reference return

I hope that the discussion in subsection 9.4.3 will have given the (correct) impression that function `fred` may be generating an awful lot of unnecessary work.

In many cases, a better solution is the following:

```
call:       c2= fred1(c3); // call remains the same

Array fred1(const Array& a){
  Array b(a);
  // do things to b
  return b;
}
```

Here we have dispensed with one call to a copy constructor — `a`. But the others remain — creation of `b` and creation of the temporary in the caller. Actually, as mentioned above, we could equally well use:

```
Array fred11(Array a){
  // a is a local *copy*
  // do things to a
  return a;
}
```

Reference return It would now be tempting to continue the trend and return a reference to `b` (or `a` — if `fred11`).

In this circumstance, we cannot easily do that. Since `b`, or `a` are local variables, they are destroyed upon return. Thus,

```
call:       c2= fred21(c3);

Array& fred2err(Array a){
  // a is a local *copy*
```

```
  // do things to a
  return a;
}
```

is **very seriously flawed**; upon return, a reference (remember — a reference is just an *alias* to `a` is returned (to a temporary); next, `a` is destroyed, so that when, in the caller, the assignment has a dangling reference on its right-hand-side!.

This is subtle. Just because an object references heap-memory, it itself is not a heap object unless it is created using `new`.

Returning a reference to a *local object* is a cardinal sin.

Pointer return Obviously, if `b` is to become a revised version of the argument (`c3`), we cannot avoid a constructor for `b`. However, what about the temporary? This can be accomplished by the code in `fred2`:

```
//Warning: ugly code follows
Array* pc2;

call:      pc2= fred1(c3); // now returns a pointer

Array* fred2(const Array& a){
  Array* pb = new Array(a);
  int len= pb->length();
  int x;
  for(int i= 0; i< len; i++){
    x= pb->get(i);
    pb->set(x+100, i);
  }
  return *pb;
}
```

The price to pay is that `b`, or rather `pb`, must reference a heap object; to create this, we must use `new`.

The real disadvantage of `fred2` is that is creates a heap object that someone else must take responsibility for destroying — i.e. potential garbage. However, you may be relieved to hear that this example is quite artificial, and that properly designed software will normally use `new` only within a class, and that a corresponding `delete` will be provided (by the class designer).

Return value optimisation Actually, the following version, which is by far the prettiest, may turn out to be the most efficient — due to *return value optimisation*.

```
Array fred11(Array a){
  // a is a local *copy*
  // do things to a
  return a;
}
```

When the compiler notices a local `Array a` being created, followed by a `return a`, it knows that considerable optimisation can achieved by creating `a` in the callers space.

9.9 Privacy is class-based, not object-based

Up to now, we have been careful to use, where possible, *accessor* functions such as `get` in member functions, e.g. in `copy`:

```
void Array::copy(const Array& source){
  len_= source.length();
  if(len_== 0){dat_ = 0; return;}
  dat_ = new int[len_];
  assert(dat_!= 0);
  for(unsigned int i= 0; i< len_; i++)dat_[i] = source.get(i);
}
```

```
for(unsigned int i= 0; i< len_; i++)dat_[i] = source.get(i);
```

Here we use `get` for access to `source`, but we go straight to `dat_` for the object itself.

However, we could easily rewrite this line as:

```
for(unsigned int i= 0; i< len_; i++)dat_[i] = source.dat_[i];
```

Thus, `source`'s `private` members are visible to the member functions of other objects of the same class.

Privacy is class-based, not object-based.

On the other hand, maybe the use of accessor functions is no harm at all; it further limits the effects of change of representation to even member functions. However, use of `source.get(i)` may offend the performance ultra-fastidious since it may incur some small performance penalty.

If we wanted to take use of interface functions to the limit, we could have:

```
for(unsigned int i= 0; i< len_; i++)set(source.get(i), i);
```

Here, `set` refers to the object itself — the so called *implicit argument*.

Chapter 10

Operator Overloading

10.1 Introduction

Already, in the previous chapter, we have glanced at the provision of `operator=` for the `Array` class. We have also noted that these so-called *overloaded* operators have exactly the same meaning as an equivalent traditionally named function.

Here, we go into a little more detail, and, at the end of the chapter, we discuss some of the consequences of the asymmetry cause by the difference between *implicit* and *explicit* arguments, e.g. implicit: `object1` versus explicit: `object2` in `object1.member_function(object2)`.

It should be said that `operator overloading` is not favoured by everyone. As we have stated, operators calls are no different to function calls. Neither, as we show below is there any magic involved, nor any performance advantages. Java does not allow operator overloading; *in my opinion* that was a wise design decision. **However**, whether you like operator overloading or not, there are at least three good reasons to know about it: (i) for some classes, you need to be able to create an assignment operator (`operator=`) — if, like the `Array` classes in the previous chapter and in this chapter, the classes are heap based, then you have no choice, you must provide an assignment operator; (ii) many programs that you will use or modify will employ operator overloading; (iii) if you want use standard library containers and you want to use `sort` and other *algorithms* on objects of a class, you will need to supply the *comparison operator*, `operator<`.

In addition, some argue that it is beneficial to be able to use + when we want to add, e.g. `Arrays`, just as adding `ints` or `doubles`; the claim is that this makes the language more expressive.

10.2 Lead-in — Add Functions for Array

As a lead-in to our discussion on operator overloading, we first provide `add` functions for `Array`; subsequently, we will replace these with overloaded operators `+, +=`. We present `Array2.h, Array2.cpp` and `ArrayT2T1.cpp` almost without comment — except for the *non-member* function `add` there should be nothing new. See Figures 10.1, 10.2, 10.3 and 10.4.

```
//----- Array2.h -------------------------------------
// j.g.c. 8/1/99, 2007-01-07
// copied from Array1.h (ch09);
// getting ready for operators.
//----------------------------------------------------
#ifndef ARRAYH
#define ARRAYH

#include <iostream>
#include <cassert>

class Array{
public:
  Array(unsigned int len= 0);
  Array(unsigned int len, int val);
  Array(const Array& source);
  ~Array();
  Array& operator=(const Array& source);
  void addTo(const Array& other);
  Array& addTo1(const Array& other);
  void copy(const Array& source);
  void set(int val, unsigned int i);
  int get(unsigned int i) const;
  int length() const;
  void print() const;
private:
  unsigned int len_;
  int* dat_;
};
Array add(const Array& a1, const Array& a2);
#endif
```

Figure 10.1: Array2.h

```
//----- Array2.cpp ----------------------------------
// j.g.c. 8/1/99, 2007-01-07
// getting ready for operators
//---------------------------------------------------
#include "Array2.h"
using namespace std;

Array::Array(unsigned int len) : len_(len) {
  if(len_== 0){ dat_ = 0; return;}
  dat_ = new int[len];
  assert(dat_!= 0);
  for(unsigned int i=0; i< len_; i++)set(0, i);
}

Array::Array(unsigned int len, int val) : len_(len) {
  if(len== 0){ dat_ = 0; return;}
  dat_ = new int[len];
  assert(dat_!= 0);
  for(unsigned int i=0; i< len_; i++)set(val, i);
}

Array::Array(const Array& source){
  copy(source);
}

Array::~Array(){
  delete [] dat_;
}

Array& Array::operator=(const Array& source){
  if(this!= &source){ // beware a= a;
    delete [] dat_;
    copy(source);
  }
  return *this;
}

void Array::copy(const Array& source){
  len_= source.length();
  if(len_< 1){dat_ = 0; return;}
  dat_ = new int[len_];
  assert(dat_!= 0);
  for(unsigned int i= 0; i< len_; i++)dat_[i] = source.dat_[i];
}  continued ...
```

Figure 10.2: Array2.cpp, part 1

```
... Array2.cpp ...
void Array::addTo(const Array& other){
  assert(len_== other.len_);
  for(unsigned int i= 0; i< len_; i++)dat_[i] += other.dat_[i];
}

Array& Array::addTo1(const Array& other){
  assert(len_== other.len_);
  for(unsigned int i= 0; i< len_; i++)dat_[i] += other.dat_[i];
  return *this;
}

void Array::set(int val, unsigned int i){
  assert(i<len_);
  dat_[i]= val;
}

int Array::get(unsigned int i) const {
  assert(i<len_);
  return dat_[i];
}

int Array::length() const {
  return len_;
}

void Array::print() const {
  unsigned int len= length();
  cout<<"length= " << len<< ": ";
  for(unsigned int i=0; i<len; i++){
     cout<< get(i)<< " ";
  }
  cout<< endl;
}

// notice no Array:: add is *not* a member
Array add(const Array& a1, const Array& a2){
  Array res(a1);
  return res.addTo1(a2);
}
```

Figure 10.3: Array2.cpp, part 2

```
//----- Array2T1.cpp ----------------------------------
// j.g.c. 8/1/99, 2007-01-07
//-----------------------------------------------------
#include "Array2.h"
using namespace std;

int main()
{
  Array c2(5, 7);
  cout<< "Array c2(5, 7): "; c2.print();

  Array c3(5, 10);
  cout<< "Array c3(5): "; c3.print(); cout<< endl;

  Array c4(5);
  cout<< "Array c4(5): "; c4.print();

  c3.addTo(c2);
  cout<< "c3.addTo(c2): "; c3.print(); cout<< endl<< endl;

  cout<< "c2: "; c2.print();
  cout<< "c3: "; c3.print(); cout<< endl;

  c4= add(c2, c3);
  cout<< "c4= add(c2, c3): "; c4.print(); cout<< endl;

  Array c5(5), c6(5);
  cout<< "Array c5(5): "; c5.print();
  cout<< "Array c6(5): "; c6.print();
  c6= c5.addTo1(c4);
  cout<< "c6= c5.addTo1(c4);"<< endl;
  cout<< "c5: "; c5.print();
  cout<< "c6: "; c6.print();

  cout<< endl;
  return 0;
}
```

Figure 10.4: Array2T1.cpp

Dissection of Array2.h, .cpp

1. Method `AddTo` performs addition of the argument object, to the calling object:

   ```
   array1.addTo(array2);
   ```

2. Method `AddTo1` does the same as `addTo`, only it returns a value; thus, it can be used in the C/C++ style:

   ```
   result = array1.addTo1(array2);
   ```

   `addTo1` will become a model for `operator+=`.

3. Notice that `add` is declared outside the class.

   ```
   Array add(const Array& a1, const Array& a2);
   ```

   This is because we want to call this function with its input arguments symmetrically:

   ```
   result = add(array1, array2);
   ```

   If it was a member function, one of the arguments would have to be implicit (the calling object), i.e. it would be declared exactly as like `addTo1` and the call would be asymmetric:

   ```
   result = array1.add(array2);
   ```

4. Later, we will see that it is normal to define a two argument function `add` (and `operator+`) in terms of `AddTo1` (or `operator-=-`).

10.3 Chaining calls to member functions

As noted, `AddTo1` returns a value; not only can it be used as follows:

```
result = array1.addTo1(array2);
```

but also in chained set of calls:

```
result = array10.addTo1(array1.addTo1(array2));
```

We will see a similar principle in action when we require `operator=` to return a reference to its object; this allow the familiar C/C++ chaining of assignments such as:

```
object1 = object2 = object3;
```

10.4 Operators

In this section, we show how to replace the `add` functions of the previous section with overloaded operators:

- `addTo` replaced by `operator+=`
- `add` replaced by `operator+`; as with `add`, `operator+` is declared as a non member.
- In addition, we provide an overloaded `operator<<` so that we can output `Array` objects using `cout<<`; this operator also must be declared as a non-member.

  `operator<<` is really handy to have; as you will notice, I include it in many of the classes I write; it makes use of the classes very easy, especially testing.
- Finally, we include an indexing `operator[]` which replaces `get` and `set`.

See Figures 10.5, 10.6, 10.7 and 10.8.

```
//----- Array3.h ------------------------------------
// j.g.c. 14/1/99, 2007-01-07
// copied from Array2.h
// introducing operators
//---------------------------------------------------
#ifndef ARRAYH
#define ARRAYH

#include <iostream>
#include <cassert>

class Array{
public:
  Array(unsigned int len= 0);
  Array(unsigned int len, int val);
  Array(const Array& source);
  ~Array();
  Array& operator=(const Array& source);
  Array& operator+=(const Array& other);
  void copy(const Array& source);
  int& operator[](unsigned int i) const;
  unsigned int length() const;
private:
  unsigned int len_;
  int* dat_;
};
Array operator+(const Array& a1, const Array& a2);
std::ostream& operator<<(std::ostream& os, Array& a);

#endif
```

Figure 10.5: Array3.h

```
//----- Array3.cpp ----------------------------------
// j.g.c. 14/1/99, 2007-01-07
//---------------------------------------------------
#include "Array3.h"
using namespace std;

Array::Array(unsigned int len) : len_(len) {
  if(len_==0){dat_ = 0; return;}
  dat_ = new int[len];
  assert(dat_!= 0);
  for(unsigned int i=0; i< len_; i++)dat_[i]= 0;
}

Array::Array(unsigned int len, int val) : len_(len) {
  if(len==0){dat_ = 0; return;}
  dat_ = new int[len];
  assert(dat_!= 0);
  for(unsigned int i=0; i< len_; i++)dat_[i]= val;
}

Array::Array(const Array& source){
  copy(source);
}

Array::~Array(){
  delete [] dat_;
}

Array& Array::operator=(const Array& source){
  if(this!= &source){ // beware a= a;
    delete [] dat_;
    copy(source);
  }
  return *this;
}

void Array::copy(const Array& source){
  len_= source.length();
  if(len_==0){dat_ = 0; return;}
  dat_ = new int[len_];
  assert(dat_!= 0);
  for(unsigned int i= 0; i< len_; i++)dat_[i] = source.dat_[i];
}
... continued ...
```

Figure 10.6: Array3.cpp, part 1

```
... Array3.cpp ...

Array& Array::operator+=(const Array& other){
  assert(len_==other.len_);
  for(unsigned int i= 0; i< len_; i++)dat_[i] += other.dat_[i];
  return *this;
}

int& Array::operator[](unsigned int i) const{
  assert(i<len_);
  return dat_[i];
}

unsigned int Array::length() const {
  return len_;
}

// notice no Array:: -- add is *not* a member
Array operator+(const Array& a1, const Array& a2){
  Array res= a1;
  return res+= a2;
}

// and operator<< is not a member
ostream& operator<<(ostream& os, Array& a){
  unsigned int len= a.length();
  os<< "[" << len << "]{ ";
  for(unsigned int i=0; i< len; i++){
    os<< a[i];
    if(i!= len-1)os<<", ";
  }
  os<<"}"<< endl;
  return os;
}
```

Figure 10.7: Array3.cpp, part 2

```
//----- Array3T1.cpp ----------------------------------
// j.g.c. 9/1/99, 2007-01-07
//-----------------------------------------------------
#include "Array3.h"
using namespace std;
int main(){
  Array c2(5, 7);  cout<< "Array c2(5, 7): "<< c2;

  Array c3(5, 10);  cout<< "Array c3(5): "<< c3<< endl;

  Array c4(5);  cout<< "Array c4(5): "<< c4;

  c3+= c2;    //previously c3.addTo(c2);
  cout<< "c3+= c2: "<< c3<< endl<< endl;

  cout<< "c2: "<< c2;   cout<< "c3: "<< c3<< endl;

  c4 = c2 + c3;// previously c4= add(c2, c3);
  cout<< "c4 = c2 + c3: "<< c4<< endl;

  Array c5(5), c6(5);
  cout<< "Array c5(5): "<< c5;
  cout<< "Array c6(5): "<< c6;
  c6= c5+= c4;// previously c6= c5.addTo1(c4);
  cout<< "c6= c5+=c4;"<< endl;
  cout<< "c5: "<< c5;
  cout<< "Array c6: "<< c6<< endl;

  int len= c6.length();
  for(int i= 0; i< len; i++){ c6[i]= i+50; }
  cout<< "Using 'cout<< c6[i]'"<< endl;
  for(int i= 0; i< len; i++){ cout<< c6[i] << " "; }
  cout<< endl<< endl;

  cout<<"Now operators called in 'function' manner"<< endl<< endl;
  c3.operator+=(c2);    //c3+= c2;
  cout<< "c3.operator+=(c2);: "<< c3<< endl<< endl;

  cout<< "c2: "<< c2; cout<< "c3: "<< c3<< endl;

  c4= operator+(c2, c3);  // c4 = c2 + c3;
  cout<< "c4= operator+(c2, c3);: "<< c4<< endl;

  for(int i= 0; i< len; i++){ c6.operator[](i)= i+50; }
  cout<< "Using 'cout<< c6.operator[](i)'"<< endl;
  for(int i= 0; i< len; i++){ cout<< c6.operator[](i) << " "; }
  cout<< endl<< endl;
  return 0; }
```

Figure 10.8: Array3T1.cpp

Dissection of Array3

1. Here we see `operator+=` compared to `addTo`

   ```
   c3+= c2;    //previously c3.addTo(c2);
   ```

2. Here we see that we can call `operator+=` using normal function-call syntax:

   ```
   c3.operator+=(c2);    //c3+= c2;
   ```

3. Here we see `operator+` compared to `add`:

   ```
   c4 = c2 + c3;// previously c4= add(c2, c3);
   ```

4. Here we show `operator+` can also be called as a normal function:

   ```
   c4= operator+(c2, c3);  // c4 = c2 + c3;
   ```

5. Here we show use of `operator[]` and how it replaces *both* `put` and `get`:

   ```
   for(int i= 0; i< len; i++){
     c6[i]= i+50;  // replaces c6.put(1+50, i);
   }

   for(int i= 0; i< len; i++){
     cout<< c6[i] << " "; // replaces c6.get(i);
   }
   ```

6. And here we show it called as a function:

   ```
   cout<< c6.operator[](i) << " ";
   ```

7. Use of `operator[]` in `put` (write) mode is a bit subtle. The operator returns `int&` — a reference to an `int`.

   In verb+get+ (read) mode, that's understandable enough — when the reference is returned to a temporary (anonymous) variable, as discussed before in the context of returning references, the = operator immediately copies the value to the destination: `destination= source1 + source2;`, and the temporary is destroyed.

   In the case of `put` (write), the reference is returned to a temporary (anonymous) variable, the value on the right-hand-side of the assignment is assigned to the variable that the reference *aliases*, i.e. the element of the array. Subtle!

The stream output operator << needs special consideration.

Non-member operator, argument order We must declare `operator<<` outside class `Array`:

```
ostream& operator<<(ostream& os,const Array& a);
```

The operator << takes two arguments: an object of the class `ostream` (usually `cout` — which is defined in `iostream`, hence `#include <iostream>`), and a `Array` object. This will be invoked as:

```
Array x;
cout<< x;
```

hence, because of the convention for *infix* operators in C++, the first argument must be `ostream` (actually a reference), and the second a `Array`.

Friend functions and operators If non-member functions (or operators) need to access verb+private+ data (not the case here, but will be the case in later chapters), we have a problem — do we have to make the data public? No, we can declare the function/operator as `friend` in the class, e.g. let us say that `operator<<` needed to access the `private` members of `Array`, the following declaration, in the class, would give it the appropriate access:

```
friend ostream& operator<<(ostream& os,const Array& a);
```

10.5 Member versus Non-member Functions, Conversions

10.5.1 Introduction

The objective of this section is to discuss circumstances in which it is better to make functions or operators non-member (even if we need to make them `friend`, and to discuss the related matters of implicit arguments (the object itself) and conversion / coercion. We use a `String` class as an example for some of these phenomena.

