Learning to Program in C and C++

by Graham J. Ellis

Learning to Program in C and C++

by Graham J. Ellis

Well House Consultants, Ltd
404, The Spa
Melksham
Wiltshire SN12 6QL
England

+44 (0) 1225 708 225
+44 (0) 1225 707 126 (fax)

Find us on the World Wide Web at: *http://www.wellho.net*
Or contact us at: *enquiries@wellho.net*
(Graham's direct email is: *graham@wellho.net*).

Design by Lisa Ellis.

Printing History

October 2006	1.0	First Edition
June 2008	1.1	New modules added

This manual was printed on 22 March 2015.

Notice of Rights

Notice of Liability

Well House Consultants LTD.

Contents

1

Learning to Program

This module introduces newcomers to computer programming to programming principles (and those who are rusty or who feel they have gaps in their basic knowledge).

At Well House Consultants, we offer courses at two levels in a number of programming languages. Our "learning to program in ..." courses are for delegates who have never programmed before, or who are rusty, who lack confidence, or want a refresher of the basic principles. Our "... programming" courses are for delegates who have prior programming experience, but are converting from another programming language to the one that we're teaching at the time. By offering two different starting points in this way, we can ensure that newcomers to programming aren't swamped in the first hours, yet experienced programmers don't have to sit through a day of the basics.

This module accompanies the "learning to" courses and intentionally leaves out code examples. It's very easy for me to show you "one I wrote earlier", but that doesn't teach you how to write a program for yourself. You need to see the thought process, so it will be done by demonstration. The examples written will be made available to you after the course, and you'll also find code in the remaining modules in these notes which also apply to the courses where we start with experienced programmers.

1.1 So - let's start!

Stored programs
- We start with a series of instructions in a text file
- Each instruction is separated from the next somehow
- Instructions are run sequentially
- Possibly wrapped in some sort of named block to say "Start here"

Running a stored program
- We need to translate from text to runnable
- That may be through an intermediate process; there are three common ways (compiler, pure interpreter, and virtual machine)
- There will be common code to all or most applications, and that will be loaded from standard libraries

Hello World and Hello delegates too

It's traditional on all programming courses for the first example to output the words "hello world" onto the programmer's screen. Of course, that's much simpler than any practical program, but it shows you
- How you input the program, what you write and where you store the "source code"
- How (and if) you translate that program into a different format from which you can run it
- How you run the program

This will be demonstrated, and we'll have you try it out too. It will feel clunky at first. Don't worry about that, you'll speed up later, and there will be lots of other things to learn about in your chosen language that will make it easier too.

At this point, one of things to think about is how portable your program will be between different computer architectures and operating systems. You may feel it's too early to look at this, but right from the start you'll want to know about the portablity and re-usability of your work.

Commenting your code

Also at this point, we'll take a first look at code documentation. "Hello World" really doesn't need too much backup information for the maintenance programmer – either a colleague of yours, or you yourself when you come back and try to remember what you did in 6 months or 6 years time because it needs updating.

- Every language allows for comments in some way or another, and we'll add some to "Hello World".

Comments make no difference to the running of your code, but they make a huge difference later on. You should include information about what the blocks of code do, and also any notes about the environment in which they are designed to run. Version numbers, copyright, support contact details and terms and conditions are also worth considering in serious code blocks.

As well as programmer's comments, instructions for your user should be provided. Most languages provide you with specific tools and an ability to build these instruction into your source code. And it's very likely too, with modern programming, that you'll also provide some sort of test suite that lets you and your customers check that the program's functionality is still working as planned after upgrades and changes.

1.2 Operators and operands (or commands)

- The two language patterns, and which [target] uses
- Writing a numeric expression
- Bodmas and brackets

1.3 Variables

- Storing a result under a name for later use
- Variable naming rules
- Declaring variables – type, size and scope, perhaps?
- Integer, Float, String and Boolean Families
- Other types, and your own types
- Strong or weak typing
- Casting, converting and co-ercing
- Outputting a variable's content

1.4 Constants

- Writing constants – implicit type
- Language support for constants
- Giving constants a name for maintainability

variable basics

Information – data – needs to be stored in a program between statements. Or rather it needs to be stored in the computer's memory. At the lowest of levels, that's a binary pattern of 0s and 1s in a numbered memory location that's encoded in such a way that it can be formed back into something that represents a number, or a string of text. In the very early days I've programmed computers that work like this, and it works, but it's pretty impractical for anything but elementary programs. So what do we do?

- We give descriptive names to the places we need to store the data
- We allow the programming language stuff to decide where to store the data in memory so we don't have to bother
- We have the programming language deal with all the low-level formatting too

Variable names are typically the programmer's choice, subject to a strict series of rules that differ from langauge to language. They'll be comprised of a letter, followed by more letters, digits and underscores. Maximum name length, whether upper and lower case letters have a different meaning, whether a variable name may start with an underscore differ. Also

- In some languages, variable names are [sometimes] preceeded by a special character – a "sigil"
- And in some languages, certain words can't be used as variable names - "reserved words" such as **`if`** and **`break`**.

In some languages, the programmer is required to state the names of the variable that will be used so that the compiler can allocate memory efficiently; that also has the benefit of making the programmer think about exactly what's going on. In other languages, it's the langauge internals which work out what storage is needed, and how it's to be used and coded, based on the context in which it's used in the code. Although this latter solution sounds easiest to write and is good, it *does* have the disadvantage that it's all too easy for a variable to be misnamed, and for the programmer to end up with a bug that's hard to find.

On types and Scope

I've mentioned different types of data that need to be stored. There are whole numbers, numbers with decimals, pieces of text, and others too which are collections or groups of variables, booleans ("true" or "false") and indeed composite variables of our own type definition. As you get deeper into programming, you'll need to understand these various type.

Data sometimes need to be converted between types; for example, a string of characters input by our user at the keyboard needs converting into a number on which calculations can be done. In some languages, this is done automatically for you, but in others you have to request explicitly that it be done.

There's also the matter, as programs grow, of how long the data in them (and the name) is retained. It would be wasteful in a long running program to retain data that's only required for a very short period as the program starts up right through to when the program finished running, but it would be frustrating if the programmer came to use a piece of data to find that it had gone away. There's the further matter here of a program that's got sections written by different programmers, and the need for variable names used by each of the programmers to be distinct from each other. Think of two families living in the same town, both of whom have daughters they name "Lorna". That's fine and good around the home, but when the two young ladies end up in the same class as school, the teacher says "please stand up Lorna" and both will stand up. So there needs to be something extra. This subject is called the "scope" of a variable, and it's so important that we raise it even on your very first day of programming to make you aware of an upcoming issue, though solutions and detailed discussions must wait until we're further into the course.

Constants

There are some numbers – "constants" – which won't change (or you don't expect to change) in your program. There are 24 hours in a day, and seven days in a week. And there are some values which are constant to you, such as the maximum number of delegates on a public course might always be 7, the number of working days in the week might be 5, the BMI levels below which and above which a person is regarded as being unhealth may be 20.0 and 25.0.

There's nothing to stop you writing these numbers directly into your program, but that's not a good idea:

- You'll find it hard to find all occurrences of the number should it ever change
- You'll write code that's confusing in the extreme if the same constant happens to apply to two things
- The code won't be very descriptive when you come to read it back.

So you'll find that early on we recommend you assign constants into named locations, and on this first day of the course we'll use variables. However, many languages

also support a special notation for named constants, and if you use that:
- Your code can run more efficiently as there needs to be no mechanism to amend a value
- In languges which statically assign memory, a whole heap of complexity can be solved if the constant is a "maximm number of ..."
- The maintenance programmer is clearly told "this value won't be changing at run time"
- The constant can be much more widely scoped so that it's available right through your code without scope conflicts.

1.5 Your first useful program ... needs user input

- Reading from the user
- Converting a string into the right type
- The need for validation (to come back later)
- Exercises!

1.6 Conditionals

- Boolean Conditions
- Optional coding
- need for blocks to define how much is optional
- **`elseif`** and **`else`**
- Testing needs increased
- **`if`** - **`unless`** - **`switch`** - note shorthands for later
- what is equality
- equality in floats
- nesting
- Exercises!

Every language has some sort of conditional statement. That's a way of looking at some sort of setting or status in the program and performing some sort of action based on that setting or status.

Such statements take the form:

```
if {some sort of condition is true} then {run a group of statements}
```

if

The word **`if`** applies in every language that we teach at Well House Consultants at present, but how we define the condition, how we signify the end of the condition and the start of the group of statements, and how we stop and start that group varies.
- a "block" of code is a series of statements grouped together. Actually zero or more statements, as at times you'll want to have a "do nothing" group
- "Delimiters" are the characters or character groupings thet start and end blocks. They may be the words **`then`**, they may be the characters **`{`** and **`}`**, or they may be a pattern of spaces and tabs that insets the block in the source code. Sometimes they may be left out, and if the language supports that they imply a block of a single statement.

The condition that's used in an **`if`** statement is going to be an expression that evaluates to a "yes" or "no" value – true or false. Exactly what comprises **`true`** and **`false`** varies between languages. Very often if your conditional expression works out at zero, that's false and if it works out to any other number, that's true. But numbers are only used in this way a small proportion of the time, as languages come with special operators that compare two values and return true or false based on that comparison.

Commonly they are:

==	"is equal to"
!=	"is not equal to"
<	"is less than"
<=	"is less than or equal to"
>	"is greater than"
>=	"is greater than or equal to"

But beware, in some languages (SQL and Lua) even these vary, and in many languages (Perl, PHP, Shell for example) there are alternatives which do somewhat different things, and in some (Java, Python, Ruby, SQL, C, C++ for example) there are functions which you may call to make alternative comparisons. In other words, you're never limited to just the six comarisons.

These are notes to accompany your "Learning to program in xxxxx" course at Well House Consultants, so I'm not going to attempt to describe all the options here. Instead, I will demonstrate the first, most basic conditional statements to you at this point and let you try them out.

One of the questions that comes up for newcomers to programming is "why do I need a closing delimiter?". It's needed to tell your program that you've reached the end of code that's only run in certain circumstances and you're back into "always" code beyond that point. Imagine that you're driving a car and you decide to pull into a service station:

```
 if (ineed == "loo") then { ......}
```

The block of code in the curly braces defines what you need to do in the service station, but then when you get back on the main road, it's "carry on as before", even if one or two of the variables (such as your comfort level) have been amended. The closure is vital as it removes the need for all subsequent code to be repeated. It's an indication of the coming together.

elseif and else

Usually, you'll want to perform one action if a condition is true, and some different action if the condition is false. Whilst it would be possible for you to write the "opposite" **`if`** statement, that's inefficient (at writing and run time) and prone to error, so languages support some sort of "otherwise" statement.

You can follow an **`if`** with one or more **`elseif`** (or **`elsif`** or **`elif`** clauses[1]) which in each case will have a further condition attached to them, and they'll have a block of code that runs if that alternative condition is true.

Note that the order of the various conditions is important, as once a true condition is found as the code runs, that's the block that will be run and the following ones won't be, even if the condition on them is also true.

Finally, you may finish your **`if`** statement with **`else`** and a block to accompany it. This is your 'catch all' or safety net which will be performed if neither the **`if`** condition, nor any of the **`el[se]if`** conditions were true. The **`else`** is optional; you can only have one of them, and there is no condition attached to it.

nested and joined conditions

You'll often find that you want to test for conditions within conditions (i.e. within the block of what to do) and you can do this. If you've stopped at the service area above because you needed a natural break, you'll be making other subsidiary decisions in there about whether to use the loo, have a coffee, buy sweets for the kids, call your desination to update your arrival time, etc. Note that you'll complete all of those extra actions before you complete the main action of making a stop at the service area. So use a nested conditional that starts in the order of 1, 2, 3 but ends 3, 2, 1.

[1] the keyword varies from language to language

There are also times that you want to perform a certain action only if two conditions are true. You could do this with nested blocks, but you'll also have an alternative, typically using the words **`and`** or **`or`** to link up conditions into a single composite conditions. Very often, either **`&&`** or **`&`** are alternatives (with subtle differences) for **`and`**, and **`or`** may be relplaced by **||** or **|**. This is a subject for much deeper study later in the course.

testing

As soon as you start introducing conditional code, you introduce multiple routes through the code so it becomes very important to give throught to a thorough testing regime.

As a minimum, you should test your progran before its use in a live application by running every single possible condition through its true and false routes. And you should consider also:

- Running your code such that all combinations of conditions are tested
- Testing your data where both valid and invalid user inputs are made
- Remember to test "boundary" conditions; if you're testing for age under 18, run your code with (say) 16 and 20 , but also with 18 itself.

Testing gets to be repetitive and (let's admit it) a bit boring at times, and it's far too easy for us to skip. Yet it really should be repeated in full for each and every iteration and release of the code. We'll broach the detail of testing later on the course, but for the moment bear in mind that a standard set of tests, automated in a file so that you can easily rerun them, and with extra software to pick up hundreds or thousands of passes and the occasional fail is going to be far better that your programmer working through each and every test at every upgrade. You might even want to write the tests before you write the code that it's going to be testing – that's "Test Driven Development "or "TDD".

1.7 Loops

- repeating block of code
- difference to conditional
- need to ensure you always exit the loop
- break and perhaps others
- Exercises!

If your program always ran each statement just once (indeed skipping over statements which were in blocks in false conditions) it would run very quickly and would have little use. You couldn't (for example) run a program which went through a whole series of results from a database query and displayed particular data from each of them (plus perhaps a summary on the end).

while

So all programming languages have the ability to repeat a block of code, and the most fundamentel of these repeats ("loops") looks like this:

```
while {some sort of condition is true} then {run a group of statements}
```

You'll note that this is exactly the same format as the **`if`** statement in the previous section, apart from the replacement of the word **`if`** by the word **`while`**. Operationally, it differs in that once the group of statements in the block has been performed, the program rechecks the condition and if it's still true it runs the block again, keeping doing so until the condition becomes false.

Note:

- If the condition is false when first checked, the block isn't performed at all; a loop

runs zero or more times
- If the condition never goes false you potentially have an infinite loop that goes on forever
- Conditions are just the same as the conditions mentioned for the **`if`** statement in the language you're learning.

Newcomers to programming sometimes take a few minutes to grasp their first program with a loop statement, as for the first time the code jumps backwards as well as forwards as it runs. And they sometimes have trouble working out which statements go where.
- If something may need to be run multiple times, it goes within the block (or within the condition to the **`while`**)
- If something only runs once and that's before the code in the block that may repeat, it goes before the **`while`** ("initialisation")
- If something runs once after any repeated code, that goes after the block that may repeat (e.g. printing a total)

I will show you a **`while`** loop in the language you're learning at this point, and have you write one too.

breaking out of loops

At times, you'll want to jump out of the middle of a loop and continue running code below ; if (for eaxmple) you've identified an incoming record that you were looking for within a stream of data, or if you have reached a threshold. Some languages provide you with a statement that you can put within a loop to get out of that loop even though the condition at the top hasn't been checked and has gone false. The keyword used is usually **`break`** but sometimes (in some languages) it's **`last`**.

Putting a break into a loop to be run unconditionally would be rather pointless, so you'll find that any **`break`** statements will be within a conditional statement such as (but not limited to) an **`if`** within the loop.

Most languages also support a **`continue`** statement (sometimes **`next`**) which allows you to skip the rest of the block code and go back up to test the condition straight away. Very useful if you're filtering a stream of data and you've identified a record such as a comment that you want to skip over without further processing.

Some languages have other flow controls in loops too; you may come across **`redo`** and **`retry`**. Early programming languages supported **`goto`** statements (and indeed some still do), but other than exceptional circumstances, their use is discouraged. They make for code that is very difficult to debug or to follow, and often impractical to upgrade when specifications change, and there are now far better ways.

Other loops

`while` is the most fundamental loop statement, and it's the natural one for us to choose to introduce on this first day of our "learning to program in ..." course. But there are many other forms of loops that you'll come across later in the course, and in practice you'll find such other loops used more than **`while`**. Keywords are **`for`**, **`foreach`**, **`loop`**, **`until`** and **`repeat`**.

If you are writing web applications (Java Servlets or JSPs, PHP scripts, or CGI or otherwise server based code in other languages such as Ruby, Python, Tcl, Lua, C or C++), your whole application will potentially be run in a loop. Each time a user submits a form to your code, (or an automated client calls up your URL), the code is run in a loop that's controlled by the web server with your code being the innerds of the loop. This means that there are times where code towards the top of your web application may be run after code that appears later in your web application (due to condtionals) even though it doesn't look like it's in a loop at first glance.

1.8 Algorithms - a first bite

- Accumulator
- Min, max, average

There are common themes for how programming statements are put together to give a complete section of code to perform combined tasks. And typically these are putting together building blocks in a similar way to how we would do things if we were working something out by hand.

Looking for the maximum value in a column of numbers, for example, we would start off by guessing that the first number was the maximum, and then we would check against each of the following numbers to see if it was greater, updating our guess if it was. Come the end of the reading down the column, the final value is no longer just a guess, it really is the maximum.

Such standard application of coding is known as the application of "algorithms" or "design patterns" –that latter term is especially applicable to what we'll describe to you later on the course as "Object Oriented Programming".

All the languages that we teach have at least some algorithms or design patterns built into the language, as standard pieces of code in the library that we've referred to earlier in this module. In some languages, such as Lua and Tcl the standard libraries are quite small, and on the course we'll be showing you how to code certain algorithms yourself. In others such as PHP, we have a standing joke around the class that says "there's a function to do that" and indeed a huge array of common functions are available to you which you can call up in a single line to run a particular algorithm against some of your variables, passing back a result into another variable. Java, Python, Perl and Ruby – and some of the other languages – have a large number of algorithms available to you included in the language distribution (and present on your computer's disc or file system) but only loaded into memory at run time, typically on your request through a program statement asking for them to be loaded. And resources are available in virtually every language on the web to provide shared algorithms which, whilst commonly enough needed to be included in the distribution, are nevertheless worth sharing.

1.9 Documentation - a first bite

- Comments
- Commentish code
- User documentation
- Exercises!

In this final session of "learning to program in xxx", I'm going to stress the importance of some of the more mundane elements, but practical requirements of programming.

1. You should comment your code so that the programmer who follows behind you (or yourself in a year or two) can understand what has been done, and pick it up, fix issues, and update it. You should also include things like a version number so that you're not left puzzling which version is which, nor fixing an old version
2. You should document your code so that the user who makes use of it knows how to do so. That user could be someone visiting your web page, running your application, or another programmer calling in elements of your code as part of his program; we've seen the use of library routines during the day so far, and you may be providing such library routines.
3. You should provide smple and/or test code so that you can test the continued operatation of the application and routines in the future, and ensure that they're still running right when you transfer a copy to a computer running a different operating system.

Comments and documentation are not, entirely, extras that you have to provide in addition to the running code. By a good choice of variable names, and by insetting the code sensibly and spacing it out, you're doing much of that work within the code. And your test code not only helps you get the code right in the first place, but also provides examples of how it should be used for others who'll be making use of your code.

1.10 Looking ahead

From here on, we're looking far more at demonstrations to show you the way ahead. There's a revision of how the subjects above relate to the particular language on the main programming course that follows the "learning to program" day, and we'll go on to cover many of the following topics in much greater depth too, and with practicals!

Structure and more blocking
- Avoid repeated code via loops
- Avoid repeated code via named blocks
- Parameters in, parameters out
- Why default global is convenient but bad
- static variables or a clean start?
- Scoping in [target]
- Namespaces, and Structured and OO code

Collections
- Need to store multiple values under a single name
- Accessing via indexes
- Keys 0 based, 1 based, or not based at all.
- Fixed or variable length?
- Dealing with overflow

Pointers
- Extended collections (objects, structures, unions)
- Passing multiple bits of data as one
- Multiple names
- Symbol table and heap model
- Garbage Collection

Loading and Libraries
- Don't reinvent the wheel
- Sharing code between programs
- Sharing code between programmers
- Library Load order
- Mixing languages
- Version Control

Design
- Structured programming and the OO model
- What the user requires
- UML - using the concepts at least
- Future Proofing

And also ...