I have changed this section very little since 1997; I include it here just to point out some quite subtle aspects of member versus non-member functions and conversions. You will not be examined on any of this, but I would feel uncomfortable if I hadn't pointed out some of these pitfalls.

10.5.2 A String Class

Although the standard library provides a perfectly good `string` class, we need the following home-grown one to furnish examples for the later discussion.

Class Interface, Methods

The interface for the `String` class is shown in `String.h`.

```
// ---- String.h -----------------------------------
// j.g.c. 1992 -- 2007-08-28
//-------------------------------------------------
#ifndef STRINGH
#define STRINGH
#include <iostream>
#include <cstring> // for C-String operations
#include <cctype>

class String {
public:
  String(const int len = 0);
  String(const char* cstr);
  String(const String& other);

  void assign(const String& rhs);
  String& operator=(const String& other);

  ~String();

  String& operator+=(const String& other);
  void insertCharAt(char c, int i);
  char charAt(int i) const;
  int length() const;
```

```
  int compare(const String& other) const;
  void print() const;
  void gets(); // reads up to \n, includes whitespaces
  void get(); // uses cin.get(c), reads up to any whitespace
  String concatM(const String& source2);
  String operator / (const String& source2);
private:
  char* p_; // string data - C form, with '\0'.
  int len_; // length, incl. '\0'
};

String concatNM(const String& s1, const String& s2);
String operator + (const String& s1, const String& s2);
#endif
```

Brief Dissection of String.h

1. There are three constructors, and, as it must be, they all have the same name — the class name. Again, note that each is distinguishable from the other by its *signature* — the types in parameter list. It is instructive to inspect each constructor in turn:

   - `String(const int len = 0);` This constructor is called when we define a `String` object as: `String s(22);`. It doubles as a *default* constructor, since `String s;` is taken as `String s(0);`
   - `String(const char* cstr);` This constructor is called when we define a `String` object as: `String s("abcdef");`, i.e. it constructs a `String` object from a 'C-string'.
   - Copy constructor `String(const String& source);`.

2. Concatenate. `String& operator+=(const String& other);` is simply an *operator* presentation of *concatenate*. Thus, `String s1("abc"); String s2("xyz"); s1 += s2` yields `s1 == "abcxyz"`.

Class implementation code

The implementation code for the `String` class is shown in `String.cpp`.

```
// ---- String.cpp ----------------------------------
// j.g.c. 1992 -- 2007-08-28
//--------------------------------------------------
#include "String.h"
#include <cstring>

String::String(const int len)
//Note: *don't* specify default len=0 here
{
   len_=len;
```

```
   p_=new char[len_+1];
   p_[0]='\0';
}

String::String(const char* cstr){
   len_=strlen(cstr);  //length of C string, excl. '\0'
   p_=new char[len_+1]; // include '\0'
   strcpy(p_, cstr);
}

// copy constructor
String::String(const String& other){
   len_ = other.length();
   p_=new char [len_+1];
   strcpy(p_, other.p_);
}

//only for pedagogical reasons - see '=' for comments
void String::assign(const String& source){
  if(this != &source) {   // beware s = s;
    delete [] p_;
    len_= source.length();
    p_= new char[len_+1];
    strcpy(p_, source.p_);
  }
}

String& String::operator=(const String& other){
  if(this != &other) { // beware s=s - Stroustrup p.238
    delete [] p_; //old text buffer
    len_ = other.length();
    p_ = new char[len_+1];
    strcpy(p_, other.p_);
  }
  return *this;
}

String::~String()
{
  delete [] p_;
  p_=0;
}

String& String::operator+= (const String& other){
  int oldl = length();
  len_=oldl + other.length(); //NB character count
  char* pp= new char[len_ + 1];
  strcpy(pp, p_);
  delete [] p_;
```

```
  strcpy(&(pp[oldl]), other.p_);
  p_ = pp;
  return *this;
}

String concatNM(const String& s1, const String& s2){
  String s = s1;
  s += s2;
  return s;
}

String operator + (const String& source1, const String& source2){
  String s = source1;
  s += source2;
  return s;
}

String String::concatM(const String& source2){
  String s = *this;
  s += source2;
  return s;
}

String String::operator / (const String& source2){
  String s = *this;
  s += source2;
  return s;
}


void String::insertCharAt(char c, int i){
  if(i >= length())return;
  p_[i]=c;
}

char String::charAt(int i) const{
  if(i >= length())return '\0';
  else return p_[i];
}

int String::length() const{
   return len_;
}

int String::compare(const String & other) const{
  char *s = p_;
  char *t = other.p_;
  return strcmp(s,t);
}
```

```
void String::print() const {
   std::cout<< p_;
}

//reads up to \n
void String::gets(){
  char c,s[81];
  int i = 0;
  while(std::cin.get(c)){
    if(c=='\n')break;
    s[i] = c;
    i++;
  }
  s[i]='\0';
  *this = String(s);
}

// uses std::cin.get(c), reads up to any whitespace
void String::get(){
  char c,s[81];
  int i = 0;
  while(std::cin.get(c)){
    if(isspace(c))break;
    s[i] = c;
    i++;
  }
  s[i]='\0';
  *this = String(s);
}
```

A simple test program

The program in `StringT1.cpp` demonstrates the use of the parts of the `String` class.

```
// ------------ StringT1.cpp -------------------------------
// j.g.c. 1997 -- 2007-08-26
// ---------------------------------------------------------
#include "String.h"
using namespace std;

int main(){
  String s1(0), s4;
  String s2("abcdef");
  String s3(s2);

  cout<< "gets(): enter a string -- end with Enter:";
```

```
  s1.gets();
  cout<< "s1: length = "<< s1.length()<< endl; s1.print();
  cout<< endl;

  cout<< "get(): enter a string -- end with _any_ white-space:";
  s4.get();
  cout<< "s4: length = "<< s4.length()<< endl; s4.print();
  cout<< endl;

  cout<< "s2: length = "<< s2.length()<< endl; s2.print();
  cout<< endl;
  cout<< "s3: length = "<< s3.length()<< endl; s3.print();
  cout<< endl;
  s2 += s1;
  cout<< "s2 += s1;"<< endl;
  cout<< "s2: length = "<< s2.length()<< endl; s2.print();
  cout<< endl;

  s4 = concatNM(s1, s2);
  cout<< "s4 = concatNM(s1, s2);"<< endl;
  cout<< "s4: length = "<< s4.length()<< endl; s4.print();
  cout<< endl;

  s4 = s1 + s2;
  cout<< "s4 = s1 + s2;"<< endl;
  cout<< "s4: length = "<< s4.length()<< endl; s4.print();
  cout<< endl;

  s4 = s1.concatM(s2);
  cout<< "s4 = s1.concatM(s2);"<< endl;
  cout<< "s4: length = "<< s4.length()<< endl; s4.print();
  cout<< endl;

  s4 = s1 / s2;
  cout<< "s4 =  s1 / s2;"<< endl;
  cout<< "s4: length = "<< s4.length()<< endl; s4.print();
  cout<< endl;

  return 0;
}
```

10.5.3 Member versus Non-member functions

Assume that we want to write a function that concatenates two `String`s to produce a third, and leaving the two sources intact. There are at least four ways we can do it:

1. Member function.

2. Non-member function — possibly `friend`.

3. Member operator, say `+`.

4. Non-member operator, say `/`; we name it `/` simply to avoid name conflict with `+`..

Whether a *function* is a member or not has an obvious consequence on the way it is called: if it is a member we must use the `object.member();` notation. For operators, the consequences are more subtle, but can be important.

Member function

The member function version, `concatM`, is defined as follows:

```
String String::concatM(const String& source2){
  String s = *this;
  s += source2;
  return s;
}
```

Notice that it uses operator `+=`; further, notice that it returns a `String` *by value*, it is tempting to return by reference (`String&`), but this would lead to a dangling reference, since `String s` is destroyed on return from `concatM`.

We notice too that the first argument is implicit — it is the object itself, as can be seen from the call:

```
String s1, s2, s4;
s4 = s1.concatM(s2);
```

The lack of symmetry in the call is not appealing, we would expect `s1` and `s2` to be treated uniformly. This can be corrected by developing `concat` as a non-member function.

Non-member function

The non-member function version, `concatNM`, is defined as follows:

```
String concatNM(const String& source1, const String&source2) {
  String s = source1;
  s += source2;
  return s;
}
```

The first argument is no longer implicit, and the call is symmetric:

```
String s1, s2, s4;
s4 = concatNM(s1, s2);
```

If `concatNM` had required to access private member data, which is actually not the case, then `concatNM` would have had to be been given `friend` privileges by including the following in the declaration of `String, string.h`:

```
friend String concatNM(const String& source1, const String& source2);
```

Non-member operator

The non-member operator version, +, is defined as follows:

```
String operator + (const String& source1, const String& source2)
{
  String s = source1;
  s += source2;
  return s;
}
```

The call is now both symmetric and infix:

```
String s1, s2, s4;
s4 = s1 + s2;
```

Again we note that if + had required to access private member data, then we would have had to include the following in the declaration of `String, string.h`:

```
friend String operator +(const String& source1, const String& source2);
```

Member operator

For completeness, we give a member operator version, /, which is defined as follows:

```
String String::operator / (const String& source2){
  String s = *this;
  s += source2;
  return s;
}
```

The call appears to be both symmetric and infix:

```
String s1, s2, s4;
s4 = s1 / s2;
```

Nonetheless, it has one slight asymmetry, what is really happening is:

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

There are consequences, as pointed out in subsection 10.5.4.

10.5.4 Coercion of Arguments

There is a very subtle difference between the non-member `operator+` and the member `operator /` above. The relevant rule is the following: if an operator function *is* a member, its first argument *must* match the class type. The second argument is not so restricted, since, if one is available, an appropriate constructor will be invoked to perform a *coercion* (an implicit conversion).

In the case of a non-member operator, coercion can be applied to either argument. Example:

```
String a, b, c;

c = "abcd" + "xyz" ; //OK if non-member +;
                //constructor String(char *) is used to convert

c = a + "asdf"; //OK in either case, non-member,
c = a / "asdf"; // and member.

c = "abdcef" + b; //OK only if non-member + .
```

10.5.5 Constructors for conversion — explicit

In subsection 10.5.4, we referred to the automatic invocation of the constructor `String(char *)` when e.g. `"xyz"` appears where a `String` object is expected.

C++ allows us to forbid the use of a constructor as a conversion function: in `string.h` we can declare the constructor `explicit`, i.e.

```
explicit String(const char* cstr);
```

Chapter 11

Templates

11.1 Introduction

There are many examples of cases where we develop a class, e.g. `Array`, or `List`, which manipulates objects of a particular type, e.g. `int`, only to find that we need to adapt it to work for `float` or other types, or, indeed, for user defined classes. In the `Array` and `List` examples in Chapter 12 the *element-type* is the only part which changes; thus, it seems a pity to have to have to develop additional classes.

The same goes for functions, e.g. `swap`: all `swap` functions have the same functional requirements and usually they are implemented as three assignments. If you were to adapt an `int` swap function to `float`, or even `string`, you would merely exchange all occurrences of `int` for `float` – or `String`, etc..., as the case may be.

In such cases, what we require is often termed *generic* software; we have been using *standard library* templates in the previous chapter; Java introduced *generic* (template) collection classes in Java 1.5.

This chapter discusses *templates*. With templates, we can develop a module or function in which the type / class of one or more elements is given as a *parameter*. The actual value of the type parameters is specified at compile time.

Of course, the main objective is *reuse*. As we have argued before, the least acceptable form of reuse is *duplicate-and-modify* – simply because each duplication creates an *new*, additional, module to be tested, documented and maintained.

First we will give a quick discussion of the implementation of a template `Array` – the `int` version of which we have become very familiar with. Although the syntax of template classes looks complex at first, we will see that there is little new to be learned – beyond what was learned in the development of the original `Array(int)` class. In fact, the main advice is: design your data abstraction (array, list, queue, whatever), then add the template bits – using an established template class as an example (a template, in fact!).

First, we provide an example of a template function.

11.2 Template Functions

11.2.1 Overloaded Functions recalled

The two *overloaded* `swap` functions shown below clearly invite use of a type parameter.

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

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

Overloading function names – Ad Hoc Polymorphism The overloading of function names is yet another example of *polymorphism*, in this case called *ad hoc polymorphism*. It is called *ad hoc* because, even though the name of the function (swap) is the same, there is nothing to ensure that they are similar in other than name; e.g. we could define `int swap(String s)` which gives the length of the String!

11.2.2 Template Function

A template version of `swap` is shown below:

```
template <class T> void swap(T& p, T& q)
{
  T temp = p;  p = q; q = temp;
}
```

The program `swapT.cpp` shows use of this `swap`. Here, we are able to swap `int`s, `float`s, and `string` objects. In the case of user defined classes, the only requirement is that the assignment operator = is defined.

```
//--- swapT.cpp ---------------------------
// j.g.c. 14/1/99, 2007-02-11
//-----------------------------------------
#include <iostream>
#include <string>

using namespace std;

// there is a std::swap, so we call this one swap1
template <class T> void swap1(T& p, T& q)
{
  T temp = p;
  p = q;
  q = temp;
}

int main()
{
  int a = 10, b = 12;
  float x = 1.25, y = 1.30;

  cout<< "a, b = "<< a<< " " << b<< endl;
  swap1(a, b);
  cout<< "a, b (swapped) = "<< a<< " " << b<< endl;

  cout<< "x, y = "<< x<< " " << y<< endl;
  swap1(x, y);
  cout<< "x, y (swapped) = "<< x<< " " << y<< endl;

  string s1("abcdef"), s2("1234");
  cout<< "s1 = "<< s1<< endl;
  cout<< "s2 = "<< s2<< endl;

  swap1(s1, s2);

  cout<< "swapped: ";
  cout<< "s1 = "<< s1<< endl;
  cout<< "s2 = "<< s2<< endl;
```

```
  return 0;
}
```

11.3 Polymorphism – Parametric

Templates are another form of *polymorphism*; using templates one can define, for example a polymorphic `List` – a List of `int`, or `float`, or `string`, etc... Likewise we can define a polymorphic `swap`. Unlike the polymorphism encountered in chapter 11 (`Person` and `Cell` class hierarchies), the type of each instance of a generic unit is specified at compile time – *statically*.

Hence, the polymorphism offered by templates is called *parametric polymorphism*.

11.4 Template Array

11.4.1 Class declaration and implementation

The declaration for the template `Array` class is shown in file `arrayt.h`.

The first thing to note about templates, at least using the GNU C++ compiler, is that both declaration and implementation (normally `.cpp` file) must be `#included`. Hence we include them in the same file, here `arrayt.h`.

```
//----- ArrayT.h ----------------------------------
// j.g.c. 9/1/99, 2007-02-11
// template array
//-------------------------------------------------
#ifndef ARRAYTH
#define ARRAYTH

#include <iostream>
#include <cassert>

using std::ostream; using std::endl;

template <class T> class Array{
public:
  Array(const unsigned int len= 0);
  Array(const unsigned int len, const T& val);
  Array(const Array<T>& source);
  ~Array();
  Array& operator=(const Array<T>& source);
  void copy(const Array<T>& source);
  T& operator[](unsigned int i) const;
  unsigned int length() const;
```

```
private:
  unsigned int len_;
  T* dat_;
};

template <class T> ostream& operator<<(ostream& os, const Array<T>& a);

template <class T> Array<T>::Array(const unsigned int len) : len_(len) {
  if(len== 0){dat_ = 0; return;}
  dat_ = new T[len_];
  assert(dat_!= 0);
  T zero(0);
  for(unsigned int i=0; i< len_; i++)dat_[i]= zero;
}

template <class T> Array<T>::Array(const unsigned int len, const T& val)
      : len_(len) {
  if(len== 0){dat_ = 0; return;}
  dat_ = new T[len_];
  assert(dat_!= 0);
  for(unsigned int i=0; i< len_; i++)dat_[i]= val;
}

template <class T> Array<T>::Array(const Array<T>& source){
  copy(source);
}

template <class T> Array<T>::~Array(){
  delete [] dat_;
}

template <class T> Array<T>& Array<T>::operator=(const Array& source){
  if(this!= &source){ // beware a= a;
    delete [] dat_;
    copy(source);
  }
  return *this;
}

template <class T> void Array<T>::copy(const Array<T>& source){
  len_= source.length();
  if(len_== 0){dat_ = 0; return;}
  dat_ = new T[len_];
  assert(dat_!= 0);
  for(unsigned int i= 0; i< len_; i++)dat_[i] = source.dat_[i];
}

template <class T> T& Array<T>::operator[](unsigned int i) const {
  assert(i<len_);
```

```
  return dat_[i];
}

template <class T> unsigned int Array<T>::length() const {
  return len_;
}
    // notice no Array<T>:: -- *not* a member
template <class T> ostream& operator<<(ostream& os, const Array<T>& a)
{
  unsigned int len= a.length();
  os<< "[" << len << "]{ ";
  for(unsigned int i=0; i< len; i++){
    os<< a[i];
    if(i!= len-1)os<<", ";
  }
  os<<"}"<< endl;
  return os;
}
#endif
```

Dissection of ArrayT.h

1. The class-type parameter <T>. As already mentioned, `Array` is now a a parameterised class – the type/class of its element is given as a 'class' parameter, <T>. Just as in declaration of a function with parameters (variables – formal parameters), we must announce this fact, and give the (as yet unknown) parameter an identifier. This is done as follows:

   ```
   template <class T> class Array
   ```

   Here we are saying: `Array` uses an as yet unspecified type/class `T`.

2. Instead of <class T>, we may use <typename T>;

3. When we want to define an *actual* `Array` in a program, we do so as, e.g., <int>:

   ```
   Array <int> c1(5);
   Array <int> c2(5, 127);
   ```

   or `double`:

   ```
   Array <double> c3(3, 3.14159);
   ```

   or even an `Array` of `Array<int>` (a matrix):

   ```
   Array <Array<int> > c4(3);
   ```

   Thus, as with (parameterised) functions, we can invoke the abstraction with any number of (different) parameters.

4. Likewise, if we ever need to declare a `Array` object as a parameter, we use the form:

   ```
   Array(const Array<T> & other);
   ```

   i.e. `other` is a reference to a `Array` object of `classT` elements.

5. Instantiation. Just as variables and objects are *instantiated* (created), so are template classes. In the case of template classes and functions, this is done at compile time.

6. In GNU C++, the `Array<int>` class, for example, is instantiated, at compile time, by a form of macro-expansion. This is the reason that function implementations must also be in the `.h` file.

11.4.2 A simple client program

As we have seen, there is little to templates.

The program `ArrayTT1.cpp` demonstrates the use of the template `Array` class; it creates, in the same program, `Arrays` of both `int`, `double` and `Array<int>` (an array of int arrays); note the the array-of-array contains empty arrays.