- Usability, maintainability, robustness and legality
- Debugging and tools
- Other algorithms – sorting and selecting
- Coroutines, parallel processing, threads, network resources
- Coding environments and standards
- Updates and language upgrades
- Security – abuse, misuse and error
- Tailoring Standard Applications
- User training and support
- How does it work on The Web
- Open source, sell your programs, or just use yourself?

Just to note ...

- HTML, XML and SQL are not programming languages
- But stored procedures, XSLT and even bash are

Having started off with a gentle introduction to programming in your language of choice for the first three quarters of this module, we'll move on during this last session to give you an insight into where we're headed. You've been introduced to programming, or had a good chance to review and pick up some of the formalities if you've not previously been formally taught. But you won't have seen the depth of capabilties of the language, nor its practical application, that will come as the course carries on with further modules. If this is a public course, we may be joined by other delegates with prior programming experience at the start of tomorrow, and we'll be covering the same headline topics as a revision for you, but widening out and looking at them in a quicker, more complete way. As of now, I expect you to have lots of questions and you may wonder "what have I let myself in for?". Don't worry, we have all (including tomorrow's joiners) been there, and it's important for you to be sure of all the basic facets of the subject at the end of the course, rather than at the end of each day.

2

C Programming Introduction

What is the C language? When do we use it and why? How does it compare to other languages?

2.1 Introduction to C

The C language is the bedrock of modern computing. So why is it that a company like Well House Consultants, who specialise in niche training, are running a C course? It's because the bedrock is something that, whilst it's there and vital, most people don't need to understand. I expect that most of you couldn't tell me very much about your home's foundations – how deep are they, what are they made of, what material do they lie on. And in the same way, C is vital to us all, but only a few of us need to learn it.

C is not an "object oriented" language. If you require the bedrock of C and also the extra facilities offered by OO, you can select C++ which offers compatibility with C, and also the extra facilities. But note that they are offered at a price, and that price is a complexity that is not necessary for most people, and not necessary if the underlying C compatibility can be foregone. That's why you have languages such as Java (from Sun) and C# (from Microsoft) which are developed using the approach of C, and the power of object orientation, but without the C compatibility, and so without much of the low-level coding that C's been so successful with.

Perhaps you're looking at a PHP script, or a piece of Python and saying "C isn't important to me".

Actually, it is important.

Your PHP and your Python are written in C (Jython is written in Java and that has the underlying C level). Your web server is written in C. Your operating system is written in C. Don't underestimate C; it is vital to you. It's just that it may not be vital for you to understand it.

2.2 Compiling and loading C programs

1. Enter your source code (a file extension *.c* is common).

2. Compile into an object file (extension *.o* or *.obj*). This is a binary file that contains machine code for the machine that you'll be running on, but it's not yet a complete program; it's a program component. In effect, your compiling has turned a raw potato into a roasted one, but it's still not a complete meal.

3. Link / Load / Taskbuild your *.o* or *.obj* files; that joins them together into a single conglomerate executable file, and brings in standard library files too, so that the file as a whole can be run. You have now added your Roast Beef, Yorkshire Pudding, and brussel sprouts and made up a complete course.

The compiler will initially run the C pre-processor, which will act on lines starting with a # character. It allows for other files to be included, constants defined, and selective-debug code and system-dependent code to be included as appropriate.

The whole process of one or more compiles followed by a link may be defined in a makefile. The Makefile defines the commands necessary for each step of the process, and also lets you define which file depends on which other file. The net effect of this is to enable the compiler to skip over files that haven't been changed since you last did a compile by looking at the timestamp on the *.c* file in relation to the timestamp on the *.o*. It's very clever; I remember back to "pre-make" days and running compiles and loads of a big CAD system I wrote that took nearly an hour to process!

Sample makefile

```
# makefile for module C202

# all target (also first target) - make all executables
# clean target - cleans up all intermediate files
#     note use of @ to supress output reports

all:    adder tconv nranges

clean:
        @rm -f *.o
        @rm -f adder tconv nranges

adder:  adder.o
        gcc -o adder adder.o

adder.o:        adder.c
        gcc -c adder.c

tconv:  tconv.o
        gcc -o tconv tconv.o

tconv.o:        tconv.c
        gcc -c tconv.c

nranges:        nranges.o
        gcc -o nranges nranges.o

nranges.o:      nranges.c
        gcc -c nranges.c
```

3

C Language Fundamentals

"Hello World" is the shortest program in C. This example teaches you about compiling, loading and running a C Program.

3.1 "Hello World"

Example hello.c

```
/*
First C program
*/

int main() {

        /* This is a comment in C */

        printf("This is a C program\n");
        printf("Another line of program\n");
        }
```

Compile and run instructions:

```
[trainee@daffodil c201]$ gcc -c hello.c
[trainee@daffodil c201]$ gcc -o hello hello.o
[trainee@daffodil c201]$ ./hello
This is a C program
Another line of program
[trainee@daffodil c201]$
```

Makefile for hello.c

```
# makefile for C version of "hello world"

hello:  hello.o
        gcc -o hello hello.o

hello.o:        hello.c
        gcc -c hello.c
```

Using the makefile to compile and run:

```
[trainee@daffodil c201]$ make hello
gcc -c hello.c
gcc -o hello hello.o
[trainee@daffodil c201]$ ./hello
This is a C program
Another line of program
[trainee@daffodil c201]$
```

4

A First Practical Program

Variables, types and numeric types. Float and ints. Short long and double. Calculations. Basic use of printf.

4.1 Variables and arithmetic in C

Example adder.c

```
/*
# Variables and arithmetic in C
*/

int main() {

        /* Variables declare at start of function */

        int children;
        int adults;

        /* Multiple variables can be declared and init'd */

        float total;
        float child_price = 3.50, adult_price = 8.00;

        children = 5;
        adults = 7;

        /* Standard "BODMAS" rules */

        total = children * child_price +
                        adults * adult_price;

        /* Formatted printing - a float, total of 8 columns
        wide including 2 figures after the decimal point */

        printf("The total price is %8.2f pounds\n",total);
        }
```

Compile and run:

```
[trainee@daffodil c202]$ make adder
gcc -c adder.c
gcc -o adder adder.o
[trainee@daffodil c202]$ ./adder
The total price is    73.50 pounds
[trainee@daffodil c202]$
```

4.2 Temperature conversions

Example tconv.c

```
/*
#%% Temperature conversions
*/

int main() {

        float units_in, units_out;

        /* Read a user input (better ways later!) */

        printf ("Please enter temperature ");
scanf("%f",&units_in);

/* Use of brackets to change precedence */

units_out = (units_in - 32.0) / 9.0 * 5.0;

printf("Your input was %.2f degrees F\n",units_in);
printf("Which converts to %.2f degrees C\n",units_out);

/* Tell calling program that you worked OK */

return 0;

}
```

Compile and run:

```
[trainee@daffodil c202]$ make tconv
gcc -c tconv.c
gcc -o tconv tconv.o
[trainee@daffodil c202]$ ./tconv
Please enter temperature 55
Your input was 55.00 degrees F
Which converts to 12.78 degrees C
[trainee@daffodil c202]$
```

4.3 Testing number ranges in C

Example nranges.c

```
/*
Testing number ranges in C
Bring in a header file at compile time
*/

#include <limits.h>

int main() {

        /* Different types of integers */

        int val1, val1a, val1b;
        short int val2, val2a, val2b;
        long int val3;
        unsigned short int val4;
        unsigned int val5;
        unsigned long int val6;

        /* What are their ranges? and sizes */

        printf ("Range of int is %d to %d and it takes up %d bytes\n",
                                INT_MIN, INT_MAX, sizeof(val1));
        printf ("Range of short is %d to %d and it takes up %d bytes\n",
                                SHRT_MIN, SHRT_MAX, sizeof(short int));
        printf ("Range of long is %d to %d and it takes up %d bytes\n",
                                LONG_MIN, LONG_MAX, sizeof(val3));

        val1 = val2 = 20000;
        val1a = val2a = 25000;

        val1b = val1 + val1a;
        val2b = val2 + val2a;

        /* Need to be careful of overflow - look at this! */

        printf ("Totals are %d and %d\n",val1b, val2b);

        /* Type converted early on */

        val3 = val2 + val2a;
        printf ("Long total %d\n",val3);

        /* Calculation result forced to be short */

        val3 = (short)(val2 + val2a);
        printf ("Long total %d\n",val3);

        return 0;

        }
```

Compile and run:

```
[trainee@daffodil c202]$ make nranges
gcc -c nranges.c
gcc -o nranges nranges.o
[trainee@daffodil c202]$ ./nranges
Range of int is -2147483648 to 2147483647 and it takes up 4 bytes
Range of short is -32768 to 32767 and it takes up 2 bytes
Range of long is -2147483648 to 2147483647 and it takes up 4 bytes
Totals are 45000 and -20536
Long total 45000
Long total -20536
[trainee@daffodil c202]$
```

4.4 Makefile

Makefile for C202 examples

```
#%% makefile for module C202

# all target (also first target) - make all executables
# clean target - cleans up all intermediate files
#      note use of @ to supress output reports

all:    adder tconv nranges

clean:
        @rm -f *.o
        @rm -f adder tconv nranges

adder:  adder.o
        gcc -o adder adder.o

adder.o:        adder.c
        gcc -c adder.c

tconv:  tconv.o
        gcc -o tconv tconv.o

tconv.o:        tconv.c
        gcc -c tconv.c

nranges:        nranges.o
        gcc -o nranges nranges.o

nranges.o:      nranges.c
        gcc -c nranges.c
```

Exercise

5

Conditionals and Loops

The if statement. Writing conditions in C. if, else, elseif. switch, case, default. while. ++ and += operator families. do while. for. break and continue. goto and labels. Layout of your blocks, and comments revisited.

5.1 The if statement

Example tall.c

```
/*
#%% Temperature conversions using if
*/

int main() {

        char units_in_type;
        float units_in_value;

        float faren, celcius, kelvin;

        /* Read a user input - temperature and units */

        printf ("Please enter temperature and units: ");
        scanf("%f",&units_in_value);
        scanf("%c",&units_in_type);

        /* What is that in Celcius */

        if (units_in_type == 'C') {
                celcius = units_in_value;
        } else if (units_in_type == 'F') {
                celcius = (units_in_value -32.0) / 9.0 * 5.0;
        } else if (units_in_type == 'K' ||
                units_in_type == 'A' ) {
                celcius = units_in_value - 273.1;
        } else {
                printf ("Units not understood\n");
                return (1);  /* Failure */
        }

        faren = celcius * 9.0 / 5.0 + 32.0;
        kelvin = celcius + 273.1;

        printf ("%.2f C = %.2f K = %.2f F\n",
                celcius, kelvin, faren);

        return 0;

        }
```

Compile and run:

```
[trainee@daffodil c203]$ make tall
gcc -c tall.c
gcc -o tall tall.o
[trainee@daffodil c203]$ ./tall
Please enter temperature and units: 45F
7.22 C = 280.32 K = 45.00 F
[trainee@daffodil c203]$ ./tall
Please enter temperature and units: 45C
45.00 C = 318.10 K = 113.00 F
[trainee@daffodil c203]$
```

5.2 Using switch

Example tall2.c

```
/*
#%% Temperature conversions using switch
*/

int main() {

        char units_in_type;
        float units_in_value;

        float faren, celcius, kelvin;

        /* Read a user input - temperature and units */

        printf ("Please enter temperature and units: ");
        scanf("%f",&units_in_value);
        scanf("%c",&units_in_type);

        /* What is that in Celcius */

        switch (units_in_type) {
        case 'C':
                celcius = units_in_value;
                break;
        case 'F':
                celcius = (units_in_value -32.0) / 9.0 * 5.0;
                break;
        case 'A':
        case 'K':
                celcius = units_in_value - 273.1;
                break;
        default:
                printf ("Units not understood\n");
                return (1);  /* Failure */
        }

        faren = celcius * 9.0 / 5.0 + 32.0;
        kelvin = celcius + 273.1;

        printf ("result: %.2f C = %.2f K = %.2f F\n",
                celcius, kelvin, faren);



        return 0;

        }
```

Compile and run:

```
[trainee@daffodil c203]$ make tall2
gcc -c tall2.c
gcc -o tall2 tall2.o
[trainee@daffodil c203]$ ./tall2
Please enter temperature and units: 45F
result: 7.22 C = 280.32 K = 45.00 F
```

```
[trainee@daffodil c203]$ ./tall2
Please enter temperature and units: 45C
result: 45.00 C = 318.10 K = 113.00 F
[trainee@daffodil c203]$ ./tall2
Please enter temperature and units: 45A
result: -228.10 C = 45.00 K = -378.58 F
[trainee@daffodil c203]$ ./tall2
Please enter temperature and units: 45K
result: -228.10 C = 45.00 K = -378.58 F
[trainee@daffodil c203]$ ./tall2
Please enter temperature and units: 45Z
Units not understood
[trainee@daffodil c203]$
```

5.3 In a while loop

Example tall3.c

```
/*
#%% Temperature conversions in a while loop
*/

int main() {

        float units_in_value;
        float faren;
        int running = 1;

        while (running) {
                printf ("Please enter temperature: ");
                scanf("%f",&units_in_value);
                faren = units_in_value * 9.0 / 5.0 + 32.0;
                printf ("result: %.2f C = %.2f F\n",
                        units_in_value, faren);
                if (units_in_value == 0.0) running = 0;
        }

        return 0;
}
```

Compile and run:

```
[trainee@daffodil c203]$ make tall3
gcc -c tall3.c
gcc -o tall3 tall3.o
[trainee@daffodil c203]$ ./tall3
Please enter temperature: 44
result: 44.00 C = 111.20 F
Please enter temperature: 55
result: 55.00 C = 131.00 F
Please enter temperature: 33
result: 33.00 C = 91.40 F
Please enter temperature: 0
result: 0.00 C = 32.00 F
[trainee@daffodil c203]$
```

5.4 Example with ++ and +=

Example tall4.c

```
/*
# Temperature conversions with ++ and +=
*/

int main() {

        float units_in_value;
        float faren, avg;
        int counter = 0;
        float grand = 0.0;

        while (counter < 10) {
                counter ++;
                printf ("Please enter temperature %d: ",counter);
                scanf("%f",&units_in_value);
                faren = units_in_value * 9.0 / 5.0 + 32.0;
                printf ("result: %.2f C = %.2f F\n",
                        units_in_value, faren);
                grand += faren;
        }
        avg = grand / (float) counter;
        printf ("average was %.2f F\n",avg);

        return 0;
}
```

Compile and run:

```
[trainee@daffodil c203]$ make tall4
gcc -c tall4.c
gcc -o tall4 tall4.o
[trainee@daffodil c203]$ ./tall4
Please enter temperature 1: 44
result: 44.00 C = 111.20 F
Please enter temperature 2: 43
result: 43.00 C = 109.40 F
Please enter temperature 3: 33
result: 33.00 C = 91.40 F
Please enter temperature 4: 34
result: 34.00 C = 93.20 F
Please enter temperature 5: 35
result: 35.00 C = 95.00 F
Please enter temperature 6: 38
result: 38.00 C = 100.40 F
Please enter temperature 7: 33
result: 33.00 C = 91.40 F
Please enter temperature 8: 20
result: 20.00 C = 68.00 F
Please enter temperature 9: 19
result: 19.00 C = 66.20 F
Please enter temperature 10: 22
result: 22.00 C = 71.60 F
average was 89.78 F
[trainee@daffodil c203]$
```

5.5 Makefile

```
#%% makefile for module C203

# all target (also first target) - make all executables
# clean target - cleans up all intermediate files
#     note use of @ to supress output reports

all:    tall tall2 tall3 tall4

clean:
        @rm -f *.o
        @rm -f tall tall2 tall3 tall4

tall:   tall.o
        gcc -o tall tall.o

tall.o: tall.c
        gcc -c tall.c

tall2:  tall2.o
        gcc -o tall2 tall2.o

tall2.o:        tall2.c
        gcc -c tall2.c

tall3:  tall3.o
        gcc -o tall3 tall3.o

tall3.o:        tall3.c
        gcc -c tall3.c

tall4:  tall4.o
        gcc -o tall4 tall4.o

tall4.o:        tall4.c
        gcc -c tall4.c
```

Exercise

6

Functions, Macros and Programs in Multiple Files

Defining functions. Parameters and return values. Function prototypes. Variable scope and global variables. Constants. Header files and other Macros.

6.1 Function definition and call

Example tcall.c

```
/*
#%% Function definition and call
*/

float ctof(float celcius) {
        float result;
        result = celcius * 9.0 / 5.0 + 32.0;
        return result;
        }

float average(float total, int count) {
        return (total / (float) count);
        }

int main() {

        float units_in_value;
        float faren, avg;
        int counter = 0;
        float grand = 0.0;

        while (counter < 10) {
                counter ++;
                printf ("Please enter temperature %d: ",counter);
                scanf("%f",&units_in_value);
                faren = ctof(units_in_value);
                printf ("result: %.2f C = %.2f F\n",
                        units_in_value, faren);
                grand += faren;
        }
        avg = average(grand,counter);
        printf ("average was %.2f F\n",avg);

        return 0;
}
```

Compile and run:

```
[trainee@daffodil c204]$ make tcall
gcc -c tcall.c
gcc -o tcall tcall.o
[trainee@daffodil c204]$ ./tcall
Please enter temperature 1: 4
result: 4.00 C = 39.20 F
Please enter temperature 2: 5
result: 5.00 C = 41.00 F
Please enter temperature 3: 6
result: 6.00 C = 42.80 F
Please enter temperature 4: 0
result: 0.00 C = 32.00 F
Please enter temperature 5: 2
result: 2.00 C = 35.60 F
```

```
Please enter temperature 6: 3
result: 3.00 C = 37.40 F
Please enter temperature 7: 4
result: 4.00 C = 39.20 F
Please enter temperature 8: 5
result: 5.00 C = 41.00 F
Please enter temperature 9:
3
result: 3.00 C = 37.40 F
Please enter temperature 10: 44
result: 44.00 C = 111.20 F
average was 45.68 F
[trainee@daffodil c204]$
```

Exercise

7

Arrays

Declaring and initialising an array. Techniques for iterating through an array.