If you ever needed to create a two-dimensional array, `Array <Array<int> >`, would be the way to do it.

```
//----- ArrayTT1.cpp ----------------------------------
// j.g.c. 9/1/99, 2007-02-11
//-----------------------------------------------------
#include "ArrayT.h"
using std::cout;

int main(){
  Array <int> c1(5);
  cout<< "Array <int> c1(5): "<< c1;

  Array <int> c2(5, 127);
  cout<< "Array <int> c2(5, 127): "<< c2;

  Array <double> c3(3, 3.14159);
  cout<< "Array <double> c3(3, 3.14159): "<< c3;

  Array <Array<int> > c4(3);
  cout<< "Array <Array<int> > c4(3): "<< c4;

  cout<< endl;
  return 0;
}
```

Chapter 12

Array Containers

12.1 Introduction

We call this chapter *array containers* for want of a better term. We describe *sequence containers* which remedy some of the shortcomings of the basic C++ array (e.g. `int a[50];`).

The two most common *sequence containers* that we encounter in C++ are `std::vector` and `std::list`; we use the general term *sequence* to signify that the elements are held in strict sequential (linear) order.

The main difference between *array-like* sequence containers (e.g. `std::vector` and `Array` that we develop here) and *linked* (list-like) sequence containers are the two related characteristics: (i) array-like have *random access*, e.g. `std::cout<< a[i];`, whereas lists must be sequentially accessed; and (ii) array-like use contiguous storage, whereas lists used linked storage (e.g. singly and doubly linked lists).

Roughly speaking, you can do anything with an array-like sequence that you can with a linked sequence and *vice-versa*, but the allocation of storage and random access issues mean that there are major performance drawbacks if you choose the wrong type for your application — i.e. the wrong type may *work*, but work comparatively very slowly.

Java programmers are aware of the distinction, they have `ArrayList` and `LinkedList`.

In this chapter we will comment on the inadequacy of the C++ basic array; we will develop an `Array` class that performs rather like `std::vector`; in doing that we will develop some understanding of what goes on inside `std::vector`, so that when you use `std::vector` you will have some sympathy for performance issues. Then we will examine use of `std::vector` itself.

In developing `Array`, we will identify some inadequacies of contiguous storage that lead to the need for linked storage. We will cover (linked) lists in Chapter 13.

The `Array` class here is more or less identical to the `vector` class in (Budd 1997a).

12.2 An Array class

In the lecture, we will discuss the inadequacies of *plain-vanilla* arrays (`int a[25];`) for the task of sequence container. For example, just look in `Array.h` at what needs to be done when we

insert (add) an element into the middle of a list, i.e. moving everything else up one. To ask a programmer who is concentrating of doing *sequence operations* to remember this every time is to ask for increased errors and poor productivity.

Figure 12.1 shows the interface of `Array`. Figures 12.2 to 12.5 give the implementation.

```
/* ----- Array.h ----------------------------------
  j.g.c. 9/1/99, 2007-02-11, 2008-01-13, 2008-01-19
  template unordered array
  2008-01-22, iterator added based on Budd (1998)
  2008-02-05 bugs fixed in insert, insert1
-------------------------------------------------*/
#ifndef ARRAYH
#define ARRAYH

#include <iostream>
#include <cassert>

using std::ostream; using std::endl;
typedef unsigned int uint;

template <class T> class Array{
public:
  typedef T* Iterator;
  Array(uint len= 0);
  Array(uint len, const T& val);
  Array(const Array<T>& source);
  ~Array();
  void reserve(uint cap);
  void resize(uint sz);
  Array& operator=(const Array<T>& source);
  void copy(const Array<T>& source);
  T& operator[](uint i) const;
  void push_back(const T& val);
  void pop_back();
  T back() const;
  void insert1(uint pos, uint n, const T& val);
  uint erase(uint pos);
  Iterator begin(){ return dat_;}
  Iterator end(){ return dat_ + sz_;}
  void insert(Iterator itr, uint n, const T& e);
  uint size() const;
  bool empty() const;
  uint capacity() const;
private:
  uint sz_, cap_;
  T* dat_;
};
template <class T>
ostream& operator<<(ostream& os, const Array<T>& a);
```

Figure 12.1: Declaration of Array.

```
template <class T>
Array<T>::Array(uint sz) : sz_(sz), cap_(sz) {
  if(cap_== 0){dat_ = 0; return;}
  dat_ = new T[cap_];
  assert(dat_!= 0);
  T zero = T();
  for(uint i = 0; i< sz_; ++i)dat_[i]= zero;
}

template <class T>
void Array<T>::reserve(uint cap){
  //std::cout<< "*reserve* cap = "<< cap<< " cap_ = "<< cap_<< std::endl;
  if(cap <= cap_)return;
  T* newdat = new T[cap];
  assert(newdat != 0);
  for(uint i = 0; i< sz_; ++i)newdat[i] = dat_[i];
  delete [] dat_;
  dat_ = newdat;
  cap_ = cap;
}

template <class T>
void Array<T>::resize(uint sz){
  assert(sz<= cap_);
  if(sz <= sz_)return;
  else if(sz <= cap_){
    T zero = T();
    for(uint i = sz_; i< sz; ++i)dat_[i] = zero;
    sz_ = sz;
  }
  else {
    T* newdat = new T[sz];
    assert(newdat != 0);
    for(uint i = 0; i< sz_; ++i)newdat[i] = dat_[i];
    T zero = T();
    for(uint i = sz_; i< sz; ++i)newdat[i] = zero;
    delete [] dat_;
    dat_ = newdat;
    cap_ = sz_ = sz;
  }
}
```

Figure 12.2: Array implementation, part 1

```
template <class T>
Array<T>::Array(uint sz, const T& val)
      : sz_(sz), cap_(sz) {
  if(cap_ == 0){dat_ = 0; return;}
  dat_ = new T[cap_];
  assert(dat_!= 0);
  for(uint i=0; i< sz_; ++i)dat_[i]= val;
}

template <class T>
Array<T>::Array(const Array<T>& source){
  copy(source);
}

template <class T>
Array<T>::~Array(){
  delete [] dat_;
  dat_ = 0;
}

template <class T>
Array<T>& Array<T>::operator=(const Array& source){
  //std::cout<< "copy ctor"<< std::endl;
  if(this!= &source){ // beware a= a;
    delete [] dat_;
    copy(source);
  }
  return *this;
}

template <class T>
void Array<T>::copy(const Array<T>& source){
  sz_= cap_ = source.size();
  if(cap_== 0){dat_ = 0; return;}
  dat_ = new T[cap_];
  assert(dat_!= 0);
  for(uint i= 0; i< sz_; ++i)dat_[i] = source.dat_[i];
}

template <class T>
void Array<T>::insert1(uint pos, uint n, const T& val) {
  uint sz = sz_ + n;
  assert(cap_>= sz);
  uint id= sz - 1;  // dest
  uint is= sz_ - 1; // source
  for(; is> pos-1; --is, --id)dat_[id] = dat_[is];
  for(id = pos; id< pos+n; id++)dat_[id] = val;
  sz_ = sz;
}
```

Figure 12.3: Array implementation, part 2

```
template <class T>
uint Array<T>::erase(uint pos) {
  for(uint i = pos + 1; i< sz_; ++i)dat_[i - 1] = dat_[i];
  --sz_;
  return pos;
}

template <class T>
void Array<T>::insert(Iterator pos, uint n, const T& e){
  uint sz = sz_ + n;
  assert(cap_>= sz);
  Iterator itrd= dat_ + sz - 1;  // dest
  Iterator itrs= dat_ + sz_ - 1; // source
  for(; itrs!= pos-1; --itrs, --itrd)*itrd = *itrs;
  for(itrd = pos; itrd!= pos + n; ++itrd)*itrd = e;
  sz_ = sz;
}

template <class T>
void Array<T>::push_back(const T& val) {
  //std::cout<< "*push_back* cap_ = "<< cap_<< " sz_ = "<< sz_<< std::endl;
  if(!(cap_> sz_))reserve(2*cap_);
  dat_[sz_] = val;
  ++sz_;
}

template <class T>
void Array<T>::pop_back() {
  assert(sz_> 0);
  --sz_;
}

template <class T>
T Array<T>::back() const {
  assert(sz_> 0);
  return dat_[sz_ - 1];
}

template <class T>
T& Array<T>::operator[](uint i) const {
  assert(i < sz_);
  return dat_[i];
}

template <class T>
uint Array<T>::size() const {
  return sz_;
}
```

Figure 12.4: Array implementation, part 3

```
template <class T>
bool Array<T>::empty() const {
  return sz_ == 0;
}

template <class T>
uint Array<T>::capacity() const {
  return cap_;
}


    // notice no Array<T>:: -- *not* a member
template <class T>
ostream& operator<<(ostream& os, const Array<T>& a){
  uint sz= a.size();
  uint cap = a.capacity();
  os<< "[" << sz<<", "<< cap << "]{ ";
  for(uint i=0; i< sz; i++){
    os<< a[i];
    if(i!= sz-1)os<<", ";
  }
  os<<"}"<< endl;
  return os;
}
#endif
```

Figure 12.5: Array implementation, part 4

12.2.1 Major points to note in Array.h

1. Note the difference between `size(), sz_` and `capacity(), cap_`; `size(), sz_` is the *used* size of the sequence, while `capacity(), cap_` is what it can grow to before we need to allocate more memory.

2. When using `Array` (and `std::vector`) it is always a good idea to use `reserve` to allocate either the size that you know you will need, or a decent chunk at the beginning.

3. Notice the difference between `reserve` and `resize`.

4. Notice that when `push_back` detects that we are at full capacity, it reserves double the current capacity; this might work well or badly, depending on the application. I'm not sure how `std::vector` handles this.

   [Here we repeat some messages from Chapter 9.]

5. Destructors. A *destructor* is called, automatically, when control reaches the end of the block in which the object was declared, i.e. when the object goes out of scope. The compiler will always provide adequate destruction of stack-based objects, but, for heap-based objects, proper destructor memory management must be provided if `garbage` and memory-leaks (or worse, *dangling pointers*) are to be avoided.

6. Copy constructor. A *copy constructor* is called when the object is passed (*by value*) to and from functions. Again, for classes which use stack memory, the compiler will always provide an adequate copy constructor. For heap-based objects the case is quite analogous to that of the destructor: proper constructor memory management must be provided.

7. Assignment. *Assignment operator* (the '=' operator) needs treatment similar to the copy constructor.

8. The Big-Three. The C++ FAQs (Cline et al. 1999a) uses the term *the Big-Three* for these three functions: copy constructor, assignment operator, and destructor. More on this in section 12.4.

   This is because, for classes that use heap storage, it is almost certainly necessary to explicitly program all three. If they are not programmed, the compiler will provide default versions which will probably not do what you would wish; moreover the inadequacies of these defaults may be most subtle, and may require quite determined and skilled testing to detect.

9. We use `assert` to report errors if allocations were unsuccessful; `new` returns a null pointer if it is unsuccessful, e.g. due to resources of free memory having become exhausted. This may not be ideal, but the error message that `assert` issues is a lot more helpful than what will happen if we charge on and ignore the fact that we have run out of memory.

10. Function copy.

```
void Array<T>::copy(const Array<T>& source){
  sz_= cap_ = source.size();
  if(cap_== 0){dat_ = 0; return;}
  dat_ = new T[cap_];
  assert(dat_!= 0);
  for(uint i= 0; i< sz_; ++i)dat_[i] = source.dat_[i];
}
```

(a) This is very similar to the default constructor — except that this time we have passed another object to be copied.

(b) Notice that we have passed a reference (`Array& source`). If we don't we'll end up constructing many multiple copies, each of which must also be destroyed. And when the object becomes large, as may be the case for an `Array` the performance drain can be considerable.

(c) Of course, we guarantee the safety of the referenced object (in the caller) by making the reference `const`.

11. Copy constructor.

```
Array::Array(const Array& source){
  copy(source);
}
```

Here, all the work is done by `copy`. Again notice the use of reference and `const`.

12. Destructor.

```
Array::~Array(){
 delete [] dat_;
}
```

(a) Here we use `delete` to release the allocated memory.

(b) Owing to the fact that `dat_` points to an array, we must use `delete []`.

13. Assignment operator '='.

```
Array& Array::operator=(const Array& source){
  if(this!= &source){ // beware a= a;
    delete [] dat_;
    copy(source);
  }
  return *this;
}
```

(a) Let us say we have two `Array` objects, `x, y` and `x = y`. Then this assignment is *exactly* equivalent to

```
x.operator=(y);
```

(b) Again notice the use of a reference and `const`.

(c) Since we can envisage `x = x`, however improbable, we have to be careful to check whether this is the case — and if it is, do nothing.

(d) **this** is an implicit pointer variable which points at the object itself. Thus `if(this!= &source)` checks if the calling object and the source object share the same memory location — and so are the same object!

(e) `return *this;` returns the object (actually a reference, see next comment).

(f) Why does `operator=` return `Array&`?

Answer. In C and C++ it is standard for an assignment to have the value of the object assigned, e.g.

```
int x = y = 10;
```

we want the same for objects.

And, as usual, we want the efficiency of reference passing.

14. If you are unsure about operator overloading, refer to Chapters 9 and 10
15. Iterators are described in section 12.2.2.

12.2.2 Iterators

Because of the fact that the internal representation is a C++ built-in array, we can do everything we want to using array subscripting, overloading the array subscript operator `[]` e.g. `a[index] = 10;` and integer subscripts.

But the STL collections use the more general concept of an *iterator*. We will see that *iterator*s are made to behave like pointers, i.e. you can dereference an iterator, and you can do *pointer-like arithmetic* on them.

1. Normally an iterator is implemented as a class, but in the case of `Array` it is easy to declare an iterator as

   ```
   typedef T* Iterator;
   ```

2. Because `Iterator` is declared within the *scope* of `Array`, when we define an iterator we use the scope resolution operator `::`

   ```
   Array<int>::Iterator itr;
   ```

3. We need an iterator that points to the beginning of the array

   ```
   Iterator begin(){ return dat_;}
   ```

   It simply returns a pointer to the first element of data array.

4. We need an iterator that points **one element past the end of the array**;

   ```
   Iterator end(){ return dat_ + sz_;}
   ```

5. One element past the end of the array is the convention; you *never* dereference this iterator value. The usual pattern for iterating over a collection is as follows:

```
Array<int>::Iterator itr;
for(itr = c5.begin(); itr!= c5.end(); ++itr){
  cout<< *itr<< ' ';
}
```

`itr!= c5.end()` is our way of checking that we are not at the end — really one element past the end.

This is very like the normal array pattern:

```
int a[n];
for(int i = 0; i< n; ++i){
  cout<< a[i]<< ' ';
}
```

which could be written:

```
int a[n];
for(int i = 0; i!= n; ++i){
  cout<< a[i]<< ' ';
}
```

or even:

```
int a[n];
for(int* pos = &a[0]; pos!= &a[0]+ n; ++pos){
  cout<< *pos<< ' ';
}
```

or:

```
int a[n];
for(int* pos = a; pos!= a + n; ++pos){
  cout<< *pos<< ' ';
}
```

6. Here is *insert* written using subscripts

```
template <class T>
void Array<T>::insert1(uint pos, uint n, const T& val) {
  uint sz = sz_ + n;
  assert(cap_>= sz);
  uint id= sz - 1;  // dest
  uint is= sz_ - 1; // source
  for(; is> pos-1; --is, --id)dat_[id] = dat_[is];
  for(id = pos; id< pos+n; id++)dat_[id] = val;
  sz_ = sz;
}
```

7. And here it is using `Iterator`

```
template <class T>
void Array<T>::insert(Iterator pos, uint n, const T& e){
  uint sz = sz_ + n;
  assert(cap_>= sz);
  Iterator itrd= dat_ + sz - 1;  // dest
  Iterator itrs= dat_ + sz_ - 1; // source
  for(; itrs!= pos-1; --itrs, --itrd)*itrd = *itrs;
  for(itrd = pos; itrd!= pos + n; ++itrd)*itrd = e;
  sz_ = sz;
}
```

We'll have a good deal more to say about iterators when we get to Chapter 13.

12.3 A Simple Client Program for Array

Figures 12.6 and 12.7 give a program, `ArrayT1.cpp` which uses `Array`.

```
/* ----- ArrayT1.cpp ---------------------------------
  j.g.c. 9/1/99, 2007-02-11, 2008-01-13, 2008-01-19, 2008-01-22
-----------------------------------------------------*/
#include "Array.h"
using std::cout; using std::endl;  typedef unsigned int uint;

int main(){
  Array <int> c1(5);
  cout<< "Array <int> c1(5): "<< c1;

  Array <int> c2(5, 127);
  cout<< "Array <int> c2(5, 127): "<< c2;

  Array <double> c3(3, 3.14159);
  cout<< "Array <double> c3(3, 3.14159): "<< c3;

  Array <Array<int> > c4(3);
  cout<< "Array <Array<int> > c4(3): "<< c4;

  c4[0] = c1;  c4[1] = c2;
  cout<< "c4[0] = c1, c4[1] = c2: "<< c4;  cout<< endl;

  c1.reserve(20);
  cout<< "c1.reserve(20): "<< c1;

  c1.resize(10);
  cout<< "c1.resize(10): "<< c1;

  c1.push_back(22);
  cout<< "c1.push_back(22): "<< c1;

  c1.insert1(2, 5, 33);
  cout<< "c1.insert1(2, 5, 33): "<< c1;

  c1.push_back(44);
  cout<< "c1.push_back(44): "<< c1;

  c1.pop_back();
  cout<< "c1.pop_back(): "<< c1;

  uint j = c1.erase(4);
  cout<< "j = c1.erase(4): "<< "j = "<< j<< " c1 = "<< c1;

  for(uint i = 0; i< 20; ++i){
    c1.push_back(i);
    cout<< "c1.push_back(i = "<< i<< "): "<< c1;
  }

  cout<< "c1.back(): " << c1.back();  cout<< endl;
                                        // continued ....
```


Figure 12.6: ArrayT1.cpp part 1

```
  Array<int> c5;
  c5.reserve(10);
  for(uint i = 0; i< 5; ++i){
    c5.push_back(i);
    cout<< "c5.push_back(i = "<< i<< "): "<< c5;
  }

  cout<< "using Iterator"<< endl;
  Array<int>::Iterator itr;
  for(itr = c5.begin(); itr!= c5.end(); ++itr){
    cout<< *itr<< ' ';
  }
  cout<< endl;

  cout<< "Array<int>::Iterator itr1 = c5.begin() + 2 "<< endl;
  cout<< "c5.insert(itr1, 5, 33) "<< endl;
  c5.insert1(2, 5, 33);
  cout<< c5;

  cout<< endl;
  return 0;
}
```

Figure 12.7: ArrayT1.cpp part 2

And here is the output from `ArrayT1`.

```
Array <int> c1(5): [5, 5]{ 0, 0, 0, 0, 0}
Array <int> c2(5, 127): [5, 5]{ 127, 127, 127, 127, 127}
Array <double> c3(3, 3.14159): [3, 3]{ 3.14159, 3.14159, 3.14159}
Array <Array<int> > c4(3): [3, 3]{ [0, 0]{ }
, [0, 0]{ }
, [0, 0]{ }
}
c4[0] = c1, c4[1] = c2: [3, 3]{ [5, 5]{ 0, 0, 0, 0, 0}
, [5, 5]{ 127, 127, 127, 127, 127}
, [0, 0]{ }
}

c1.reserve(20): [5, 20]{ 0, 0, 0, 0, 0}
c1.resize(10): [10, 20]{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
c1.push_back(22): [11, 20]{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 22}
c1.insert(2, 5, 33): [16, 20]{ 0, 0, 33, 33, 33, 33, 33, 0, 0, 0, 0, 0, 0,
0, 0, 22}
c1.push_back(44): [17, 20]{ 0, 0, 33, 33, 33, 33, 33, 0, 0, 0, 0, 0, 0, 0,
0, 22, 44}
c1.pop_back(): [16, 20]{ 0, 0, 33, 33, 33, 33, 33, 0, 0, 0, 0, 0, 0, 0, 0,
22}
j = c1.erase(4): j = 4 c1 = [15, 20]{ 0, 0, 33, 33, 33, 33, 0, 0, 0, 0, 0,
0, 0, 0, 22}
c1.push_back(i = 0): [16, 20]{ 0, 0, 33, 33, 33, 33, 0, 0, 0, 0, 0, 0, 0,
0, 22, 0}
c1.push_back(i = 1): [17, 20]{ 0, 0, 33, 33, 33, 33, 0, 0, 0, 0, 0, 0, 0,
0, 22, 0, 1}
  ... etc. ...
c1.push_back(i = 19): [35, 40]{ 0, 0, 33, 33, 33, 33, 0, 0, 0, 0, 0, 0, 0,
0, 22, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19}
c1.back(): 19

c5.push_back(i = 0): [1, 10]{ 0}
c5.push_back(i = 1): [2, 10]{ 0, 1}
c5.push_back(i = 2): [3, 10]{ 0, 1, 2}
c5.push_back(i = 3): [4, 10]{ 0, 1, 2, 3}
c5.push_back(i = 4): [5, 10]{ 0, 1, 2, 3, 4}
using Iterator
0 1 2 3 4
Array<int>::Iterator itr1 = c5.begin() + 2
c5.insert(itr1, 5, 33)
[10, 10]{ 0, 1, 33, 33, 33, 33, 33, 0, 3, 4}
```

12.4 The Big-Three

The C++ FAQs (Cline et al. 1999a) introduced the term *the Big-Three* for the three functions: *copy constructor*, *assignment operator*, and *destructor*.

This is because, for classes that use heap storage, it is almost certainly necessary to explicitly program all three. If they are not programmed, the compiler will provide default versions which will probably not meet the requirements of client programs. Nevertheless, the inadequacy of these defaults may be most subtle, and may require very detailed testing to detect.

The lack of a proper destructor would be particularly difficult to detect — it simply causes garbage, whose effect is to *leak memory*, which may not be detected until the class is used in some application which runs for a long time, e.g. an operating system, or an embedded control program.

Other than the *Big-Three* care must be exercised with *comparison*, e.g. equality check ==. If left to its own devices the compiler will provide a *shallow* compare, which compares just the explicit member(s).

12.4.1 Defence against naive defaults

If the developer of a heap-based class is quite sure that assignment, =, will never be required, it still may be quite dangerous to trust that client programmers will never be tempted to use it; and, normally, as we have said above, the compiler will provide a naive default – but silently, with no warning of its possible inadequacy.

Fortunately, there is a simple and sure defence; this is to declare a *stub* of the function, and to make it `private`, i.e.

```
private:
  Array& operator = (const Array & rhs){};
```

This means that any client program which unwittingly invokes an assignment will be stopped by a compiler error; client programs are not allowed to access private members.

This would work also for compare, ==.

Nevertheless, it is hard to envisage a class which can operate successfully without a proper copy constructor; likewise destructor. These will have to be programmed.

12.5 Overloading the Stream Output Operator

The stream output operator << is overloaded as follows:

```
    // notice no Array<T>:: -- *not* a member
template <class T>
ostream& operator<<(ostream& os, const Array<T>& a){
  uint sz= a.size();
  uint cap = a.capacity();
  os<< "[" << sz<<", "<< cap << "]{ ";
  for(uint i=0; i< sz; i++){
    os<< a[i];
    if(i!= sz-1)os<<", ";
  }
  os<<"}"<< endl;
  return os;
}
```

`ostream` is a *output-stream*, so that the parameter `os` is an output-stream object; the familiar `cout` is one such object.

It is important, to make it conform to its pattern of operation for the built-in types, that << returns a reference to the `ostream` object. This allows concatenated calls to it, e.g.:

```
cout<< x<< y<< z<< ";"<< endl;
```

If you are feeling confused about the distinction between member functions and *non*-member functions such as `ostream& operator<<` above, please consult Chapter 10.

12.6 std::vector

As stated previously, the chief reason for developing `Array` is as an educational experience. `std::vector` will do everything that `Array` does only faster and with much less chance of a bug lurking somewhere in its implementation.

If you read games programming books of more than five years ago, you may find ambivalence to the *standard library* (STL) — for a start, not all compilers included it, and in addition there was suspicion about its efficiency.

That's in the past. I cannot imagine any reason why anyone would ever *roll their own* array class — except for educational or some very special reasons.

Figures 12.8 and 12.9 show a program which uses `std::vector` in almost the same way as Figure 12.6 uses `Array`.

```
/* ----- vectorT1.cpp ----------------------------------
  from Arrayt1.cpp, j.g.c. 2008-01-15
-------------------------------------------------------*/
#include <vector> #include <iostream>
#include <ostream> #include <iterator>
#include <algorithm>
using namespace std;

int main(){
  vector <int> c1(5);
  cout<< "vector <int> c1(5): ";
  copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " "));  cout<< endl;

  vector <int> c2(5, 127);
  cout<< "vector <int> c2(5, 127): ";
  copy(c2.begin(), c2.end(), ostream_iterator<int>(cout, " "));  cout<< endl;

  vector <double> c3(3, 3.14159);
  cout<< "vector <double> c3(3, 3.14159): ";
  copy(c3.begin(), c3.end(), ostream_iterator<double>(cout, " "));  cout<< endl;

  c1.reserve(20);  cout<< "c1.reserve(20): ";
  copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " "));  cout<< endl;

  c1.resize(10);  cout<< "c1.resize(10): ";
  copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " "));  cout<< endl;

  c1.push_back(22);  cout<< "c1.push_back(22): ";
  copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " "));  cout<< endl;

  c1.insert(c1.begin() + 2,  5, 33);
  cout<< "c1.insert(2, 5, 33): ";
  copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " "));  cout<< endl;

  c1.push_back(44);  cout<< "c1.push_back(44): ";
  copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " "));  cout<< endl;

  c1.pop_back();  cout<< "c1.pop_back(): ";
  copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " "));  cout<< endl;

  c1.erase(c1.begin());  cout<< endl;
  cout<< "c1.erase(c1.begin()): "<< " c1 = ";
  copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " "));  cout<< endl;

  for(size_t i = 0; i< 20; ++i){
    c1.push_back(i);       cout<< "c1.push_back(i = "<< i<< "): ";
    copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " ")); cout<< endl;
  }
```

Figure 12.8: Use of std::vector

```
  copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " "));
  cout<< endl;

  uint n = c1.size();
  for(uint i = 0; i!= n; ++i){
    cout<< c1[i]<< " ";
  }
  cout<< endl;

  vector<int>::iterator it;
  for(it = c1.begin(); it!= c1.end(); ++it){
    cout<< *it<< " ";
  }

  cout<< "\nsort(c1.begin(), c1.end())"<< endl;
  sort(c1.begin(), c1.end());
  copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " "));
  cout<< endl;

  return 0;
}
```

Figure 12.9: Use of std::vector

12.6.1 Points to note in vectort1.cpp

1. `#include <vector>`.
2. Unless you want to fully qualify the type as `std::vector`, you must include `using namespace std`.
3. You can *randomly access* elements of `vector` using `[]` notation, for example `cout<< c1[2]`, see Figure 12.9.
4. You can access elements of `vector` using and `iterator`, for example

```
vector<int>::iterator it;
for(it = c1.begin(); it!= c1.end(); ++it){
  cout<< *it<< " ";
}
```

5. We declare the `iterator` as `vector<int>::iterator it;` because that `iterator` is declared within the scope of `vector`; recall `Array<T>::Iterator` above; `::` is the scope resolution operator.
6. When we come to using `std::list`, we will have to use an `iterator` because `std::list` does not support *random access*.
7. An `iterator` is meant to behave rather like a pointer: (a) `++it` and `--it` move the `iterator` forwards and backwards; (b) `*it` access the element that it refers to.
8. `c1.begin()` is an `iterator` that references the first element of `c1`.
9. `c1.end()` is an `iterator` that references *one element past the last element* of `c1`. Hence `it!= c1.end()`.
10. Although we show explicit use of `iterator` and `[]` indexed random access, the standard library gives us alternatives, for example

```
copy(c1.begin(), c1.end(), ostream_iterator<int>(cout, " "));
```

11. The standard library algorithms have been written in such a manner that they can be applied to *pointers* as well as *iterators*.
12. Notice how to sort a vector:

```
sort(c1.begin(), c1.end());
```

Chapter 13

Linked Lists

13.1 Introduction

The two most common form of *sequence* data structures are (a) contiguous *array-based* as covered in Chapter 12 and (b) *linked-list-based* that we cover in this Chapter. In what follows, when comparing them, we'll use the term *array* for the former and *linked-list* for the latter. Apart from *sequence* data structures, we will encounter other forms of collection, e.g. trees and graphs; some of these use linked representations, some use arrays, and there are other implementations.

`std::vector` (and likewise `Array`) that we covered in Chapter 12 do a fine job for many applications. However, they run into performance difficulties in two major circumstances: (i) you need to insert one or more elements into the array at other than the back; in fact `std::vector` has no `push_front` function because it is reckoned that to use `push_front` on on array would be rather silly; (ii) you get it wrong when you declare the initial size of the array (or when you use `reserve`).

These difficulties can be seen by examining Figure 13.1.

First, `push_back` works fine until we run out of space, but then we must `reserve` more memory (an new array), and, (not shown) copy all data from the old array to the new array; that makes `push_back` an $O(N)$ operation in the worst case, and if, unlike the example, we `reserve` only what extra space that is immediately needed, then N `push_backs` will grow as $O(N^2)$.

Next, if we look at `insert`, we see that the first thing that has to be done is to copy all data that are above the insert position, n memory positions up the array to make space for the n inserts. This is an $O(N)$ operation, so that if we want to insert all N array elements in this manner, constructing the array will be $O(N^2)$.

A *linked-list* operates by treating every element in a collection as a semi-independent entity but linked to other elements in the collection so that the sequence can be maintained. Using this representation, most insert and delete operations take constant time, i.e. $O(1)$. The one downside is that accessing the i-th item in the list (sequence) takes $O(N)$ — *sequential access*, whilst an array allows *random access* and the same can be done in $O(1)$.

As we have noted earlier, the purpose of developing our own list classes, just as for the array class in Chapter 12, is mostly to develop some sympathy with how and why they work. When it comes to application development, we will almost always use STL collections and algorithms.

```
template <class T>
void Array<T>::insert(Iterator pos, uint n, const T& e){
  uint sz = sz_ + n;
  assert(cap_>= sz);
  Iterator itrd= dat_ + sz - 1;  // dstination
  Iterator itrs= dat_ + sz_ - 1; // source
  for(; itrs!= pos; --itrs, --itrd)*(dat_ + itrd) = *(dat_ + itrs);
  for(itrd = pos; itrd!= pos + n; ++itrd)*(dat_ + itrd) = e;
  sz_ = sz;
}

template <class T>
void Array<T>::push_back(const T& val) {
  if(!(cap_> sz_))reserve(2*cap_);
  dat_[sz_] = val;
  ++sz_;
}
```

Figure 13.1: Extracts from Array.h showing drawback of Array and std::vector

13.2 A Singly Linked List

Here we develop the simplest form of linked list — a singly linked list. This works fine for most list applications, but suffers from the disadvantage that inserting and deleting at the back (`push_back, pop_back`) take $O(N)$. Later, we will see how a *doubly linked list* can remedy this particular drawback.

13.2.1 Class Declaration

The declaration of the singly linked `SList` class is shown in Figure 13.2.

```
#ifndef LISTTH
#define LISTTH

#include <cassert>
#include <iostream>

template <class T>
class List;

template <class T>
std::ostream& operator<<(std::ostream& os, const List<T>& l);

template <class T> class Link;

template <class T> class List{
  friend  std::ostream& operator<< <T>(std::ostream& os, const List<T>& l);
public:
  List();
  List(const List<T> & other);
  ~List();
  List & operator = (const List<T> & rhs);
  T front() const;
  void push_front(T e);
  void pop_front();
  T back() const;
  void push_back(T e);
  void pop_back();
  void clear(); // empty the list and delete all elements in it
  bool empty() const;
  int size() const;
private:
  void copy(const List<T> & other); // local utility only, users use =
  Link<T>* first_;
};

template <class T>class Link{
  friend class List<T>;
  friend  std::ostream& operator<< <T>(std::ostream& os, const List<T>& l);
private:
  Link(T e, Link *next=0);
  T elem_;
  Link<T>* next_;
};
```

Figure 13.2: Declaration of Singly Linked List.

13.2.2 Dissection of List

[*You know all this already, but it may be worth repeating.*]

1. The type parameter `T`. As already mentioned, `List` is a parameterised class – the type / class of its element is given as a `template class` parameter. Just as in declaration of a function with parameters (variables — formal parameters), we must announce this fact, and give the (as yet unknown) parameter an identifier. This is done as follows:

   ```
   template <class T> class List
   ```

   Here we are saying: `List` uses an as yet unspecified type / class `T`.

2. When we want to define an *actual* `List` in a program, we do so as, e.g.:

   ```
   List<int> s;
   List<float> t;
   List<string> x;
   ```

   Thus, as with (parameterised) functions, we can 'use' the abstraction with any number of (different) parameters.

3. Likewise, if we ever need to declare a `List` object as a parameter, we use the form:

   ```
   List(const List<T> & other);
   ```

   i.e. `other` is a reference to a `List` object of class or type `T` elements.

4. Instantiation. Just as variables and objects are *instantiated* (created), so are template classes. In the case of template classes and functions, this is done at compile time.

5. In GNU C++, the `List<int>` class, for example, is instantiated, at compile time, by a form of macro-expansion. This is the reason that function implementations must also be in the `.h` file.

6. The only data member for a `List` is `Link<T>* first_`, a *pointer* to a `Link`.

7. When first constructed a List is empty, in which case `first_ == NULL`.

8. `List` has *the Big-Three*: copy constructor, assignment operator, and destructor.

9. Ordinarily, e.g. in the copy constructor and the assignment operator, we pass `Lists` *by reference*, e.g.

   ```
   List<T>& List<T>::operator = (const List<T>& rhs)
   ```

   This is because we don't want to make unnecessary copies.

.

Figure 13.3 gives a brief indication of the operation of `List`.

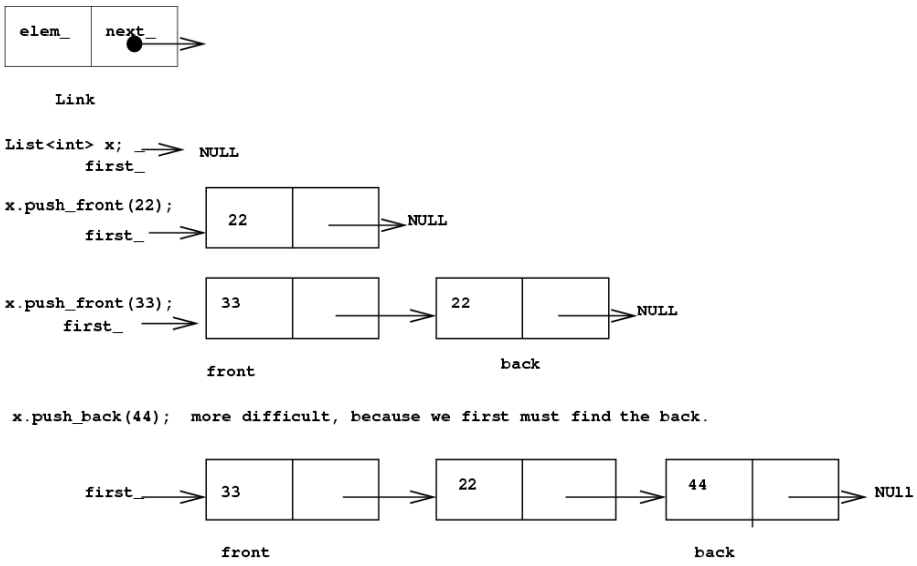


Figure 13.3: Operation of a singly linked list.

13.2.3 Class Implementation

The implementations of selected `List` class functions are shown in Figures 13.4 to 13.6. Most of these are straightforward, but it will be useful to ensure that we discuss them properly in lectures; diagrams will do a lot to aid your understanding. You should also read appropriate sections pf (Penton 2003) and execute his programs which have interactive graphics representations of singly linked and doubly linked lists.

```
template <class T>
Link<T>::Link(T e, Link<T> *next) {
  elem_ = e;
  next_ = next;
}

template <class T>
List<T>::List(){
  first_ = 0;
}

template <class T>
void List<T>::copy(const List<T> & other){
  if(other.empty())first_ = 0;
  else{
    Link<T> *pp = other.first_; //cursor to other
    Link<T> *pt = new Link<T>(pp->elem_, 0);
    first_ = pt;
    while(pp->next_ != 0){
      pp = pp->next_;
      pt->next_ = new Link<T>(pp->elem_,0);
      pt = pt->next_;
    }
  }}

template <class T>
List<T>::List(const List<T>& other){
  copy(other);
}

template <class T>
List<T> & List<T>::operator = (const List<T>& rhs){
  if(this != &rhs){ //beware of listA=listA;
    clear();
  }
  copy(rhs);
  return *this;
}

template <class T>
List<T>::~List(){  clear();     }

template <class T>
void List<T>::push_front(T e){
  Link<T> *pt= new Link<T>(e, first_);
  assert(pt != 0);
  first_=pt;
} // continued ...
```

Figure 13.4: Implementation code for Singly Linked List, part 1.

```
template <class T>
T List<T>::front() const{
  assert(!empty());
  return first_->elem_;
}

template <class T>
void List<T>::pop_front(){
  Link<T>* pt= first_;
  first_= pt->next_;
  delete pt;  pt = 0;
}

template <class T>
void List<T>::push_back(T e){
  //std::cout<< "push_back"<< std::endl;
  if(empty()){
    first_ = new Link<T>(e, 0);
    assert(first_ != 0);
  }
  else{
    Link<T>* pp = first_;
    // walk to the back
    while(pp->next_ != 0)pp = pp->next_;
    // and add a new Link with e in it and next = null
    pp->next_ = new Link<T>(e, 0);
    assert(pp->next_ != 0);
  }
}

template <class T>
void List<T>::pop_back(){
  //std::cout<< "pop_back"<< std::endl;
  assert(!empty());
  if(first_->next_ == 0){   /*kludge for one element */
    delete first_;
    first_ = 0;
    return;
  }
  Link<T> *pp(first_), *prev(first_);
  // walk to the back
  while(pp->next_ != 0){
    prev = pp;    pp = pp->next_;
  }
  // delete the last Link and set prev->next = null
  delete pp;
  pp = 0;
  prev->next_ = 0;
} // continued ...
```

Figure 13.5: Implementation code for Singly Linked List, part 2.

```
template <class T>
T List<T>::back() const{
  //std::cout<< "back"<< std::endl;
  assert(!empty());
  Link<T>* pp (first_);
  // walk to the back
  while(pp->next_ != 0)pp = pp->next_;
  return pp->elem_;
}

template <class T>
void List<T>::clear(){
  Link<T> *next,*pp(first_);
  while(pp != 0){
    next = pp->next_;
    pp->next_ = 0; // why did Budd include this?
    delete pp;
    pp = next;
  }
  first_ = 0;
}

template <class T>
bool List<T>::empty() const{
  return (first_ == 0);
}

template <class T>
int List<T>::size() const{
  int i = 0;
  Link<T> *pt = first_;
  while(pt != 0){
    pt = pt->next_;
    ++i;
  }
  return i;
}

template <class T>
std::ostream& operator<< (std::ostream& os, const List<T>& lst){
  os<<"f[ ";
  Link<T> *pp = lst.first_; //cursor to lst
  while(pp != 0){
    if(pp != lst.first_)os<<", ";
    os<< pp->elem_;
    pp = pp->next_;
  }
  os<<" ]b"<<std::endl;
  return os;
}
```


Figure 13.6: Implementation code for Singly Linked List, part 3.