7.1 Triangle numbers

Example arr1.c

```
/*
#%% Arrays - triangle numbers
*/

/* Compile time constant via the C preprocessor */

#define SIZE 10

int main() {


        int triang[SIZE];
        int k;

        /* Set up the array forward */

        triang[0] = 1;
        for (k=1; k<SIZE; k++) {
                triang[k] = triang[k-1]+k+1;
                }

        /* Print it out in reverse */

        for (k=SIZE-1; k>=0; k--) {
                printf ("Position %d value %d\n",k,triang[k]);
                }

}
```

Compile and run:

```
[trainee@daffodil c205]$ make arr1
gcc -c arr1.c
gcc -o arr1 arr1.o
[trainee@daffodil c205]$ ./arr1
Position 9 value 55
Position 8 value 45
Position 7 value 36
Position 6 value 28
Position 5 value 21
Position 4 value 15
Position 3 value 10
Position 2 value 6
Position 1 value 3
Position 0 value 1
[trainee@daffodil c205]$
```

7.2 Triangle numbers with error

```
Example arrbad.c

/*
#%% Arrays - triangle numbers WITH ERROR
*/

#define SIZE 10

int main() {


        int triang[SIZE];
        int k;

        triang[0] = 1;
        for (k=1; k<SIZE; k++) {
                triang[k] = triang[k-1]+k+1;
                }

        /* Print it out in reverse */
        /* Goes one too far! */

        for (k=SIZE; k>=0; k--) {
                printf ("Position %d value %d\n",k,triang[k]);
                }

}
```

Compile and run:

```
[trainee@daffodil c205]$ make arrbad
gcc -c arrbad.c
gcc -o arrbad arrbad.o
[trainee@daffodil c205]$ ./arrbad
Position 10 value 134517984
Position 9 value 55
Position 8 value 45
Position 7 value 36
Position 6 value 28
Position 5 value 21
Position 4 value 15
Position 3 value 10
Position 2 value 6
Position 1 value 3
Position 0 value 1
[trainee@daffodil c205]$
```

7.3 Arrays — days and months

Example morearr.c

```
/*
#%% Arrays - days and months
*/

int main() {

        /* Set up and initialise an array */
        int months[12] = {31,28,31,30,31,30,31,31,30,31,30,31};
        /* Two dimensional array */
        int dayno[12][31];

        int mno,dno;
        int dref = 0;
        int j,k;

        /* Set up 2D array */
        for (k=0; k<12; k++) {
                for (j=0; j<months[k]; j++) {
                        dayno[k][j] = ++dref;
                }
        }

        /* Look up in 2D array */
        while (1) {
                printf("Month no (0 to exit): ");
                scanf("%d",&mno);
                /* break to get out of loop */
                if (mno == 0) break;
                printf("Day no: ");
                scanf("%d",&dno);
                printf("That is day %d of the year\n",
                        dayno[mno-1][dno-1]);
        }
        return (0);
}
```

Compile and run

```
[trainee@daffodil c205]$ make morearr
gcc -c morearr.c
gcc -o morearr morearr.o
[trainee@daffodil c205]$ ./morearr
Month no (0 to exit): 4
Day no: 5
That is day 95 of the year
Month no (0 to exit): 1
Day no: 1
That is day 1 of the year
Month no (0 to exit): 12
Day no: 31
That is day 365 of the year
Month no (0 to exit): 11
Day no: 31
That is day -18419628 of the year
Month no (0 to exit): 0
[trainee@daffodil c205]$
```

7.4 Makefile

Makefile for module C205

```
# makefile for module C205

# all target (also first target) - make all executables
# clean target - cleans up all intermediate files
#     note use of @ to supress output reports

all:    arr1 arrbad morearr

clean:
        @rm -f *.o
        @rm -f arr1 arrbad morearr

arr1:   arr1.o
        gcc -o arr1 arr1.o

arr1.o: arr1.c
        gcc -c arr1.c

arrbad: arrbad.o
        gcc -o arrbad arrbad.o

arrbad.o:       arrbad.c
        gcc -c arrbad.c

morearr:        morearr.o
        gcc -o morearr morearr.o

morearr.o:      morearr.c
        gcc -c morearr.c
```

Exercise

8

Character Strings

Defining string variables. String manipulation functions.

8.1 Characters and strings in C

Example ST!1.c

```
/*
# Characters and strings in C
*/

int main() {

        char capital = 'A';
        char *company = "Well House Consultants";

        printf ("The letter is %c.\n",capital);
        printf ("The company is %s.\n",company);

        char *oldcompany = company;
        capital+=5;  /* Move up the alphabet */
        company+=5;  /* Move up the string */
        printf ("The letter is %c.\n",capital);
        printf ("The company is %s.\n",company);
        printf ("The company was %s.\n",oldcompany);

        /* Following can be dangerous. */
        /* Under modern / gcc grabs more memory for new string
        and places it at a new location */
        company = "Well House Manor and Consultants";
        printf ("The letter is %c.\n",capital);
        printf ("The company is %s.\n",company);
        printf ("The company was %s.\n",oldcompany);

        return 0;
}
```

Compile and run:

```
[trainee@daffodil c206]$ make sti1
gcc -c sti1.c
gcc -o sti1 sti1.o
[trainee@daffodil c206]$ ./sti1
The letter is A.
The company is Well House Consultants.
The letter is F.
The company is House Consultants.
The company was Well House Consultants.
The letter is F.
The company is Well House Manor and Consultants.
The company was Well House Consultants.
[trainee@daffodil c206]$
```

Exercise

9

Pointers and References

Declaring and using a pointer. Pointer arithmetic. Passing pointers. The use of pointers.

9.1 Pointers

Example ptr1.c

```
/*
#%% Temperature conversions - pointers
*/

/* & = 'address of' and * = 'contnts of' */

int main() {

        float units_in, units_out;

        /* One way */

        printf ("Please enter temperature ");
        scanf("%f",&units_in);
        units_out = (units_in - 32.0) / 9.0 * 5.0;
        printf("Your input was %.2f degrees F\n",units_in);
        printf("Which converts to %.2f degrees C\n",units_out);

        /* Illustrative alternative */

        float * inpointer;
        inpointer = &units_in;
        /* Note that inpointer and &units_in are now synonymous */
        /* as are *inpointer and units_in */
        printf ("Please enter another temperature ");
        scanf("%f",inpointer);
        units_out = (*inpointer - 32.0) / 9.0 * 5.0;
        printf("Your input was %.2f degrees F\n",*inpointer);
        printf("Which converts to %.2f degrees C\n",units_out);

        return 0;

        }
```

Compile and run:

```
[trainee@daffodil c207]$ make ptr1
gcc -c ptr1.c
cc   ptr1.o   -o ptr1
[trainee@daffodil c207]$ ./ptr1
Please enter temperature 44
Your input was 44.00 degrees F
Which converts to 6.67 degrees C
Please enter another temperature 55
Your input was 55.00 degrees F
Which converts to 12.78 degrees C
[trainee@daffodil c207]$
```

10

Programming Techniques and Tools

The Preprocessor and Macros. Libraries. Enumerators. Using command line parameters and environment variables.

10.1 Command line arguments and the environment

Example cle:

```
/*
#%% Command line arguments and the environment
*/

#include <stdlib.h>
#include <string.h>

int main(int argc, char **argv) {

        /* parameters are an array of character arrays (strings) */

        while (argc--) {
                printf ("Parameter %s\n",*argv++);
                }

        char *zat;
        zat = getenv("PATH");
        printf ("PATH %s\n",zat);

        /* Let's use some string handling on that */

        char *pathel;

        pathel = strtok(zat,":");
        do {
                printf ("element %s\n",pathel);
                }
        while (pathel = strtok(NULL,":")) ;

        return 0;

        }
```

Compile and run:

```
[trainee@daffodil c208]$ make cle
gcc -c cle.c
cc   cle.o   -o cle
[trainee@daffodil c208]$ ./cle pieces "of cheese" and bread
Parameter ./cle
Parameter pieces
Parameter of cheese
Parameter and
Parameter bread
PATH /usr/local/java/bin:/usr/kerberos/bin:/usr/local/bin:/bin:/usr/
bin:/usr/X11R6/bin:/home/trainee/bin
element /usr/local/java/bin
element /usr/kerberos/bin
element /usr/local/bin
element /bin
element /usr/bin
element /usr/X11R6/bin
element /home/trainee/bin
[trainee@daffodil c208]$
```

Exercise

11

Structures & Unions

If you want to hold a collection of different values of different types in a variable, you may use a structure. If you want the various elements to share memory, use a union.

11.1 Structures and Unions - Introduction

In C you can hold a whole series of variables of the same type in an Array. There are, however, times that you want to hold a number of variables of different types in association with each other or under the same name, and that's where you will use a structure.

A **typedef** defines a structure type ~ an additional variable type ~ which you can then use with a dot notation. Here's a definition of a structure of type person:

```
typedef struct {
     char forename[20];
     char surname[20];
     float age;
     int childcount;
     } person;
```

and you can then declare variables of that type, and make use of them:

```
   person jimmy
```

and

```
  strcpy(jimmy.surname,"Conway");
  jimmy.age = 32.0;
```

A First Complete Example

```
/* The mechanism of a structure */

/* Define a type "person" to include
        - forename and surname
        - age and childcount */

typedef struct {
     char forename[20];
     char surname[20];
     float age;
     int childcount;
     } person;

int main (int argc, int *argv) {

/* Create two persons */
        person jimmy, flossie;

        float avage;

/* Fill up each of the persons with information */
        strcpy(jimmy.forename,"James");
        strcpy(jimmy.surname,"Conway");
        jimmy.age = 32.0;
        jimmy.childcount = 2;

        strcpy(flossie.forename,"Florence");
        strcpy(flossie.surname,"Conway");
        flossie.age = 21.0;
        flossie.childcount = 2;

/* Calculate the average age of the persons */
        avage = (flossie.age + jimmy.age) / 2.0;

/* Print out the persons names and the average age */
        printf ("About %s and %s ...\n",flossie.forename,
                jimmy.forename);
        printf ("Their average age is %.2f\n",avage);

        return 0;

        }
/* st1.c

Sample output
[graham@saturday c]$ ./st1
About Florence and James ...
Their average age is 26.50
[graham@saturday c]$

*/
```

An array of structures

```
/*

Example of setting up and using an array
of structures */

#define MAXTRAINS 3
typedef struct {
        char *descript;
        char *topscode;
        int cars;
        float topspeed; } railwaytrain;
void reportfleet(railwaytrain *toc, int stockcount) {
        int k;
        railwaytrain current;
        for (k = 0; k<stockcount; k++) {
                current = toc[k];
                printf("class %s, length %d\n",
                        toc[k].topscode,
                        toc[k].cars);
                /* Alternative notation */
                printf("from %s, runs at up to %.1f mph\n",
                        current.descript,
                        current.topspeed);
                }
        }
int main() {
        int k; int traincount = MAXTRAINS;
        railwaytrain transwilts[MAXTRAINS];
        for (k = 0; k<MAXTRAINS; k++) {
                transwilts[k].cars = 2;
                transwilts[k].topscode = "159/1";
                transwilts[k].descript = "Angel Trains";
                transwilts[k].topspeed = 90.0; }
        transwilts[2].cars = 1;
        transwilts[2].topscode = "153";
        transwilts[2].descript = "Porterbrook";
        reportfleet(transwilts,traincount);
        }

/* Results ..

[trainee@daisy cd07]$ ./trains
class 159/1, length 2
from Angel Trains, runs at up to 90.0 mph
class 159/1, length 2
from Angel Trains, runs at up to 90.0 mph
class 153, length 1
from Porterbrook, runs at up to 90.0 mph
[trainee@daisy cd07]$

*/
```

11.2 Structure Pointers

You can use pointers to structures in just the same way that you use pointers to other variables, but the syntax gets difficult to follow and hard to maintain, so you are provide with an extra syntax -> which provides a more readable piece of code.

```
(*current).descript
```

becomes

```
current->descript
```

Here's a complete example showing that syntax in use

```
/*

The -> operator is used as a pointer to a
structure, whereas the . operator is used
to access directly. Both are illustrated
in this example

*/

#define MAXTRAINS 3
typedef struct {
        char *descript;
        char *topscode;
        int cars;
        float topspeed; } railwaytrain;
void reportfleet(railwaytrain *toc, int stockcount) {
        int k;
        railwaytrain *current;
        for (k = 0; k<stockcount; k++) {
                current = &(toc[k]);
                printf("class %s, length %d\n",
                        toc[k].topscode,
                        toc[k].cars);
                /* Alternative notation */
                /* -> replaces (*xxx). - syntatic icing! */
                /* Very useful when you pass addresses around */
                printf("from %s, runs at up to %.1f mph\n",
                        (*current).descript,
                        current->topspeed);
                }
        }
int main() {
        int k; int traincount = MAXTRAINS;
        railwaytrain transwilts[MAXTRAINS];
        for (k = 0; k<MAXTRAINS; k++) {
                transwilts[k].cars = 2;
                transwilts[k].topscode = "159/1";
                transwilts[k].descript = "Angel Trains";
                transwilts[k].topspeed = 90.0; }
        transwilts[2].cars = 1;
        transwilts[2].topscode = "153";
        transwilts[2].descript = "Porterbrook";
        reportfleet(transwilts,traincount);
        }

/* Result of running

[trainee@daisy cd07]$ ./mytrains
class 159/1, length 2
from Angel Trains, runs at up to 90.0 mph
class 159/1, length 2
from Angel Trains, runs at up to 90.0 mph
class 153, length 1
from Porterbrook, runs at up to 90.0 mph
[trainee@daisy cd07]$

*/
```

11.3 Unions

Sometimes, you'll only want to hold some elements of your structure in specific circumstances. A good example is where you have a whole lot of similar things you're describing, but only a limited range of elements applied to each.

A very good example in real system programming is an X Windows event ~ each has a timestamp and a sequence number, and an event type, and each has a target window. But only some of them have X and Y co-ordinates, only some of them have keypresses and statuses, and so on. One possible way of holding such events would be a sparsely populated structure, but in practice this would be very wasteful of memory, so a union is used.

Here's a small example to introduce you to unions - which are typedefs again with the critical word union saying that a memory location is shared between two variables rather than being successive location. *Beware - you need to be careful when programming with unions!*

```
/*
A Union is like a structure, except that each
element shares the same memory. Thus in this
example, the coords and about members overlap.

Note that the setting of about[2] to 'X' CORRUPTS
the float in this demonstration.
```

```
In a practical use, a variable such as un_type
(provided but not used in this example) would be
set up to indicate which particular use is being
made of the union

*/
typedef union {
        float coords[3];
        char about[20];
        } assocdata;

typedef struct {
        char *descript;
        int un_type;
        assocdata alsostuff;
        } leaf;

int main() {
        leaf oak[3];
        int i;

        printf ("Hello World\n");

        for (i=0; i<3; i++) {
                oak[i].descript = "A Greeting";
                oak[i].un_type = 1;
                oak[i].alsostuff.coords[0] = 3.14;
                }
        oak[2].alsostuff.about[2] = 'X';

        for (i=0; i<3; i++) {
                printf("%s\n",oak[i].descript);
                printf("%5.2f\n",oak[i].alsostuff.coords[0]);
                }



        }

/* Sample of output from this program

[trainee@daisy cd07]$ ./union
Hello World
A Greeting
 3.14
A Greeting
 3.14
A Greeting
 3.39
[trainee@daisy cd07]$

*/
```

11.4 Practical use of Unions

Our first union example was too small to be of practical use, but it did show you some of the principles. Here's a longer example in which we have a shop with type types of products, one of which sells in integer units (diesel motor bikes), and the other which sells in kgs (apples), and you can see how the code works for this, sharing memory for a float and an int , and with the programmer making sure that the right value is used at the right time.

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


typedef union {
        int units;
        float kgs;
        } amount ;

typedef struct {
        char selling[15];
        float unitprice;
        int unittype;
        amount howmuch;
        } product;

int main() {

        product dieselmotorbike;
        product apples;
        product * myebaystore[2];

        int nitems = 2; int i;

        strcpy(dieselmotorbike.selling,"A Diesel Motor Cycle");
        dieselmotorbike.unitprice = 5488.00;
        dieselmotorbike.unittype = 1;
        dieselmotorbike.howmuch.units = 4;

        strcpy(apples.selling,"Granny duBois");
        apples.unitprice = 0.78;
        apples.unittype = 2;
        apples.howmuch.kgs = 0.5;

        myebaystore[0] = &dieselmotorbike;
        myebaystore[1] = &apples;

        for (i=0; i<nitems; i++) {
                printf("\n%s\n",myebaystore[i]->selling);
                switch (myebaystore[i]->unittype) {
                case 1:
                        printf("We have %d units for sale\n",
                                myebaystore[i]->howmuch.units);
                        break;
                case 2:
                        printf("We have %f kgs for sale\n",
                                myebaystore[i]->howmuch.kgs);
                        break;
                }
        }
}
```

11.5 Towards Object Orientation

You'll notice that we're using structures and unions to hold information about different types of thing ("objects") - and indeed Structure and Unions are the first steps in C towards object oriented programming. In C you need to do a lot of the hard work yourself, but you do have the tools.

When C extends to C++, the language provides a lot more of the tools, and relieves the programmer from some of the detailed work with keeping note of how unions are being used (for example). But even C++ is somewhat long-winded in its object oriented coding, and you may find that -- away from system programming -- Java, Perl or Python may be more suitable for many tasks.

12

File Handling in C

C supports file handling on a byte-by-byte basis. And there are also libraries and functions that will work line by line. Further libraries allow directory structures to be traversed, and file statuses checked.

12.1 File Handling in C

Being a system programming language, C offers you a very wide range of low-level handlers, using included headers and libraries such as fnctl and ioctl which allow you to manipulate echo conditions, read from a device without awaiting a new line, test whether input is available without waiting for input if it is not, and so on. Many of these are specialised uses; please ask your tutor if you would like some pointers towards them.

In this module, we look at examples of file reading at a byte (binary, low) level, then at a line-by-line level (ascii, higher level) and at enquiring of the file system.

12.2 Binary and low-level file handling

This example reads a specific number of bytes at a time from a file. It's what you'll do with random access records of fixed size, in association with other functions such as **fseek** and **ftell**.

```
/*

A file opened with 'open' is accessed in
a binary mode ... "read X bytes" stuff.

Note that the buffer is one character
longer that the maximum read to allow
space for us to add a null and turn it
into a string */


#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#include <stdlib.h>
#include <stdio.h>

int main() {

        int fh;
        char buffer[65];
        int gotten;

        fh = open("abc.txt",O_RDONLY);
        printf ("File handle %d\n",fh);
        while (gotten = read(fh,buffer,64)) {
                buffer[gotten] = '\0';
                printf("******%s",buffer);
                }
        }

/* Sample output

[trainee@daisy cd07]$ ./filesinc
File handle 3
******Tom Southampton
Aaron Weymouth
Mark Derby
Graham Melksham
Tom So******uthampton
```

```
Aaron Weymouth
Lisa Melksham
Mark Derby
Graham Melksha******m
Tom Southampton
Aaron Weymouth
Mr Christmas Melksham
Mark Derb******y
Graham Melksham
Tom Southampton
Aaron Weymouth
Mark Derby
Grah******am Melksham
[trainee@daisy cd07]$

*/
```

12.3 Higher level (line-by-line) file handling

In this example, we use fopen rather than open, and fgets rather than read, to handle our data line by line.

```
/*
fopen (as opposed to open) lets you read
files line by line - you still need to give
a maximum line length though.

Note that fopen returns a FILE whereas open
returned an int

*/

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

int main() {

        FILE *fh, *fh2;
        char buffer[65];
        int gotten;

        fh = fopen("abc.txt","r");
        fh2 = fopen("def.txt","w");

        printf ("File handle %p\n",fh);
        while (fgets(buffer,64,fh)) {
                printf("******%s",buffer);
                if (strstr(buffer,"Melksham")) {
                        fprintf(fh2,"%s",buffer);
                        }
                }
        fclose(fh2);
        }

/* Sample output
```

```
[trainee@daisy cd07]$ ./ffiles
File handle 0x9331008
******Tom Southampton
******Aaron Weymouth
******Mark Derby
******Graham Melksham
******Tom Southampton
******Aaron Weymouth
******Lisa Melksham
******Mark Derby
******Graham Melksham
******Tom Southampton
******Aaron Weymouth
******Mr Christmas Melksham
******Mark Derby
******Graham Melksham
******Tom Southampton
******Aaron Weymouth
******Mark Derby
******Graham Melksham
[trainee@daisy cd07]$

*/
```

Note that there remains a "number of bytes" parameter in the read function. This is a maximum number, as you need to be very careful in C not to overrun the char array into which you're reading. The function returns the number of bytes actually read, **and an extra null terminator is added to the string to that the receiving array need to be one longer then the maximum string length**!

12.4 Parsing directory structures in C

Using a structure of type DIR that's defined in dirent.h, you can parse directories. Here's an example ...

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <dirent.h>

#include <sys/stat.h>
#include <unistd.h>


// Very simple directory listing

int main () {

        DIR * target;
        struct dirent *about;
        struct stat myfile;

        target = opendir(".");

    while (about = readdir(target)) {
        printf("\n%s\n",about->d_name);
        // Need to use strcat to add directory name on front
// Only works here 'cos it's current directory
// lstat for links

lstat(about->d_name,&myfile);

printf("File is %d bytes\n",myfile.st_size);
printf("File permissions %o\n",myfile.st_mode);
if (myfile.st_mode & 16384)
        printf("WE GOT A DIRECORY, MATE!\n");
if (myfile.st_mode & 8192)
        printf("OH GOWD - HE'S USING LINKS!\n");
        }

}
```

13

The C Preprocessor

One element of the C language is the preprocessor, which allows headers, macros and settings to be loaded before the main compile.