13.3 A simple List client program

As we have seen, there is little to templates.

The program `ListT1.cpp` in Figures 13.7 and 13.8 demonstrate the use of the `List` class.

Figure 13.9 shows the output of `ListT1.cpp`. Please note that if you make minor modifications to the beginning of `ListT1.cpp` and change occurrences of `List` to `list` (`std::list`), the program performs the same. Although `std::list` is a *doubly-linked-list*, the interface functions hide that fact.

In the next section, we develop a *doubly linked list*.

```
/* --- ListT1.cpp -------------------------------------------
 j.g.c. 31/12/96//j.g.c. 1/1/97, 5/1/97, 2007-12-28, 2008-01-18
 changed j.g.c. 2007-12-30 to use new SList.h (std::list compatible)
---------------------------------------------------------- */
#include "SList.h"
//#include "ListStd1.h"
#include <string>
//#include <list>
#include <iostream>

using namespace std;

typedef List<double> ListD;
typedef List<int> ListI;
typedef List<string> ListS;

int main(){
  ListD x;
  x.push_front(4.4);  x.push_front(3.3);  x.push_front(2.2);  x.push_front(1.1);

  ListD y(x);
  ListD z = x; //NB. equiv. to ListD z(x); see prev. line

  cout<< "x.front = "<< x.front()<< endl;

  //note that the following destroys the List
  cout<< "List x ="<<endl;
  cout<< "x.size() ="<< x.size()<< endl;
  while(!x.empty()){
    cout<< x.front()<< endl;
    x.pop_front();
  }
  cout<< "x.size() now = "<< x.size()<< endl;

  cout<< "List y ="<<endl;
  cout<< y<< endl;

  cout<< "List z ="<<endl;
  cout<< z<< endl;
  // continued ...
```

Figure 13.7: ListT1.cpp, part 1

```
  ListD v;  v = y;
  v.pop_front();
  cout<< "List v (v = y; v.pop_front();) ="<<endl;
  cout<< v<< endl;

  ListI li; li.push_front(3); li.push_front(2); li.push_front(1);
  cout<< "List li via operator <<"<<endl;
  cout<< li<< endl;

  li.push_back(22);  li.push_back(33);

  cout<< "li.push_back(22), li.push_back(33)"<< endl;
  cout<< li<< endl;

  cout<< "back(), pop.back()"<< endl;
  while(!li.empty()){
    cout<< li.back()<< endl;
    li.pop_back();
  }

  ListS ls;
  ls.push_front("abcd");
  ls.push_front("cdefgh");
  ls.push_back("back");
  cout<< ls<< endl;

  return 0;
}
```

Figure 13.8: ListT1.cpp, part 2

```
x.front = 1.1
List x =
x.size() =4
1.1
2.2
3.3
4.4
x.size() now = 0
List y =
f[ 1.1, 2.2, 3.3, 4.4 ]b

List z =
f[ 1.1, 2.2, 3.3, 4.4 ]b

List v (v = y; v.pop_front();) =
f[ 2.2, 3.3, 4.4 ]b

List li via operator <<
f[ 1, 2, 3 ]b

li.push_back(22), li.push_back(33)
f[ 1, 2, 3, 22, 33 ]b

back(), pop.back()
33
22
3
2
1
f[ cdefgh, abcd, back ]b
```

Figure 13.9: Output of ListT1.cpp.

13.4 Doubly Linked List

In this section, we develop a *doubly linked list*. The declaration is shown in Figures 13.10, 13.11 (Link) and 13.12 (ListIterator).

```
#ifndef LISTTH
#define LISTTH

#include <cassert>
#include <iostream>

template <class T>
class List;

template <class T>
std::ostream& operator<< (std::ostream& os, const List<T>& l);

template <class T>
class Link;

template <class T>
class ListIterator;

template <class T> class List{
  friend  std::ostream& operator<< <T>(std::ostream& os, const List<T>& l);
public:
  typedef ListIterator<T> Iterator;
  List() : first_(0), last_(0) {};
  List(const List<T> & other);
  ~List();
  List & operator = (const List<T> & rhs);
  T& front() const;
  void push_front(const T& e);
  void pop_front();
  T& back() const;
  void push_back(const T& e);
  void pop_back();
  void clear(); // empty the list and delete all elements in it
  bool empty() const;
  int size() const;
  Iterator begin();
  Iterator end();
  Iterator insert(Iterator& itr, const T& val);
  void insert(Iterator& itr, int n, const T& val);
  void erase(Iterator& itr);
  void erase(Iterator& start, Iterator& stop);
private:
  void copy(const List<T> & other); // private utility only, users use =
  Link<T>* first_;
  Link<T>* last_;
};
```

Figure 13.10: Doubly linked list declaration, part 1

```
template <class T>class Link{
  friend class List<T>;
  friend class ListIterator<T>;
  friend  std::ostream& operator<< <T>(std::ostream& os, const List<T>& l);
private:
  Link(const T& e) : elem_(e), next_(0), prev_(0){}
  T elem_;
  Link<T>* next_;
  Link<T>* prev_;
};
```

Figure 13.11: Doubly linked list declaration, part 2, the Link

```
template <class T>class ListIterator{
  friend class List<T>;
  typedef ListIterator<T> Iterator;
public:
  ListIterator(List<T>* list = 0, Link<T>* cLink = 0) :
    list_(list), cLink_(cLink) {}

  T& operator *(){
    return cLink_->elem_;
  }

  bool operator == (Iterator rhs){
    return cLink_ == rhs.cLink_;
  }

  bool operator != (Iterator rhs){
    return cLink_ != rhs.cLink_;
  }

  Iterator& operator ++ (int){
    cLink_ = cLink_->next_;
    return *this;
  }

  Iterator operator ++ ();

  Iterator& operator -- (int){
    cLink_ = cLink_->prev_;
    return *this;
  }

  Iterator operator -- ();

private:
  List<T>* list_;
  Link<T>* cLink_;
};
```

Figure 13.12: Doubly linked list declaration, part 3, the ListIterator

13.4.1 Brief Discussion of the Doubly Linked List

The doubly linked list in Figures 13.10 to 13.12 is very similar to the singly linked list described earlier in the chapter. The chief differences are:

1. There are now two `List` data members (the singly linked list had just a pointer to the front; now we have pointers to *front* and *back*:

   ```
   Link<T>* first_;
   Link<T>* last_;
   ```

   This means that when we want to `push_back` or `pop_back`, we can go directly there, via `last_`, rather than having to sequentially *walk* there as in the singly linked list example.

2. `Link` now has three data members, first the element; then, as before, one pointing to the *next* link, and now a new pointer pointing to the *previous* link.

   ```
   T elem_;
   Link<T>* next_;
   Link<T>* prev_;
   ```

   In the singly linked list, we could traverse the list (*iterate*) only in the *front towards back* direction via `next_`; now we can traverse in both directions, *back towards front*, using the `prev_` pointer.

3. The other major difference is that we have equipped the List with an *iterator*. This iterator has the same effect as the Array iterator in Chapter 12, i.e. it looks the same to client programs, but it is slightly more complicated to implement.

4. `ListIterator` has two data members:

   ```
   List<T>* list_;
   Link<T>* cLink_;
   ```

   a pointer to the `List` itself, and a pointer one of the `Link`s.

13.4.2 Simple Test Program, ListT1.cpp

Figures 13.13 and 13.14 show a simple test program. The only difference between this program and the one above in Figures 13.7 and 13.8 is that we have added code to exercise the additional functions in the doubly linked list, and the iterator.

As pointed out before, we note that this doubly linked list, the singly linked list above, and `std:list` appear identical in client programs; the only differences (and there should be none) is that in the lists here, we have chosen to implement only a subset of the functions of `std::list` and in the singly linked list, the subset is even smaller. As we keep saying, the objectives of the list classes here is not to replace `std::list`, but to get some feeling how `std::list` might be implemented.

The output of `ListT1.cpp` is shown in Figure 13.15

```
/* --- ListT1.cpp -------------------------------------------
 j.g.c. 31/12/96//j.g.c. 1/1/97, 5/1/97, 2007-12-28, 2008-01-18
 changed j.g.c. 2007-12-30 to use new List.h (std::list compatible)
---------------------------------------------------------- */
#include "List.h"
//#include "ListStd1.h"
#include <string>
//#include <list>
#include <iostream>

using namespace std;

typedef List<double> ListD;
typedef List<int> ListI;
typedef List<string> ListS;

int main(){
  ListD x;
  x.push_front(4.4);  x.push_front(3.3);  x.push_front(2.2);  x.push_front(1.1);

  ListD y(x);
  ListD z = x; //NB. equiv. to ListD z(x); see prev. line

  cout<< "x.front = "<< x.front()<< endl;

  //note that the following destroys the list
  cout<< "List x ="<<endl;
  cout<< "x.size() ="<< x.size()<< endl;
  while(!x.empty()){
    cout<< x.front()<< endl;
    x.pop_front();
  }
  cout<< "x.size() now = "<< x.size()<< endl;

  cout<< "List y ="<<endl;
  cout<< y<< endl;

  cout<< "List z ="<<endl;
  cout<< z<< endl;

  ListD v;
  v = y;
  v.pop_front();
  cout<< "List v (v = y; v.pop_front();) ="<<endl;
  cout<< v<< endl;   // continued ...
```

Figure 13.13: Simple Test Program for Doubly Linked List, ListT1.cpp, part 1

```
ListI li; li.push_front(3); li.push_front(2); li.push_front(1);
cout<< "List li via operator <<"<<endl;
cout<< li<< endl;

li.push_back(22);
li.push_back(33);

cout<< "li.push_back(22), li.push_back(33)"<< endl;
cout<< li<< endl;

cout<< "back(), pop.back()"<< endl;
while(!li.empty()){
  cout<< li.back()<< endl;
  li.pop_back();
}

ListS ls;
ls.push_front("abcd");
ls.push_front("cdefgh");
ls.push_back("back");
cout<< ls<< endl;

ListI c5;
for(uint i = 0; i< 5; ++i){
  c5.push_back(i);
  cout<< "c5.push_back(i = "<< i<< "): "<< c5;
}

cout<< "using Iterator"<< endl;
ListI::Iterator itr = c5.begin();
ListI::Iterator itrb = c5.begin();
ListI::Iterator itre = c5.end();

if(itr == itrb)cout<< "itr == itrb"<< endl;
else cout<< "itr != itrb"<< endl;

if(itr != itrb)cout<< "itr != itrb"<< endl;
else cout<< "itr == itrb"<< endl;

ListI::Iterator it;
for(it = c5.begin(); it != c5.end(); ++it){
  cout<< *it<< ' ';
}

cout<< "ListI::Iterator itr2 = c5.begin(), ++, ++ "<< endl;
cout<< "c5.insert(itr2, 5, 33) "<< endl;
ListI::Iterator itr2 = c5.begin();
itr2++; itr2++;
c5.insert(itr2, 5, 33);
cout<< c5;
```


Figure 13.14: Simple Test Program for Doubly Linked List, ListT1.cpp, part 2

```
x.front = 1.1
List x =
x.size() =4
1.1
2.2
3.3
4.4
x.size() now = 0
List y =
f[ 1.1, 2.2, 3.3, 4.4 ]b

List z =
f[ 1.1, 2.2, 3.3, 4.4 ]b

List v (v = y; v.pop_front();) =
f[ 2.2, 3.3, 4.4 ]b

List li via operator <<
f[ 1, 2, 3 ]b

li.push_back(22), li.push_back(33)
f[ 1, 2, 3, 22, 33 ]b

back(), pop.back()
33
22
3
2
1
f[ cdefgh, abcd, back ]b

c5.push_back(i = 0): f[ 0 ]b
c5.push_back(i = 1): f[ 0, 1 ]b
c5.push_back(i = 2): f[ 0, 1, 2 ]b
c5.push_back(i = 3): f[ 0, 1, 2, 3 ]b
c5.push_back(i = 4): f[ 0, 1, 2, 3, 4 ]b
using Iterator
itr == itrb
itr == itrb
0 1 2 3 4 ListI::Iterator itr2 = c5.begin(), ++, ++
c5.insert(itr2, 5, 33)
f[ 0, 1, 33, 33, 33, 33, 33, 2, 3, 4 ]b
```

Figure 13.15: Output of ListT1.cpp (double linked version)

13.4.3 Doubly Linked List Implementation

Here we give the complete implementation of the doubly linked list. Since the principles involved are the same as for the singly linked list, we provide no discussion. However, it will be worthwhile to spend some time in class on the implementation, especially that of `ListIterator`.

```
// --------- ListIterator ---------------------------------
// postfix form of increment
template <class T>
ListIterator<T> ListIterator<T>::operator ++ (){
  // clone, then increment, return clone
  ListIterator<T> clone (list_, cLink_);
  cLink_ = cLink_->next_;
  return clone;
}

// postfix form of decrement
template <class T>
ListIterator<T> ListIterator<T>::operator -- (){
  // clone, then increment, return clone
  ListIterator<T> clone (list_, cLink_);
  cLink_ = cLink_->prev_;
  return clone;
}
```

Figure 13.16: Doubly Linked List Implementation, part 1, ListIterator

```
template <class T>
void List<T>::copy(const List<T> & other){
  if(other.empty())first_ = 0;
  else{
    Link<T> *pp = other.first_; //cursor to other
    Link<T> *pt = new Link<T>(pp->elem_);
    first_ = pt;
    while(pp->next_ != 0){
      pp = pp->next_;
      pt->next_ = new Link<T>(pp->elem_);
      pt = pt->next_;
    }
  }
}

template <class T>
List<T>::List(const List<T> & other){
  copy(other);
}

template <class T>
List<T> & List<T>::operator = (const List<T> & rhs){
  if(this != &rhs){ //beware of listA=listA;
    clear();
  }
  copy(rhs);
  return *this;
}

template <class T>
List<T>::~List(){
  clear();
}
```

Figure 13.17: Doubly Linked List Implementation, part 2

```
template <class T>
void List<T>::copy(const List<T> & other){
  if(other.empty())first_ = 0;
  else{
    Link<T> *pp = other.first_; //cursor to other
    Link<T> *pt = new Link<T>(pp->elem_);
    first_ = pt;
    while(pp->next_ != 0){
      pp = pp->next_;
      pt->next_ = new Link<T>(pp->elem_);
      pt = pt->next_;
    }
  }
}

template <class T>
List<T>::List(const List<T> & other){
  copy(other);
}

template <class T>
List<T> & List<T>::operator = (const List<T> & rhs){
  if(this != &rhs){ //beware of listA=listA;
    clear();
  }
  copy(rhs);
  return *this;
}

template <class T>
List<T>::~List(){
  clear();
}
```

Figure 13.18: Doubly Linked List Implementation, part 3

```
template <class T>
void List<T>::push_front(const T& e){
  Link<T> *newLink= new Link<T>(e);
  assert(newLink != 0);
  if(empty())first_ = last_ = newLink;
  else {
    first_->prev_ = newLink;
    newLink->next_ = first_;
    first_= newLink;
  }
}

template <class T>
T& List<T>::front() const{
  assert(!empty());
  return first_->elem_;
}

template <class T>
void List<T>::pop_front(){
  assert(!empty());
  Link<T>* tmp = first_;
  first_= first_->next_;
  if(first_!= 0) first_->prev_ = 0;
  else last_ = 0;
  delete tmp;
}
```

Figure 13.19: Doubly Linked List Implementation, part 4

```
template <class T>
void List<T>::push_back(const T& e){
  Link<T> *newLink= new Link<T>(e);  assert(newLink != 0);
  if(empty()) first_ = last_ = newLink;
  else{
    last_->next_ = newLink;
    newLink->prev_ = last_;
    last_ = newLink;
  }
}

template <class T>
void List<T>::pop_back(){
  assert(!empty());
  Link<T>* tmp = last_;
  last_= last_->prev_;
  if(last_!= 0) last_->next_ = 0;
  else first_ = 0;
  delete tmp;

}

template <class T>
T& List<T>::back() const{
  assert(!empty());
  return last_->elem_;
}

template <class T>
void List<T>::clear(){
  Link<T> *next, *first(first_);
  while(first != 0){
    next = first->next_;    delete first;    first = next;
  }
  first_ = 0;
}

template <class T>
bool List<T>::empty() const{
  return (first_ == 0);
}

template <class T>
int List<T>::size() const{
  int i = 0;
  Link<T> *pt = first_;
  while(pt != 0){
    pt = pt->next_;    ++i;
  }
  return i;  }
```


Figure 13.20: Doubly Linked List Implementation, part 5

```
template <class T>
ListIterator<T> List<T>::begin() {
  return Iterator(this, first_);
}

template <class T>
ListIterator<T> List<T>::end() {
  return ListIterator<T>(this, 0);
}

template <class T>
void List<T>::erase(ListIterator<T>& itr){
  erase(itr, itr);
}

// insert a new element into the middle of a linked list
template <class T>
ListIterator<T> List<T>::insert (ListIterator<T> & itr, const T& value){
  Link<T>* newLink = new Link<T>(value);
  Link<T>* current = itr.cLink_;

  newLink->next_ = current;
  newLink->prev_ = current->prev_;
  current->prev_ = newLink;
  current = newLink->prev_;
  if (current != 0)current->next_ = newLink;

  return ListIterator<T>(this, newLink);
}

// insert n new elements into the middle of a linked list
// note: iterator changed, but I think this is okay as
// iterators are known to be invalidated by insertion
template <class T>
void List<T>::insert (ListIterator<T> & itr, int n, const T& value){
  for(int i = 0; i< n; i++){
    itr = insert(itr, value);
  }
}
```

Figure 13.21: Doubly Linked List Implementation, part 6

```
// remove values from the range of elements
template <class T>
void List<T>::erase (ListIterator<T> & start, ListIterator<T> & stop){
  Link<T> * first = start.cLink_;
  Link<T> * prev = first->prev_;
  Link<T> * last = stop.cLink_;
  if (prev == 0) {  // removing initial portion of list
    first_ = last;
    if (last == 0) last_ = 0;
    else last->prev_ = 0;
  }
  else {
    prev->next_ = last;
    if (last == 0)last_ = prev;
    else last->prev_ = prev;
  }
  // now delete the values
  while (start != stop) {
    ListIterator<T> next = start;
    ++next;
    delete start.cLink_;
    start = next;
  }
}

template <class T>
std::ostream& operator<< (std::ostream& os, const List<T>& lst){
  os<<"f[ ";
  Link<T> *pp = lst.first_; //cursor to lst
  while(pp != 0){
    if(pp != lst.first_)os<<", ";
    os<< pp->elem_;
    pp = pp->next_;
  }
  os<<" ]b"<<std::endl;
  return os;
}
#endif
```

Figure 13.22: Doubly Linked List Implementation, part 7

13.5 Arrays versus Linked List, Memory Usage

On a 32-bit machine, `int` and *pointer* typed variables normally occupy four 8-bit bytes.