13.1 The C Preprocessor

You'll often wish to include the same structure definitions and other constants in file after file of C sources, and rather than repeat the code many times over (with the maintenance implications), you'll use the C preprocessor.

Lines that start with the hash character in C are not comments as they are in most other languages ~ they are preprocessor directives. The C compiler "gcc" runs as follows:

- C preprocessor (cpp)
- C Compiler
- Assembler (as)
- Loader (ld)

and it's during the cpp phase that the preprocessor directives are handled.

As well as header files that you provide, the C preprocessor is used to load standard header files, and also compile time switches.

Examples of some c preprocessor directives

```
* Use of cpp commands */

/* include from central libraries */
#include <stdio.h>

/* include from current directory */
#include "abc.h"

/* "If SNOW is already defined, define RAIN."
It could be defined in abc.h, or as a compile
time switch with -DSNOW=10 or something like that */

#ifdef SNOW
#define RAIN 10
#endif

/* "If SNOW is NOT defined, define RAID and also
define SNOW. Note the use of brackets around the
negative number; this is because th C Preprocessor
does an exact character substition when ity runs
for these define constants, and the brackets ensure
that you don't end up with two signs right up against
each other, giving a false -- operator or worse! */

#ifndef SNOW
#define RAIN 20
#define SNOW (-5)
#endif

int main () {
                int wet, verywet;

                wet = 5 + SNOW;
                verywet = 5 + RAIN;

                printf("%d %d ...\n",wet, verywet);
                }
```

Here's the include file

```
// #define SNOW (-10)

// Reminder - may use command line to set defines too!

/*
[trainee@campion cmilo]$ gcc -o cppp -DSNOW=123 cpp.c
[trainee@campion cmilo]$ ./cppp
*/
```

Note that there's an ability to define parameters for the compile on the command line.

Also note that we have defined a negative number in round brackets. That's because the definition is literally a string replacement, and as such the minus sign in front of a negative number can lead to code issues:

```
this=that-SNOW
```

would interpret at

```
this=that--10
```

whereas you'll need

```
this=that-(-10)
```

13.2 Is C a standard or portable language?

No, but it can be made so using the preprocessor. Let me explain.

C is a low-level language. You are not guaranteed how many bytes or what number range an int on a long will have on any particular system, nor which libraries will be available to you.

For some programming tasks, you can use common code, but for others you'll need code that varies to take account of these issues. And that's where the preprocessor comes in with -D parameters and ifdefs. And indeed, you'll find another stage on many pieces of open source software.

Let's take the Apache httpd web server, which you may want to build with.

Firstly, you'll run **./configure** or something similar ~ a script to discover just how your build should be done, taking into account all the differences between different operating systems and versions of C. It can look like this:

```
[trainee@trowbridge httpd-2.0.63]$ ./configure --enable-proxy --enable-
rewrite --enable-proxy-http
checking for chosen layout... Apache
checking for working mkdir -p... yes
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking target system type... i686-pc-linux-gnu

Configuring Apache Portable Runtime library ...

checking for APR... reconfig
configuring package in srclib/apr now
checking build system type... i686-pc-linux-gnu
checking host system type... i686-pc-linux-gnu
checking target system type... i686-pc-linux-gnu
Configuring APR library
Platform: i686-pc-linux-gnu
checking for working mkdir -p... yes
```

```
APR Version: 0.9.17
checking for chosen layout... apr
checking for gcc... gcc
checking for C compiler default output file name... a.out
checking whether the C compiler works... yes
checking whether we are cross compiling... no
checking for suffix of executables...
checking for suffix of object files... o
checking whether we are using the GNU C compiler... yes
checking whether gcc accepts -g... yes

checking for string.h... yes
checking for memory.h... yes
checking for strings.h... yes
checking for inttypes.h... yes
checking for stdint.h... yes
checking for unistd.h... yes
checking minix/config.h usability... no
checking minix/config.h presence... no
checking for minix/config.h... no
checking whether system uses EBCDIC... no
performing libtool configuration...
checking for a sed that does not truncate output... /bin/sed
checking for ld used by gcc... /usr/bin/ld

creating os/Makefile
creating os/unix/Makefile
creating server/Makefile
creating support/Makefile
creating srclib/pcre/Makefile
creating test/Makefile
config.status: creating docs/conf/httpd-std.conf
config.status: creating docs/conf/ssl-std.conf
config.status: creating include/ap_config_layout.h
config.status: creating support/apxs
config.status: creating support/apachectl
config.status: creating support/dbmmanage
config.status: creating support/envvars-std
config.status: creating support/log_server_status
config.status: creating support/logresolve.pl
config.status: creating support/phf_abuse_log.cgi
config.status: creating support/split-logfile
config.status: creating build/rules.mk
config.status: creating build/pkg/pkginfo
config.status: creating include/ap_config_auto.h
config.status: executing default commands
[trainee@trowbridge httpd-2.0.63]$
```

The **makefile**s which we have seen earlier include all the directives needed to alert the C preprocessor, so that when you do your **make** they will be taken into account. Here's some of the output from **make**.

```
/bin/sh /home/trainee/httpd-2.0.63/srclib/apr/libtool --silent --mode=compile gcc -g -O2 -pthread   -
DHAVE_CONFIG_H -DLINUX=2 -D_REENTRANT -D_GNU_SOURCE  -I../include -I../include/arch/unix  -c apr_cpystrn.c
&& touch apr_cpystrn.lo
```

and so on through to ...

```
/home/trainee/httpd-2.0.63/srclib/apr/libtool --silent --mode=link gcc  -g -O2 -pthread    -DLINUX=2 -
D_REENTRANT -D_GNU_SOURCE -DAP_HAVE_DESIGNATED_INITIALIZER   -I/home/trainee/httpd-2.0.63/srclib/apr/
include -I/home/trainee/httpd-2.0.63/srclib/apr-util/include -I. -I/home/trainee/httpd-2.0.63/os/unix -I/
home/trainee/httpd-2.0.63/server/mpm/prefork -I/home/trainee/httpd-2.0.63/modules/http -I/home/trainee/
httpd-2.0.63/modules/filters -I/home/trainee/httpd-2.0.63/modules/proxy -I/home/trainee/httpd-2.0.63/
include -I/home/trainee/httpd-2.0.63/modules/generators -I/home/trainee/httpd-2.0.63/modules/dav/main -
export-dynamic    -o httpd  modules.lo  modules/aaa/mod_access.la modules/aaa/mod_auth.la modules/filters/
mod_include.la modules/loggers/mod_log_config.la modules/metadata/mod_env.la modules/metadata/
mod_setenvif.la modules/proxy/mod_proxy.la modules/proxy/mod_proxy_connect.la modules/proxy/
mod_proxy_ftp.la modules/proxy/mod_proxy_http.la modules/http/mod_http.la modules/http/mod_mime.la
modules/generators/mod_status.la modules/generators/mod_autoindex.la modules/generators/mod_asis.la
modules/generators/mod_cgi.la modules/mappers/mod_negotiation.la modules/mappers/mod_dir.la modules/
mappers/mod_imap.la modules/mappers/mod_actions.la modules/mappers/mod_userdir.la modules/mappers/
mod_alias.la modules/mappers/mod_rewrite.la modules/mappers/mod_so.la server/mpm/prefork/libprefork.la
server/libmain.la os/unix/libos.la  /home/trainee/httpd-2.0.63/srclib/pcre/libpcre.la /home/trainee/httpd-
2.0.63/srclib/apr-util/libaprutil-0.la -lexpat /home/trainee/httpd-2.0.63/srclib/apr/libapr-0.la -lrt -lm
-lcrypt -lnsl -lpthread -ldl
```

14

Memory Allocation in C

C uses a static memory model, which means that you have to decide how big your arrays are when you compile your program. But the memory management functions we cover in this module give you a dynamic alternative, using pointers to a memory area known as the heap.

14.1 Dynamic Memory Allocation

In C, you give fixed sizes for arrays at compile time ... which means you either have to know ahead of time how big something will be, or allow enough space for the worst case scenario.

Rather than using arrays, you can use pointers to the memory heap, and in this way, you can use functions to give you dynamic memory allocation:

malloc	allocate a specific memoryt block
calloc	allocate an array of specific memory blocks
realloc	expand or contract a calloced block
free	release a malloced block
cfree	release a coalloced block

Here's a typical example program which grabs memory dynamically to load a data file into memory.