Ex. 1. If we have an `Array<int>` such as that in Chapter 12 (and `std::vector` will be the same) whose size and capacity are 1000, how many memory bytes will be used?

Ex. 2. Do the same calculation for a singly linked list which contains 1000 elements.

Ex. 3. Do the same calculation for a doubly linked list.

Ex. 4. If we define *efficiency* as *actual useful data memory* divided by *total memory used*, what can we say about the *efficiency* obtained in **Ex. 1.**, **Ex. 2.** and **Ex. 3.** above.

Ex. 5. As N, the number of elements, increases to a very large number, what can we say about the efficiency in the three cases: (i) array, (ii) singly linked list, (iii) doubly linked list.

13.6 Arrays versus Linked List, Cache issues

All CPU chips these days have *cache memory*; *cache memory* is extra-fast memory close to the CPU. Cache memory has access times one tenth to one twentieth of main memory.

Usually, there are two separate caches, *data cache*, and *instruction cache*.

In the case of data cache, when you access a memory location, 4560, say, the cache system will bring memory locations 4560 to 4581 (32 bytes — it could be more, depends on the CPU) into cache; the first memory access will be at the speed of main memory; however, if you access memory 4564, it will already be in cache and this memory access will be at the much faster cache speed.

Hence on machines that have cache, it makes sense to access memory in a orderly manner: e.g. 4560, 4564, 4568, ...

If you hop about through memory: e.g. 4560, 200057, 26890, . . . , you will lose the speed of the cache memory.

Ex. In connection with cache memory, what performance penalty might a program incur when using a linked list instead of an array?

Chapter 14

Case Study — Person, Student class hierarchy

14.1 Introduction

This is a case study involving a number of classes; it is useful to see how a methodical object-oriented design approach can tame a problem that looks very difficult at first glance. A secondary goal of the case study is to gain confidence in using the *standard library*.

Unfortunately, the software that is presented here is the result of a few iterations of design; thus, like software that you see in some books, it is hard to depict the thought process that resulted in the what you see here.

The mini-project started off with the requirement for a set of classes related to Institute staff and students; hence we think immediately of *Student* and *Lecturer*. It is relatively obvious that these have a lot in common; the common stuff can be factored into a *Person* class.

A *Person* will have set of names and an address. After a bit of thought, it becomes obvious that a separate *Address* part (class) is sensible; the latter for two reasons: (i) it cuts down the size and complexity of the *Person* class; (ii) an *Address* class will be useful if we ever need to work with companies and other entities that need an address.

The remainder of this chapter presents the classes and test programs.

The assignment questions at the end of the chapter helps identify some of the key learning objectives of this chapter.

The commentary here is kept to a minimum — this software will be the subject of many lectures and discussions, plus an assignment.

```
// ---------------- Address.h ---------------------------
// j.g.c version 1.1; 2007-08-26, 2007-11-04
// ------------------------------------------------------
#ifndef ADDRESS_H
#define ADDRESS_H

#include <string>
#include "StringTokenizer.h"

class Address{
  friend bool operator<(const Address& lhs, const Address& rhs);
  friend bool operator==(const Address& lhs, const Address& rhs);
private:
  std::string num_;
  std::string street_; // street = first line of address
  std::string city_;
  std::string region_; // county or state
  std::string country_;

public:
  Address(std::string& num, std::string& street, std::string& city,
          std::string& region, std::string& country);
  // below = "" does for default constructor
  // which C++ wants but will never be used
  Address(std::string str = "blank,blank,blank,blank,blank");
  std::string toString() const;
};
#endif
```

```
// ---------------- Address.cpp --------------------------
// j.g.c version 1.1; 2007-08-26, 2007-11-04
// -------------------------------------------------
#include "Address.h"

Address::Address(std::string& num, std::string& street,
                 std::string& city,
                 std::string& region, std::string& country)
          : num_(num), street_(street), city_(city),
            region_(region), country_(country){}

Address::Address(std::string str){
    StringTokenizer st = StringTokenizer(str, ",");
    num_ = st.token(0);
    street_ = st.token(1); city_ = st.token(2);
    region_ = st.token(3);  country_ = st.token(4);
  }

std::string Address::toString() const{
    return num_ + "," + street_ + "," +  city_ + ","
           + region_ + "," + country_;
}

bool operator<(const Address& lhs, const Address& rhs){
  if(lhs.country_==rhs.country_){
    if(lhs.region_==rhs.region_){
      if(lhs.city_==rhs.city_){
        if(lhs.street_==rhs.street_){
          return lhs.num_< rhs.num_;
        }
        else return lhs.street_< rhs.street_;
      }
      else return lhs.city_< rhs.city_;
    }
    else return lhs.region_< rhs.region_;
  }
  else return lhs.country_< rhs.country_;
}

bool operator==(const Address& lhs, const Address& rhs){
  return lhs.country_==rhs.country_ &&
         lhs.region_==rhs.region_   &&
         lhs.city_==rhs.city_       &&
         lhs.street_==rhs.street_   &&
         lhs.num_== rhs.num_;
}
```

```
// ----------------- AddressT1.cpp ------------------
// j.g.c. 2007-08-27, 2007-10-04
// --------------------------------------------------
#include "Address.h"
#include <vector>
#include <list>
#include <iostream>
#include <iterator>
#include <cstdlib> // for rand
#include <sstream> // for ostringstream
#include <algorithm> // for sort
using namespace std;

class AddressT1 {
public:
  AddressT1(){

    vector<string> streets = vector<string>();
    streets.push_back(string("Port Road"));
    streets.push_back(string("Pearse Road"));
    streets.push_back(string("College Terrace"));
    streets.push_back(string("Kingsbury"));
    streets.push_back(string("New Street"));

    vector<string> cities = vector<string>();
    cities.push_back(string("Newtown"));
    cities.push_back(string("Oldtown"));
    cities.push_back(string("Johnstown"));
    cities.push_back(string("Hightown"));
    cities.push_back(string("Lowtown"));

    vector<string> regions = vector<string>();
    regions.push_back(string("Oldshire"));
    regions.push_back(string("Newshire"));
    regions.push_back(string("Green County"));
    regions.push_back(string("Old County"));

    vector<string> countries = vector<string>();
    countries.push_back(string("Ireland"));
    countries.push_back(string("Wales"));
    countries.push_back(string("England"));
    countries.push_back(string("Scotland"));

    vector<Address> addList = vector<Address>();
    string num, street, city, region, country;
    unsigned int rnseed= 139139;
    srand(rnseed); // initialise random number generator.
    int n = 10;
    for(int i = 0; i< n; ++i){
```

```
      int j = rand()%streets.size();
      street = streets.at(j);
      j = rand()%100;
      ostringstream ossnum;
      ossnum<< j;
      num = ossnum.str();
      j = rand()%cities.size();
      city = cities.at(j);
      j = rand()%regions.size();
      region = regions.at(j);
      j = rand()%countries.size();
      country = countries.at(j);
      Address a = Address(num, street, city, region, country);
      addList.push_back(a);
    }
    string s= string("Rose Cottage,Lough Salt,Kilmacrennan,Donegal,Ireland");
    Address a = Address(s );
    addList.push_back(a);

    cout<<"The address list size is: "<< addList.size()<< endl;
    cout<< "The address list is:"<< endl;
    for(int i = 0; i!= addList.size(); ++i){
      cout<< addList.at(i).toString()<< endl;
    }
    sort(addList.begin(), addList.end());
    cout<<"Sorted address list size is: "<< addList.size()<< endl;
    cout<< "Sorted address list is:"<< endl;
    for(int i = 0; i!= addList.size(); ++i){
      cout<< addList.at(i).toString()<< endl;
    }
  }
};
  /** The main method constructs the test */
int main(){
  AddressT1();
}
```

```
// ---------------- Person.h ---------------------------
// j.g.c version 1.0; 2007-08-27, 2007-11-04
// -----------------------------------------------------
#ifndef PERSON_H
#define PERSON_H

#include <string>
#include "StringTokenizer.h"
#include "Address.h"

class Person{
  friend bool operator<(const Person& lhs, const Person& rhs);
  friend bool operator==(const Person& lhs, const Person& rhs);
private:
  std::string firstName_;
  std::string lastName_;
  Address address_;

public:
  Person(std::string& firstName, std::string& lastName,
         Address& address);
  // see Address.h for comment on = "..."
  Person(std::string str =
    "blankf,blankl;blanknum,blanks,blankc,blankr,blankc");
  virtual std::string toString() const;
  std::string getFirstName() const;
  std::string getLastName() const;
};
#endif
```

```
// ---------------- Person.cpp --------------------------
// j.g.c version 1.0; 2007-08-27, 2007-11-04
// ------------------------------------------------------
#include "Person.h"
#include <iostream>

Person::Person(std::string& firstName, std::string& lastName,
               Address& address)
          : firstName_(firstName), lastName_(lastName),
            address_(address){}

Person::Person(std::string str){
  // first split into name ; address
  StringTokenizer st1 = StringTokenizer(str, ";");
  std::string sName = st1.token(0);
  std::string sAddress = st1.token(1);

  address_ = Address(sAddress);
  // now split name
  StringTokenizer st2 = StringTokenizer(sName, ",");
    firstName_ = st2.token(0);  lastName_ = st2.token(1);
  }

std::string Person::toString() const{
    return firstName_ + "," +  lastName_ + ";" + address_.toString();
}

std::string Person::getFirstName() const{
  return firstName_;
}

std::string Person::getLastName() const{
  return lastName_;}

bool operator<(const Person& lhs, const Person& rhs){
  if(lhs.lastName_==rhs.lastName_){
    if(lhs.firstName_==rhs.firstName_){
      return lhs.address_< rhs.address_;
    }
    else return lhs.firstName_< rhs.firstName_;
  }
  else return lhs.lastName_< rhs.lastName_;
}

bool operator==(const Person& lhs, const Person& rhs){
  return lhs.firstName_==rhs.firstName_ &&
         lhs.lastName_==rhs.lastName_ &&
         lhs.address_ == rhs.address_;}
```

```
// ----------------- PersonT1.cpp ------------------
// j.g.c. 2007-08-27, 2007-11-04
// -------------------------------------------------
#include "Person.h"
#include <vector>
#include <list>
#include <iostream>
#include <iterator>
#include <cstdlib> // for rand
#include <sstream> // for ostringstream
#include <algorithm> // for sort

using namespace std;

class PersonT1 {
public:
  PersonT1(){

    vector<string> firstNames = vector<string>();
    firstNames.push_back(string("Seamus"));
    firstNames.push_back(string("Sean"));
    firstNames.push_back(string("Sharon"));
    firstNames.push_back(string("Sinead"));
    firstNames.push_back(string("Sarah"));
    firstNames.push_back(string("Simon"));

    vector<string> lastNames = vector<string>();
    lastNames.push_back(string("Bloggs"));
    lastNames.push_back(string("Burke"));
    lastNames.push_back(string("Blaney"));
    lastNames.push_back(string("Bradley"));
    lastNames.push_back(string("Brown"));
    lastNames.push_back(string("Black"));

    vector<string> streets = vector<string>();
    streets.push_back(string("Port Road"));
    streets.push_back(string("Pearse Road"));
    streets.push_back(string("College Terrace"));
    streets.push_back(string("Kingsbury"));
    streets.push_back(string("New Street"));

    vector<string> cities = vector<string>();
    cities.push_back(string("Newtown"));
    cities.push_back(string("Oldtown"));
    cities.push_back(string("Johnstown"));
    cities.push_back(string("Hightown"));
    cities.push_back(string("Lowtown"));

    vector<string> regions = vector<string>();
```

```
regions.push_back(string("Oldshire"));
regions.push_back(string("Newshire"));
regions.push_back(string("Green County"));
regions.push_back(string("Old County"));

vector<string> countries = vector<string>();
countries.push_back(string("Ireland"));
countries.push_back(string("Wales"));
countries.push_back(string("England"));
countries.push_back(string("Scotland"));

vector<Person> personList = vector<Person>();
string num, street, city, region, country, lastName, firstName;
unsigned int rnseed= 139;
srand(rnseed); // initialise random number generator.
int n = 10, j;
for(int i = 0; i< n; ++i){
  j = rand()%100;
  ostringstream ossnum;
  ossnum<< j;
  num = ossnum.str();

  cout<< "i = "<< i<< endl;
  j = rand()%streets.size();
  street = streets.at(j);
  j = rand()%cities.size();
  city = cities.at(j);
  j = rand()%regions.size();
  region = regions.at(j);
  j = rand()%countries.size();
  country = countries.at(j);

  j = rand()%firstNames.size();
  firstName = firstNames.at(j);
  j = rand()%lastNames.size();
  lastName = lastNames.at(j);

  Address a = Address(num, street, city, region, country);
  cout<< a.toString()<< endl;
  Person p = Person(firstName, lastName, a);
  cout<< p.toString()<< endl;
  personList.push_back(p);
}
cout<< "testing Person(std::string)"<< endl;
string s = string("James,Mulligan;Hillview,Lough Salt,Kilmacrennan,Donegal,Irela
Person p(s);
personList.push_back(p);
p = Person();
personList.push_back(p);
```

```
    cout<<"The Person list size is: "<< personList.size()<< endl;
    cout<< "The Person list is:"<< endl;
    for(int i = 0; i!= personList.size(); ++i){
      cout<< personList.at(i).toString()<< endl;
    }

    sort(personList.begin(), personList.end());
    cout<<"Sorted address list size is: "<< personList.size()<< endl;
    cout<< "Sorted address list is:"<< endl;
    for(int i = 0; i!= personList.size(); ++i){
      cout<< personList.at(i).toString()<< endl;
    }
    cout<< "Sorted names are:"<< endl;
    for(int i = 0; i!= personList.size(); ++i){
      cout<< personList.at(i).getLastName()<< ", "
          << personList.at(i).getFirstName()<< endl;
    }
  }
};
  /** The main method constructs the test */
int main(){
  PersonT1();
}
```

```
// ---------------- Student.h ---------------------------
// j.g.c version 1.1; 2007-08-27, 2007-11-04
// ------------------------------------------------------
#ifndef STUDENT_H
#define STUDENT_H

#include <string>
#include "Person.h"

class Student: public Person{
  friend bool operator<(const Student& lhs, const Student& rhs);
  friend bool operator==(const Student& lhs, const Student& rhs);
private:
  std::string id_;

public:
  Student(std::string& firstName, std::string& lastName,
          Address& address, std::string& id);
  // see Address.h for comment on = "..."
  Student(std::string str =
    "blankf,blankl;blanknum,blanks,blankc,blankr,blankc;blankid");
  std::string toString() const;
  string getId() const;
};
#endif
```

```
// ---------------- Student.cpp ---------------------------
// j.g.c version 1.1; 2007-08-27, 2007-11-04
// --------------------------------------------------
#include "Student.h"
#include <iostream>

Student::Student(std::string& firstName, std::string& lastName,
                 Address& address, std::string& id)
          : Person(firstName, lastName, address), id_(id){}

Student::Student(std::string str) : Person(str){
  // first split into name ; address ; id
  // note that Person(str) will ignore id
  StringTokenizer st1 = StringTokenizer(str, ";");
  id_ =  st1.token(2); // (0) is name, (1) is address, (2) is id
}

std::string Student::toString() const{
  return Person::toString() + ";" + id_;
}

std::string Student::getId() const{
  return id_;
}

bool operator<(const Student& lhs, const Student& rhs){
  Person pl = Person(lhs); Person pr = Person(rhs);
  if(pl == pr){
    return lhs.id_< rhs.id_;
  }
  else return pl< pr;
}

bool operator==(const Student& lhs, const Student& rhs){
  Person pl = Person(lhs); Person pr = Person(rhs);
  return pl == pr && lhs.id_ == rhs.id_;
}
```

```
// ----------------- StudentT1.cpp ------------------
// j.g.c. 2007-08-27, 2007-11-04
// --------------------------------------------------
#include "Student.h"
#include <vector>
#include <list>
#include <iostream>
#include <iterator>
#include <cstdlib> // for rand
#include <sstream> // for ostringstream
#include <algorithm> // for sort
#include <fstream>  // for files - ofstream, ifstream

using namespace std;

bool cmp0(const Student& s1, const Student& s2){
  return s1.getId()< s2.getId();
}

class StudentT1 {
public:
  StudentT1(){

    vector<string> firstNames = vector<string>();
    firstNames.push_back(string("Seamus"));
    // etc. same as PersonT1.cpp
    vector<Student> sList = vector<Student>();
    string num, street, city, region, country, lastName, firstName;
    unsigned int rnseed= 139;
    srand(rnseed); // initialise random number generator.
    int n = 10, j;
    string id1("L200702");
    for(int i = 0; i< n; ++i){
      j = rand()%100;
      ostringstream ossnum;
      ossnum<< j;
      num = ossnum.str();

      cout<< "i = "<< i<< endl;
      j = rand()%streets.size();
      street = streets.at(j);
      j = rand()%cities.size();
      city = cities.at(j);
      j = rand()%regions.size();
      region = regions.at(j);
      j = rand()%countries.size();
      country = countries.at(j);

      j = rand()%firstNames.size();
```

```
  firstName = firstNames.at(j);
  j = rand()%lastNames.size();
  lastName = lastNames.at(j);

  Address a = Address(num, street, city, region, country);
  cout<< a.toString()<< endl;
  int k = i + 10;
  ostringstream ossid;
  ossid<< id1<< k;
  string id = ossid.str();
  Student st = Student(firstName, lastName, a, id);
  sList.push_back(st);
}
string s =
  string("James,Mulligan;Hillview,Lough Salt,Kilmacrennan,Donegal,Ireland;L20079
Student st(s);
sList.push_back(st);
s = string(
  "James,Mulligan;Hillview,Lough Salt,Kilmacrennan,Donegal,Ireland;L200789");
Student st1(s);
sList.push_back(st1);
st = Student();
sList.push_back(st);
cout<<"The Student list size is: "<< sList.size()<< endl;
cout<< "The Student list is:"<< endl;
for(int i = 0; i!= sList.size(); ++i){
  cout<< sList.at(i).toString()<< endl;
}

sort(sList.begin(), sList.end());
cout<<"Sorted list size is: "<< sList.size()<< endl;
cout<< "Sorted list is:"<< endl;
for(int i = 0; i!= sList.size(); ++i){
  cout<< sList.at(i).toString()<< endl;
}
// now using alternative compare
sort(sList.begin(), sList.end(), cmp0);
cout<< "Student list sorted by student id. is:"<< endl;
for(int i = 0; i!= sList.size(); ++i){
  cout<< sList.at(i).toString()<< endl;
}

vector<Person*> ppList = vector<Person*>();
Person *pp;
for(int i = 0; i!= sList.size(); ++i){
  pp = &sList.at(i);
  ppList.push_back(pp);
}
```

```
    string s1 = string(
      "Jim,Dillon;10,Fourth Avenue,Raphoe,Donegal,Ireland;L200791");
    Person p(s1);
    ppList.push_back(&p);

    cout<<"The Person pointer list size is: "<< sList.size()<< endl;
    cout<< "The Person pointer (contents) list is:"<< endl;

    for(int i = 0; i!= ppList.size(); ++i){
      cout<< ppList.at(i)->toString()<< endl;
    }

    cout<< "\nSave that vector to a file\n"<< endl;
    ofstream fileout("students.dat");
    if(!fileout){
      cerr<< "cannot open file <students.dat> for writing\n";
      exit(-1);
    }
    for(unsigned int i=0; i< sList.size(); i++){
      fileout<< sList[i].toString();
      fileout<< endl;
    }
    fileout.close();

    cout<< "\n... and read some of them back\n"<< endl;
    vector<Student> v4;

    ifstream filein("students.dat");
    if(!filein){
      cerr<< "cannot open file <students.dat> for reading\n";
      exit(-1);
    }
    string str1;
    for(unsigned int i=0; i< 5; i++){
      getline(filein, str1);
      v4.push_back(Student(str1));
    }
    filein.close();

    for(unsigned int i = 0; i< v4.size(); i++){
      cout<< v4[i].toString()<< endl;
    }
  }
};
  /** The main method constructs the test */
int main(){
  StudentT1();
}
```

Assignment 4, Deadline Mon. 19th November 2007, 3.30pm. Demonstrations must be completed by 19th Nov., at 3.00pm. Worth 20% of CA.

1. Compile, link and execute `AddressT1.cpp`.

   [10 marks]

   Demonstration 100% of marks.

2. Compile, link and execute `PersonT1.cpp`.

   [10 marks]

   Demonstration 100% of marks.

3. Compile, link and execute `StudentT1.cpp`.

   [10 marks]

   Demonstration 100% of marks.

4. Virtual functions.

   (a) Remove the `virtual` qualifier from `toString()` in `Person.h` and `Student.h`. Recompile, link and execute `StudentT1.cpp`.

   [10 marks]

   Demonstration 100% of marks.

   (b) There should be a discrepancy between the Person pointer (contents) list in (a) and the one in 3. above. Include a printout of the two lists and explain the difference.

   [10 marks]

5. Your own test program for `Student`.

   (a) Write your own test program `StudentT2.cpp` for `Student`. You should avoid my automated test data generation and use instructions like those below

```
string s =
  string("James,Mulligan;19, First Road,Kilmacrenan,Donegal,Ireland;L20
Student st(s);
sList.push_back(Student st(s));
```

   In addition to using the constructor

```
Student(std::string str =
"blankf,blankl;blanknum,blanks,blankc,blankr,blankc;blankid");
```

   you should asl show examples of use of the constructor

```
Student(std::string& firstName, std::string& lastName,
     Address& address, std::string& id);
```

   There should be at least eight (8) students in your list. Demonstration.

   [20 marks]

   (b) Commented printout of StudentT2.cpp in hand-in report.

   [20 marks]

   (c) Commented printout of lists before and after sorting in hand-in report.

   [20 marks]

6. (a) Copy `LecturerT1.cpp` to `LecturerT2.cpp` and use `vector` rather than `list`. Compile and execute. Demonstration.

   [20 marks]

   (b) Add code to `LecturerT2.cpp` to `sort` the vector and print out the sorted list. Demonstration.

   [20 marks]

   (c) Add code to `LecturerT2.cpp` to `sort` the vector according to *office number*; print out the sorted list. See `StudentT1` for a closely related example. Demonstration.

   [20 marks]

   (d) Commented printout of relevant parts of `LecturerT2.cpp` in the hand-in report.

   [20 marks]

   (e) Commented printout of the unsorted and sorted tables in hand-in report.

   [20 marks]

7. Now you are going to write an *address book* program.

   (a) First we need a class `Contact`; `Contact` inherits from `Person`, just like `Student` and `Lecturer`.

   `Contact` should have instance variables for email address and for telephone number. It should have == and < operators; the < operator should take account of email address if the other fields are equal.

   Include properly commented `Contact.h` and `Contact.cpp` in your hand-in report.

   [50 marks]

   (b) Write a test program `AddressBook.cpp` that tests `Contact` just like `StudentT1.cpp` or `StudentT2.cpp`. The test program must demonstrate:

   (o) Basic test program;

   (i) Sorting according to the == and < operators;

   (ii) Sorting according to a *compare* function that compares `Contact` entries based on their telephone number; see `cmp0` in `StudentT1.cpp` and its use there.

   (iii) Writing to file;

   (iv) Reading from file;

   (v) Reading from a file when the program starts; use command line arguments; i.e. `>>prog addlist.txt` reads `addlist.txt` into a vector.

   Demonstration

   [$6 \times 10 = 60$ marks]

   (c) Include a properly commented `AddressBook.cpp` in your hand-in report.

   [50 marks]

   (d) Include a UML diagram showing the relationship between `AddressBook`, `Contact`, `Person`, and `Address` your hand-in report.

   [20 marks]

Bibliography

Alexandrescu, A. (2001). *Modern C++ Design*, Addison Wesley.

Blanchette, J. & Summerfield, M. (2008). *C++ GUI Programming with Qt7*, 2nd edn, Prentice Hall. ISBN: 0-13-235416-0.

Bornat, R. (1987). *Programming from First Principles*, Prentice-Hall.

Budd, T. (1997a). *Data Structures in C++ Using the Standard Template Library*, Addison Wesley.

Budd, T. (1997b). *An Introduction to Object-oriented Programming*, 2nd edn, Addison Wesley.

Budd, T. (1999a). *C++ for Java Programmers*, Addison Wesley.

Budd, T. (1999b). *Understanding Object-oriented Programming with Java, (updated for Java 2)*, Addison-Wesley.

Campbell, J. (2009). Algorithms and Data Structures for Games Programmers, *Technical report*, Letterkenny Institute of Technology. URL. http://www.jgcampbell.com/adsgp/.

Cline, M., Lomow, G. & Girou, M. (1999a). *C++ FAQs 2nd ed.*, 2nd edn, Addison Wesley.

Cline, M., Lomow, G. & Girou, M. (1999b). *C++ FAQs 2nd ed.*, 2nd edn, Addison Wesley.

Czarnecki, K. & Eisenecker, U. (2000). *Generative Programming*, Addison Wesley.

Dawson, M. (2004). *Beginning Game Programming*, Thompson Course Technology. ISBN: 1-59200-206-6.

Dewhurst, S. C. (2003). *C++ Gotchas: Avoiding Common Problems in Coding and Design*, Addison Wesley.

Dewhurst, S. C. (2005). *C++ Common Knowledge : Essential Intermediate Programming*, Addison Wesley. ISBN: 0-321-32192-8.

Dickheiser, M. J. (2007). *C++ for Game Programmers*, 2nd edn, Charles River Media. ISBN: 1-58450-452-8. This is the second edition of a first edition by Llopis.

Eckel, B. (2000). *Thinking in C++, vol. 1*, Prentice Hall.

Eckel, B. (2003). *Thinking in C++, vol. 2*, Prentice Hall.

Fitzgerald, J. & Larsen, P. (1998). *Modelling Systems: Practical Tools and Techniques in Software Development*, Cambridge University Press.

Freeman, E. & Freeman, E. (2004). *Head First Design Patterns*, O'Reilly. ISBN: 0-596-00712-4.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley.

Glassborow, F. (2006). *You Can Program in C++*, Wiley. ISBN: 0-470-01468-7.

Harbison, S. & Steele, G. (2005). *C: a Reference Manual*, 5th edn, Prentice-Hall.

Horstmann, C. (2005). *Java Concepts*, 4th edn, Wiley.

Horstmann, C. (2006). *Object Oriented Design and Patterns*, 2nd edn, Wiley.

Horstmann, C. (accessed 2006-08-28b). Moving from java to c++. online: http://horstmann.com/ccj2/ccjapp3.html.

Horstmann, C. (accessed 5006-08-28a). C++ pitfalls. online: http://horstmann.com/cpp/pitfalls.html.

Horstmann, C. & Budd, T. (2005). *Big C++*, Wiley.

Josuttis, N. (1999). *The C++ Standard Library*, Addison Wesley.

Karlsson, B. (2006). *Beyond the C++ Standard Library: An Introduction to Boost*, Addison Wesley.

Kernighan, B. & Ritchie, D. (1988). *The C Programming Language*, Prentice-Hall. This is the best book on C. However, it used to be thought that the best way to learn C++ was to learn C first; it isn't thought any more. Don't do it that way.

Koenig, A. & Moo, B. (2000). *Accelerated C++*, Addison-Wesley.

Lippman, S. B. (2005). *C++ Primer (4th Edition)*, 4th edn, Addison Wesley.

Lischner, R. (2003). *C++ in a Nutshell*, O'Reilly. ISBN: 0-596-00298-X.

Louden, K. (1993). *Programming Languages: Principles and Practice*, PWS-Kent Publishing Company.

Maguire, S. (1993). *Writing Solid Code*, Microsoft Press.

McConnell, S. (2004). *Code Complete, Second Edition*, Microsoft Press.

McShaffry, M. (2005). *Game coding complete*, 2nd edn, Paraglyph Press. ISBN: 1932111913.

Meyer, B. (1996). An introduction to design by contract, *Technical report*, Interactive Software Engineering, inc. Available from www.eiffel.com.

Meyers, S. (1996). *More Effective C++: 35 New Ways to Improve Your Programs and Designs.*, Addison-Wesley.

Meyers, S. (2001). *Effective STL*, Addison-Wesley.

Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, 3rd edn, Addison Wesley.

Mitchell, R. & McKim, J. (2002). *Design by Contract – by example*, Addison Wesley.

O'Luanaigh, P. (2006). *Game Design Complete*, Paraglyph Press. ISBN: 1-93309700-0.

Penton, R. (2003). *Data Structures for Games Programmers*, Premier Press / Thompson Course Technology. ISBN: 1-931841-94-2.

Reese, G. (2007). *C++ Standard Library Practical Tips*, Charles River Media. ISBN: 1-58450-400-5.

Romanik, P. & Muntz, A. (2003). *Applied C++: practical techniques for building better software*, Addison Wesley. ISBN: 0321108949.

Sethi, R. (1996). *Programming Languages: Concepts and Constructs*, Addison-Wesley.

Smart, J. & Csomor, S. (2005). *Cross-platform GUI programming with WxWidgets*, Prentice Hall. ISBN: 0131473816.

Stroustrup, B. (1997a). *The C++ Programming Language*, 3rd edn, Addison-Wesley.

Stroustrup, B. (1997b). *The C++ Programming Language, 3rd ed.*, 3rd edn, Addison-Wesley.

Stroustrup, B. (2009). *Programming: Principles and Practice using C++*, Addison-Wesley. ISBN: 0-321-54372-6.

Sutter, H. (1999). *Exceptional C++*, Addison Wesley.

Sutter, H. (2002). *More Exceptional C++: 40 More Engineering Puzzles, Programming Problems, and Solutions*, Addison Wesley.

Sutter, H. (2004). *Exceptional C++ style*, Addison Wesley. ISBN: 0201760428.

Sutter, H. & Alexandrescu, A. (2005). *C++ Coding Standards : 101 Rules, Guidelines, and Best Practices*, Addison Wesley. ISBN: 0-321-11358-6.

Tennent, R. (2002). *Specifying Software*, Cambridge University Press.

Vlissides, J. (1998). *Pattern Hatching: Design Patterns Applied*, Addison-Wesley.

Wilson, M. (2004). *Imperfect C++*, Addison Wesley. ISBN: 0321228774.

Appendix A

Where do procedural programs come from?

A.1 Introduction

At this stage, some of you may think that designing and writing programs is some sort of God-given talent and that programs spring fully formed from the fingers of a chosen few. This opinion may be confirmed by the fact that, in books and in notes, you see only *finished* working programs.

In another chapter we will take a quick look at *object-oriented analysis and design*. In this chapter we look at one way of *deriving* (procedural) programs by appropriate analysis of the requirements leading to a design.

For certain types of problem, one way is to make your program follow the same structure as your input or output data. (When the structures of output data differs from that of input data, there is a little more work to be done, but we won't worry about that here.) An example is that if we were writing a program to process student marks, we would expect to see a *for loop* over all students and inside that a for loop over all assessment items (courseworks and examination results).

Here, we look at a problem which has only output data: patterns on a screen. If we look at patterns in data, we may be able to identify:

- Sequence;
- Selection;
- Repetition.

Any program can be constructed from these three.

In addition, based on groupings of patterns and repetitions of these groupings, we may identify the need for:

- Procedures, (methods, functions, subprograms);
- Procedures with parameters.

During the lectures we will discuss the analogies between certain computer program structures and cookery recipes. We will show that cookery recipes are frequently composed of patterns involving: sequence, selection, repetition, subprograms (recipes within a recipe), and subprograms with parameters (for example a sauce recipe which can be changed by supplying different colours or different flavours).

See Figure A.1 below.

211. Soufflé Omelet. *Cooking time* 15 minutes

Quantities for 1 large or 2 small omelets:

4 fresh eggs
½ c. water
½ tsp. salt
Pinch of pepper
½ oz. melted butter (1 Tbs.)
Chopped parsley

1. Separate the yolks and whites of the eggs and beat the yolks with the water until thick and lemon-coloured. Add the seasonings.
2. Beat the egg whites stiffly.
3. Melt the fat in the pan.
4. Fold the egg yolks into the whites very gently and pour into the pan. Cook very slowly for about 5 minutes, when the omelet should be golden underneath and beginning to rise up in the pan. If the heat is too great it will rise up quickly and then collapse and be tough.
5. Continue cooking in a moderate oven or under a slow grill for 8–10 minutes, until the top looks dry. Do not cook too long, or it will shrivel and be tough.
6. Fold in half and turn on to a hot dish. Sprinkle with chopped parsley and serve at once.

N.B. For variety this omelet may be served with a sauce poured over – e.g. Onion Sauce, No. 85; Tomato Sauce, No. 89; Cheese Sauce, No. 75; Mussel Sauce, No. 83; or Mushroom Sauce, No. 82.

89. Tomato Sauce (for fish, meat, vegetables, eggs)

1 oz. fat (25 g.)
1 onion, chopped
2 or 3 bacon rinds
1 oz. flour (2 Tbs.)
½ pt. tomato juice or 2 Tbs. tomato paste made up to ½ pt. with water (1 c.)
½ pt. stock (1 c.)
½ tsp. sugar
Salt and pepper to taste

Method used. Fry the onion and bacon rinds in the fat, and then add the flour and finish as for the Roux Method, No. 69, adding the bay leaf with the stock and tomato. Boil for ½ hour. Strain before using and add the sugar and seasonings

69. The Roux Method of Making a Sauce. *Cooking time* 10 minutes or longer

Quantities for 1–2 helpings: (When making only half a recipe use a little less than half the thickening or thin as necessary before serving; a small quantity of sauce is always thicker because there is greater evaporation during cooking.)

Ingredients	*Thin or Pouring Sauce*	*Thick or Coating Sauce*
Butter, margarine, fat or oil	1 oz. (25 g.)	2 oz. (50 g.)
Flour	1 oz. (2 Tbs.)	2 oz. (4 Tbs.)
Liquid (milk, stock, or half and half)	½ pt. (1 c.)	1 pt. (2 c.)

Measures level.

1. Melt the fat and stir in the flour. Mix well and cook very gently for 1–2 minutes or until it looks mealy for a white sauce, or until brown for a brown sauce. This mixture of fat and flour is called a 'roux'.
2. Add the cold or warm (not hot) liquid and stir or whisk vigorously until the sauce is smooth and boiling. Boil it gently for 5 minutes or cook for 10–20 minutes over boiling water, stirring occasionally. If the sauce shows any tendency to be lumpy, either beat it hard with a small wire whisk or put it in the blender for a few seconds. Whisking or blending very much improves texture and appearance.
3. If the sauce is to be kept hot for any length of time it must either be put in a double boiler or the saucepan stood in another pan of simmering water; stir occasionally to prevent a skin from forming or run a thin layer of melted butter or margarine over the top. Stir this in before serving the sauce.

Figure A.1: Computer programs have much in common with recipes (recipes from B. Nilsson, The Penguin Cookery Book)

A.2 Patterns and Programs

Write a program to display the following.

```
*        *
 *      *
  *    *
   *  *
    **
```

Next, make it general enough to take a parameter `h` (height); the example has `h = 5`. We're going build up to it — which is sometimes a great way to solve a large problem by chipping away at it until it gets smaller and smaller and finally solved.

Note, however that by *chipping away* I **do not** mean typing some rubbish and then altering random parts until it works — which is never. If you ever find yourself or anyone else taking this approach, shout *stop!* and go for help.

```
//----- Stars1.cpp --------------------------------
// j.g.c. 2003/11/29
// prints single '*' on screen
//-------------------------------------------------
#include <iostream>
using namespace std;

int main()
{
  cout<< '*';
  cout<< '\n';
  return 0;
}
```

A.3 Sequence

Now, print: *****

```
//----- Line5.cpp --------------------------------
// j.g.c. 2003/11/29
// prints 5 '*' line on screen
//-------------------------------------------------
#include <iostream>
using namespace std;

int main()
{
  cout<< '*';
  cout<< '*';
  cout<< '*';
  cout<< '*';
  cout<< '*';
  cout<< '\n';

  return 0;
}
```

Here is `Line5.cpp` formatted differently — to show that the program replicates the pattern of the data.

```
//----- Line51.cpp -------------------------------
// j.g.c. 2003/11/29
// prints 5 '*' line on screen
// now formatted to show similarity between data pattern
// and program pattern
//-------------------------------------------------
#include <iostream>
using namespace std;

int main()
{
  cout<< '*'; cout<< '*'; cout<< '*'; cout<< '*'; cout<< '*';
  cout<< '\n';

  return 0;
}
```

A.4 Repetition

```
//----- Line5r.cpp --------------------------------
// j.g.c. 2003/11/29
// prints 5 '*' line on screen
// now using repetition to replace a sequence
//-------------------------------------------------
#include <iostream>
using namespace std;

int main()
{
  int n= 5;
  for(int i= 0; i< n; i++){
    cout<< '*';
  }
  cout<< '\n';

  return 0;
}
```

A.4.1 Sequence of Repetitions

```
//----- Tr1.cpp -----------------------------------
// j.g.c. 2003/11/29
// prints a backwards facing triangle
//*
//**
//***
//****
//*****
// Analysis:
//        Stars
// Line 0: 1
//      1: 2
//      2: 3   etc...
//-------------------------------------------------
#include <iostream>
using namespace std;

int main()
{
  int n= 1;
  for(int i= 0; i< n; i++){
    cout<< '*';
  }
  cout<< '\n';
```

```
  n= 2;
  for(int i= 0; i< n; i++){
    cout<< '*';
  }
  cout<< '\n';

  n= 3;
  for(int i= 0; i< n; i++){
    cout<< '*';              //etc ...
}
```

A.4.2 Repetitions of Repetitions — Nested Loops

We can improve a lot on `Tr1.java`.

```
//----- Tr1r.cpp --------------------------------
// j.g.c. 2003/11/29
// prints a backwards facing triangle
//*
//**
//***
//****
//*****
// Analysis:
//        Stars
// Line 0: 1
//      1: 2
//      2: 3   etc...
// i.e. generalising
// Line j  j stars
//-------------------------------------------------
#include <iostream>
using namespace std;

int main()
{
  int maxj= 5;
  for(int j= 1; j<= maxj; j++){
    //----- this loop does the stars
    for(int i= 0; i< j; i++){
      cout<< '*';
    } //------
    cout<< '\n'; // and newline after every line
  }

  return 0;
}
```

Data Pattern — Program Pattern

```
//----- Tr2.cpp --------------------------------
// j.g.c. 2003/11/29
// this is Tr1r.cpp written with the repetitions translated
// to their sequence meaning; can you see that the program
// has the same pattern as the data?
//*
//**
//***
//****
//*****
//-------------------------------------------------
#include <iostream>
using namespace std;

int main()
{
  cout<< '*'; cout<< '\n';
  cout<< '*'; cout<< '*'; cout<< '\n';
  cout<< '*'; cout<< '*'; cout<< '*'; cout<< '\n';
  cout<< '*'; cout<< '*'; cout<< '*'; cout<< '*'; cout<< '\n';
  cout<< '*'; cout<< '*';  cout<< '*'; cout<< '*'; cout<< '*'; cout<< '\n';

  return 0;
}
```

A.5 Procedures — aka Subprograms, Methods, . . . Functions

A.5.1 Without Parameters

```
//----- Tr2p.cpp --------------------------------
// j.g.c. 2003/11/29
// this is Tr2 (or Tr1r) now written with procedures
//-----------------------------------------------
#include <iostream>
using namespace std;

void nl(){
  cout<< '\n';
}

void p1(){
  cout<< '*';
}

void p2(){
  cout<< '*'; cout<< '*';
}

void p3(){
  cout<< '*'; cout<< '*'; cout<< '*';
}

void p4(){
  cout<< '*'; cout<< '*'; cout<< '*'; cout<< '*';
}

void p5(){
  cout<< '*'; cout<< '*'; cout<< '*'; cout<< '*'; cout<< '*';
}

int main()
{
  p1(); nl();
  p2(); nl();
  p3(); nl();
  p4(); nl();
  p5(); nl();
  return 0;
}
```

Question. When does it make sense to create a procedure?

A.5.2 Procedures with Parameters

```
//----- Tr3r.cpp --------------------------------
// j.g.c. 2003/11/29
// this is Tr2p now written with parameterised functions
// and repetitions
//-----------------------------------------------
#include <iostream>
using namespace std;

void nl(){
  cout<< '\n';
}

void stars(int n){
  for(int i= 0; i< n; i++){
    cout<< '*';
  }
}

void spaces(int n){
  for(int i= 0; i< n; i++){
    cout<< ' ';
  }
}

int main()
{
  int height= 5;
  for(int j= 1; j<= height; j++){
    stars(j);
    nl();
  }
  return 0;
}
```

Exercise: change h and see if it generalizes okay.

Equipped with these procedures, we are now in business, and can create nearly any pattern in very quick time. The little bit of investment in designing and implementing procedures will pay back in a major way. The same is true of spending time designing, implementing, and testing OOP classes (objects); once you have a good library of classes, you can produce new programs in no time. In the games context, I suppose the next stage would be *game engine* — once you have a good game engine, creation of a new game is easy ;-).

```
//----- Tr4.cpp --------------------------------
// j.g.c. 2003/11/29
// now onto something more complex; see Tr3r.cpp
//*****
//****
//***
//**
//*
// Analysis:
//          Stars
//  Line 0    5
//       1    4
//       2    3
//         etc ...
// Generalising: let h be height
//  Line j  no. stars = h -j
//-------------------------------------------------
#include <iostream>  using namespace std;

void nl(){  cout<< '\n';  }

void stars(int n){
  for(int i= 0; i< n; i++){
    cout<< '*';
  }  }

void spaces(int n){
  for(int i= 0; i< n; i++){
    cout<< ' ';
  }  }

// notice how little change from Tr3r.java, just //1, //2
int main(){
  int h= 5;
  for(int j= 0; j< h; j++){ //1
    stars(h - j); //2
    nl();
  }
  return 0;
}
```

Exercise: Change h and see if it generalizes okay.

Now, getting more difficult ...

```
//----- Tr5.cpp ---------------------------------
// j.g.c. 2003/11/29
// getting more difficult still; see Tr4.cpp
//    *
//   **
//  ***
// ****
//*****
// Analysis:
//         Spaces  Stars
//  Line 0    4      1
//       1    3      2
//       2    2      3
//         etc ...
// Generalising: let h be height
//  Line j  h - 1 - j    j + 1
//-------------------------------------------------
#include <iostream>
using namespace std;

void nl(){
  cout<< '\n';
}

void stars(int n){
  for(int i= 0; i< n; i++){
    cout<< '*';
  }
}

void spaces(int n){
  for(int i= 0; i< n; i++){
    cout<< ' ';
  }
}

int main()
{
  int h= 5;
  for(int j= 0; j< h; j++){
    spaces(h - 1 - j);
    stars(j + 1);
    nl();
  }
  return 0;
}
```

Exercise: Based on Tr5.cpp, analyse, design and implement a program to do:

```
*****
****
***
**
*
```

```
//----- Tr10.cpp --------------------------------
// j.g.c. 2003/11/29
// getting more difficult again; see Tr5.cpp
//               Spaces   Stars
//************     0       12
// **********      1       10
//  ********       2        8
//   ******        3        6
//    ****         4        4
//     **          5        2
// Let h be height and count from 0
//           Spaces    Stars
//  Line  0    0        h*2
//        1    1        (h-1)*2
//        2    2        (h-2)*2
//               etc...
// Generalise: line j: j spaces, (h-j)*2 stars
//-------------------------------------------------
#include <iostream>
using namespace std;

void nl(){
  cout<< '\n';
}

void stars(int n){
  for(int i= 0; i< n; i++){
    cout<< '*';
  }
}

void spaces(int n){
  for(int i= 0; i< n; i++){
    cout<< ' ';
  }
}

int main()
{
  int h= 5;
  for(int j= 0; j< h; j++){
    spaces(j);
    stars((h-j)*2);
    nl();
  }
  return 0;
}
```

Exercise: In Tr10.java, change h and see if it works okay.

```
//----- Tr20.cpp --------------------------------
// j.g.c. 2003/11/29
// finally, we get to the original problem.
// from Tr10.java
//            Analysis
//             Sp1 St  Sp2  St
//
//*        *    0  1    8   1
// *      *     1  1    6   1
//  *    *      2  1    4   1
//   *  *       3  1    2   1
//    **        4  1    0   1

// And let's generalise (height= h) & count from 0
//        Sp1       Sp2
// Line 0: 0        (h-1)*2
//      1: 1        (h-2)*2
//      2: 2        (h-3)*2   etc.
//     ---------------------
// i.e.
// Line j: j        (h-j-1)*2
//-------------------------------------------------
#include <iostream> using namespace std;

void nl(){ cout<< '\n'; }

void stars(int n){
  for(int i= 0; i< n; i++){
    cout<< '*';
  }  }

void spaces(int n){
  for(int i= 0; i< n; i++){
    cout<< ' ';
  }  }

int main() {
  int h= 5;
  for(int j= 0; j< h; j++){
    spaces(j);
    stars(1);
    stars((h-j-1)*2);
    stars(1);
    nl();
  }
  return 0;
}
```

Exercise: In Tr20.java, change h and see if it works okay.

A.6 Selection

```
//----- Tr10sel.cpp ---------------------------------
// j.g.c. 2003/11/29
// as Tr10.java, but selecting stars, stripes
//************
// ----------
//  ********
//   ------
//    ****
//     --
//-------------------------------------------------
#include <iostream>
using namespace std;

void nl(){
  cout<< '\n';}

void stars(int n){
  for(int i= 0; i< n; i++){
    cout<< '*';
  } }

void dashes(int n){
  for(int i= 0; i< n; i++){
    cout<< '-';
  }
}

void spaces(int n){
  for(int i= 0; i< n; i++){
    cout<< ' ';
  }
}

int main(){
    int h= 6;

    for(int j=0; j< h; j++){
      spaces(j);
      if(j%2==0)stars((h-j)*2); // even number if remainder of j/2 == 0
      else dashes((h-j)*2);
      nl();
    }

  return 0;
}
```

A.7 Exercises

1. Write a C++ Program to display an arbitrary filled rectangle of *s – specified by h(eight) and w(idth);

2. Write a C++ Program to display a rectangle outline, 5×5.

3.(a) Write a C++ function, `printHexDigit(int n)` which will print a Hexadecimal digit — 0, 1, ... 9, 10 (A), ... 15 (F) as follows. Design your own font for 3 onwards.

```
*****    *   *****    *****
*  **    *       *        *
* * *    *   *****    *****
**  *    *   *            *
*****    *   *****    *****
```

(b) If you were writing programs that needed to generate sequences of these digits, can you think of additional procedures that would be useful, or would you make do with `stars`, `dashes`, `spaces`, and `nl` that we already have.

4. Having a nice library of procedures saves programming time (designing the program, typing, ...) for programmers. Can you think of any other benefits? Hint: (i) what do you do after you have written what you think is a correct program? (ii) will a program that has procedures like `one`, `two`, etc. be more or less understandable by a human than one constructed from just `stars`, `dashes`, `spaces`, and `nl`? (iii) will a program that has procedures like `one`, `two`, etc. have more or less source lines than on constructed from `stars`, `dashes`, `spaces`, and `nl`.

Appendix B

Analysis of Algorithms

This chapter is a slight modification of notes provided by Robert Lyttle of Queen's University Belfast.

In considering the trade-offs among alternative solutions to problems, an important factor is the efficiency. Efficiency, in this context, is measured in terms of memory use and time taken to complete the task. Time is measured by the number of elementary actions carried out by the processor in such an execution. In the interests of brevity, this course discusses only time efficiency.

It should be noted that there is often a trade-off between time an memory; often, you can *buy* time performance (speed) by using extra memory.

It is difficult to predict the *actual* computation time of an algorithm without knowing the intimate details of the underlying computer, the object code generated by the compiler, and other related factors. But we can *measure* the time for a given algorithm, language compiler and computer system by means of some carefully designed performance tests known as *benchmarks*.

It is also helpful to know the way the running time will vary or grow as a function of the problem size — a function of the number of elements in an array, the number of records in a file, and so forth. Programmers sometimes discover that programs that have run in perfectly reasonable time for the small test sets they have used, take extraordinarily long when run with real world sized data sets or files. These programmers were deceived by the *growth rate* of the computation.

For example, it is common to write programs whose running time varies with the *square* of the problem size. Thus a program taking, say, 1 second to sort a list of 1000 items will require not two (2), but four (4) seconds for a list of 2000 items. Increasing the list size by a factor of 10, to 10,000 items, will increase the run-time to $10^2 = 100$ seconds. A list 100,000 items will require 10,000 (10^4) seconds, or about 3 hours, to complete. Finally, 1,000,000 items (e.g. a telephone directory for a small country) will need 10^6 seconds (almost two weeks) to finish! This is a long time compared to the one second taken by the 1000 item test.

This example shows that it makes sense to be able to analyse growth rates and to be able to predict (even roughly) what will happen when the problem size gets large.

B.1 O Notation (Big-oh)

Algorithmic growth rates are expressed as formulae which give the computation time in terms of the problem size N. It is usually assumed that system dependent factors, such as the programming language, compiler efficiency and computer speed, do not vary with the problem size and so can be factored out.

Discussions of growth rate normally use the *Big-oh* notation (*growth rate*, *order of magnitude*).

The most common growth rates we will encounter are the following:

- $O(1)$, or *constant*;
- $O(\log N)$, or *logarithmic* (logarithm usually taken to the base 2);
- $O(N)$, or *linear* (directly proportional to N);
- $O(N \log N)$, pronounced N *log* N;
- $O(N^2)$, or *quadratic* (proportional to the square of N).

The table that follows shows the value of each of these functions for a number of different values of N. It shows that as N grows, $\log N$ remains quite small with respect to N and $N \log N$ grows fairly rapidly, but not nearly as large as N^2.

In (Campbell 2009) we will see that simple searching grows as $O(N)$ (linear), but a binary search grows as $O(\log N)$. We also see that most good sorting algorithms have a growth rate of $O(N \log N)$ and that the slower, more obvious, ones are $O(N^2)$.

log2(N)	N	N log2 N	N^2	2^N	N!
0	1	0	1	20.000E-1	10.000E-1
1	2	20.000E-1	4	40.000E-1	20.000E-1
2	4	80.000E-1	16	16.000E+0	24.000E+0
3	8	24.000E+0	64	25.600E+1	40.320E+3
4	16	64.000E+0	256	65.536E+3	20.923E+12
5	32	16.000E+1	1024	42.950E+8	26.313E+34
6	64	38.400E+1	40.960E+2	18.447E+18	12.689E+88
7	128	89.600E+1	16.384E+3	34.028E+37	38.562E+214
8	256	20.480E+2	65.536E+3	11.579E+76	*
9	512	46.080E+2	26.214E+4	13.408E+153	*
10	1024	10.240E+3	10.486E+5	*	*
11	2048	22.528E+3	41.943E+5	*	*
12	4096	49.152E+3	16.777E+6	*	*
13	8192	10.650E+4	67.109E+6	*	*
14	16384	22.938E+4	26.844E+7	*	*
15	32768	49.152E+4	10.737E+8	*	*
16	65536	10.486E+5	42.950E+8	*	*

```
* too large to be computed in a double variable
aEp means a x 10^p
```

B.2 Estimating the Growth Rate of an Algorithm

In estimating performance we can take advantage of the fact that algorithms are developed in a structured way — that is, they combine simple statements into complex blocks in four useful ways:

- *sequence*, or writing one statement below the other;
- *selection*, or the well known if-then or if-then-else;
- *repetition*, including counting loops, while loops, etc.;
- *method calls*.

In the rest of this section, some typical algorithm structures are considered and their $O()$ estimated. The problem size is denoted by N throughout.

B.2.1 Simple Statements

An assignment statement is an example of a simple statement. If we assume that the statement contains no method calls (whose execution time may, of course, vary with problem size), the statement takes a fixed a mount of time to execute. This type of performance is denoted by $O(1)$ because when we factor out the constant execution time we are left with one.

Sequence of Simple Statements

A sequence of simple statements obviously takes an amount of time equal to the sum of the times it takes each individual statement to execute. If the performances of the individual statements are $O(1)$, then so is that of the sum.

B.2.2 Decision

When estimating performance, the `then` clause and the `else` clause of a conditional structure are considered to be independent, arbitrary structures in their own right. Then the larger of the two individual *big-Oh*s is taken to be the *big-Oh* of the decision.

A variation of the decision structure is the *switch* structure, really just a multi-way `if-then-else`. Thus in estimating performance of a `switch`, we just take the largest *big-Oh* of all of the switch alternatives.

Performance estimation can sometimes get a bit tricky. For example, the condition tested in a decision may involve a method call, and the timing of the method call may itself vary with problem size.

B.2.3 Counting Loop

A counting loop is a loop in which the counter is incremented (or decremented) each time the loop is iterated. This is different from some loops we shall consider later, where the counter is multiplied or divided by a given value.

If the body of a simple counting loop contains only a sequence of simple statements, then the performance of the loop is just the number of times the loop iterates. If the number of times the loop iterates is constant — i.e. independent of the problem size — then the performance of the whole loop is $O(1)$. An example of such a loop is:

```
for (int i = 0; i < 5; i++){
  statements with O(1)
}
```

On the other hand if the loop is something like:

```
for (int i = 0; i < N; i++){
      statements with O(1)
}
```

the number of times the loop iterates depends on N, so the performance is $O(N)$.

Discussion of Single Loop – O(n) Consider the following code in which the individual statements/instructions are numbered:

```
int sum=0; //s1
      //s2       //s3     //s4
for (int i = 0; i < N; i++){
  sum++; //s5
}
```

Here, `s1`, and `s2` are performed only once; `s3`, `s4` and `s5` are performed N times. Hence, associating a time t_j with instruction j, we have:

$$t_{tot} = t_1 + t_2 + N(t_3 + t_4 + t_5)$$

Normally, it will be the case that `s5` will be the most expensive; however, just to be fair, let us assume that all instructions take the same time – 1 unit, e.g. 1 microsecond. Let us see how t_{tot} behaves as N get large.

N	Ttot
1	5
10	32
100	302
1000	3002

Hence, we see that it is the term $N(t_3 + t_4 + t_5)$ which becomes dominant for large N. In other words, we can write:

$$t_{tot} = cN + \text{negligible terms}$$

and that the algorithm has a running time of $O(n)$.

Consider the double counting loop:

```
for (int i = 0; i < N; i++){
   for (int j = 0; j < N; j++){
       statements with O(1)
   }
}
```

The outer loop is iterated N times. But the inner loop iterates N times for each time the outer loop iterates, so the body of the inner loop will be iterated $N \times N$ times, and the performance of the entire structure is $O(N^2)$.

The next structure:

```
for (int i = 0; i < N; i++){
   for (int j = 0; j < i; j++){
      statements with O(1)
   }
}
```

looks deceptively similar to that of the previous loop. Again, the outer loop iterates N times. But this time the inner loop depends on the value of i (which depends on N): if `i = 1`, the inner loop will be iterated once, if `i = 2` it will be iterated twice, and so on, so that, in general, if `i = N`, the inner loop will iterate N times. How many times, in total, will the body of the inner loop be iterated? The number of times is given by the sum:

$$0 + 1 + 2 + 3 + \ldots\ldots\ldots\ldots + (N-2) = \sum_{i=1}^{N-2} i$$

Noting that $\sum_{i=1}^{N} i = \frac{(N+1)N}{2}$, the summation above is equivalent to $\frac{(N-2)(N-1)}{2} = \frac{N^2-3N+2}{2}$. The performance of such a structure is said to be $O(N^2)$, since for large N the contributions of the $\frac{3N}{2}$ and $\frac{2}{2}$ terms are negligible.

As a general rule: a structure with k nested counting loops — loops where the counter is just incremented (or decremented) by 1 — has performance $O(N^k)$ if the number of times each loop is iterated depends only on the problem size. A growth rate of $O(N^k)$ is called *polynomial*.

In (Campbell 2009) we will show that simple *divide-and-conquer* algorithms like *binary search* grow as $\log N$.