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


int main() {

        FILE * infile;
        char line[121];
        char ** info = NULL;
        int llen;
        int counter = 0;

        int backdown;

        infile = fopen("coldata.txt","r");

        while (fgets(line,120,infile)) {

        // Allocate memory for pointer to line just added
                info = realloc(info,(counter+1) * sizeof(char *));
        // And allocate memory for that line itself!
                llen = strlen(line);
                info[counter] = calloc(sizeof(char),llen+1);
        // Copy the line just read into that memory
                strcpy(info[counter],line);

                printf("%d characters in line %d \n",llen,counter);
                counter++;
        }

        for (backdown = counter-1; backdown >= 0; backdown--) {
                printf("%d: %s",backdown,info[backdown]);
                }
        }
```

15

Introduction to C++

There's a very wide variety of programming languages that are in common usage today. What is C++ and how does it fit in with the others?

15.1 What is C++?

C++ is an object oriented, compiled, low-level, reasonably portable programming language. It's based on C, and it maintains C's syntax. If you're going to learn C++, you'll do very well to have a grounding in C first, or to take a course that covers the relevant C syntax as well as the C++ extensions.

Object Oriented

First and foremost, you'll find the headlines telling you that C++ is Object Oriented.

When you write your first program, you'll write a few lines of code to perform a task and it'll work well for you. As you develop onwards to more complex programs, you'll find that you're repeating sections of code (through cut and paste) and that it's a much better idea to name these sections or blocks of code so that you can simply call them up again by name. Fairly soon, you'll find yourself moving these blocks of code out into separate source files so that you can use the same logic elements from one program to the next.

Name blocks of code in this way give you:

- Code re-use to cut down on development and maintenance time.
- Consistency across applications that make your code more user-friendly and less liable to show up obscure bugs.

The approach of having named blocks of code right across your suite of applications, with each code block performing a specific single task, is known as structured programming. It's streets ahead of single-block code. HOWEVER:

- As your code grows further, you can end up with conflicts between blocks of code that happen to have been given the same name.
- You would prefer a system where you're encouraged to group all the bits of code that deal with one data type in one place. That way, you'll know you have just one area to update and maintain as your data formats or user requirements change.
- You would like a mechanism where you can easily say that one type of data and its blocks of code are based on another, and just describe the changes in a file.
- You would like to be able to forget what goes on at the low level of the code when you're writing at a higher level, andjust know what you need to put in, what the effects are and what you get out.

In turning your design inside out from a structured program by thinking of the data aspects first rather than the code aspects, it turns out you can have all these benefits and more. At the expense of a few surprises early on, as your project grows in size, it turns out to be very extensible under object orientation.

Compiled

C++ is a compiled language. You write a source code file using any editor that you find convenient and that can produce a plain text file output, and you save that file.

You run the source code file through a **compiler**, and that produces an **object** file which is a snippet of binary code suitable for your target computer architecture and operating system.

The object file itself is not a complete application. You then need to use a loader/ linker/taskbuilder to roll it in with all the other pieces of code, both yours and standard pieces from the library, before you come up with the final **executable** file.

Low Level

C++ is often described as a low-level language in that you have the full control as you code over how memory is allocated, and you can program down to a bit/byte/ register level if you wish. For some applications where performance is paramount, this is exactly what you want. For other applications where you're rushing to develop the code for use just a few times, C++ won't be the language for you unless you're reusing lots of existing code, and/or you're already very familiar with C++ but you're not skilled in other languages such as VB, Python or Perl.

Portability

C++ executable programs are not portable between operating systems/CPU types. And yet I said earlier on that it's "reasonably portable", didn't I?

The portability is at a source code level. It's possible to recompile the same source code under a different compiler, to generate object files and ultimately an executable file for different operating system and hardware combinations.

However, even at the source code level libraries do vary and you need to be careful as you develop code. You'll find that there's a great deal of open source code developed in C++, and that the first stage of installing it onto a target machine is to run a configuration script that looks around the machine and finds out all the compiler details, library details, and more, that it needs to get the configuration right.

So, C++ is portable. But it is not just a case of transferring source and running (as it would be with Perl or Python), nor is it simply a case of having the right virtual machine at hand to handle object files (as it would be with Java).

Based on C

The C language has been around for many years, and is the industry standard in which many operating systems have been developed. C is also the language in which many other languages are written. There days, however, actually writing in C has become something of a niche as most applications are now written using the superb tool sets that earlier C authors have been able to provide for us, anything from languages to servers to spread sheets to word processors with macro languages.

Being based on C, and maintaining compatibility with it, has made some of the elements of C++ quite complex. Decisions to include multiple inheritance and certain other features in C++ haven't helped to make it quick and easy to learn, and a number of other languages such as C# and Java have been developed by Microsoft and Sun respectively which take much of the intent of C++ but break the C compatibility, simplify, and provide languages which are a little more mainstream and much faster to develop.

15.2 What is C++ used for?

So what would I use C++ for?

It's used for device drivers, for low-level coding, for coding where run speeds are paramount even if the code development takes far longer than it would in some other languages. Applications such as scientific ones and big financial institution systems come to mind.

If you have a lot of existing C and/or C++ code, again it's sensible to continue on using it, and you'll also find C++ used for a very large number of commercial software products. Security against source code changes, distribution via an executable file, and lightning fast execution (if you take the time to write it to do that) are all good reasons behind this choice.

Finally, you'll often find C or C++ modules built in to other languages, providing some specific code pieces that aren't well suited to that other language. Amongst other languages that we train on, and which can embed C / C++, are Perl, Python, PHP and Tcl ... indeed on the Tcl course we spend a few minutes showing you how. These days, though, many of us use these embedded extra modules without actually writing them ourselves. C code tends to be very much "write once, use thousands of places" and is sensible when you look at the high development costs, even though the result in the end can be better.

15.3 Where do I obtain C++?

On our courses, we use the C++ compiler from the Gnu Compiler Collection (gcc) which is a standard part of the Open Source Linux distribution. The compiler's been around quite a while. It's fast, generally reliable, and something of an industry standard. And it's freely distributed under an open source license and comes with your Linux distribution.

Gcc is also available on Unix systems (Solaris, OSX, etc) and it can also be obtained for Windows. On Windows, though, many users prefer to purchase a C++ compiler such as Microsoft's Visual C++, which provides a full integrated IDE environment. Whilst such environments are great for your use back at your office, on a course that's designed to teach you the syntax of the language from the bottom up, they just add another layer of learning right at the start, and you'll find we're not using them on this course.

Another alternative product suite on Windows comes from Borland. They offer a complete IDE environment, and also a compiler without the IDE which you can download and install for free via:

http://www.borland.com/downloads/download_cbuilder.html

15.4 The compiler we□ll use

We're using gcc. You'll find it on the system that's in front of you for the course. Don't worry if the operating system happens to be an unfamiliar one, as we'll be using a simple point and click editor to do most of the work, and using the makefile system to avoid the need to keep repairing long and complex compile instructions.

15.5 A first C++ program

If you don't have a working knowledge of C, variable naming conventions, if and while structures, arrays, chars, etc, now would be a good time to for you to go off and learn some of that background; we're not doing an easy "hello world" here!

Source code, main program - file app1.cpp

```
#include <iostream.h>
#include "ir.inc"

/*

Gentle introduction to C++ - main application

*/

int main () {

        int b1y;
        int b2y;
        int diff = 0;

// intread is a function that's in another file
// that we'll load in with this application and
// probably with others too

        b1y = intread ("Enter year of first book ");
        b2y = intread ("Enter year of second book ");

        if (b1y < b2y) {
                cout << "First book is older\n" ;
                diff = b2y - b1y;
        } else if (b1y == b2y) {
                cout << "Books are of same age\n" ;
        } else {
                cout << "First book is newer\n" ;
                diff = b1y - b2y;
        }

        cout << "Age difference is " << diff << "\n" ;
        cout << "first book: " << b1y << "\n" ;
        cout << "second book: " << b2y << "\n" ;

        return 0;
        }
```

Header definition file ir.inc

You'll need this file too before you can compile the main program. Tt defines headers that are needed in all files that use functions such as **`intread`** which we'll be sharing later on between applications.

```
// Function prototypes

int intread(char * instring) ;
void copyright(char * instring);
```

Using these two files

You'll compile the source code in *app1.cpp* and create an object file called *app1.o* using the command:

```
g++ -Wno-deprecated -c app1.cpp
```

This call to **`g++`** with the **`-c`** option says "compile only", and the **`-W`** option is representative of the sort of standard compile options – of which there are many – that you may choose to use later.

Let's see some things in the source code:

```
#include <iostream.h>
#include "ir.inc"
```

You're loading the C++ standard IO stream headers (the use of **<** and **>** points to the standard library location) and the file *ir.inc* from your own directory in to the compiler's pre-processor. Lines that start with a **#** are acted on before the main compile stage. They are not comments in C or C++, and they do not have a ";" character terminator.

Comments in C++ can be either "C style – a `/*` to an `*/` and run over more than one line – or can be written from **`///`** through to the line end. It's **vital** that you comment your programs well ... which particular style you use is less important.

```
/*
Gentle introduction to C++ - main application
*/
```

Here's your main function definition, where your application will start. Notice that it should return an int value (please don't make it void, even if you find that works for you). Later, you could add argc/argv in there for command line parameters. At this point, we've also declared some variables and initialised one of them to 0.

```
int main () {
        int b1y;
        int b2y;
        int diff = 0;
```

Function calls to a function called **`intread`**; in this example, the function is declared in the header file *ir.h* (which is why we needed to have that **`include`** line earlier on). The author chose to use a function here so that it can be called up several times without code repetition, and can be shared between applications if required.

```
        b1y = intread ("Enter year of first book ");
        b2y = intread ("Enter year of second book ");
```

The following code – with the exception of **`cout`** – will look very much like standard C. You have all the usual calculations, conditionals, operators, etc available to you.

In C++, we usually use the **`cout`** output stream object to write to stdout, as we've done in this example. The **<<** operator appends to the stream, and we can do as many such appends as we wish all at the same time.

```
        if (b1y < b2y) {
                cout << "First book is older\n" ;
                diff = b2y - b1y;
        } else if (b1y == b2y) {
                cout << "Books are of same age\n" ;
        } else {
                cout << "First book is newer\n" ;
                diff = b1y - b2y;
        }
```

Finally, output the results and return a code of 0 to the calling shell to indicate success:

```
        cout << "Age difference is " << diff << "\n" ;
        cout << "first book: " << b1y << "\n" ;
        cout << "second book: " << b2y << "\n" ;

        return 0;
        }
```

Having run the compile, what then? We've just got an object file, a component that's no use without the other components too.

The other components

In this first simple example, we've only got one other file that we wrote that we need to load alongside the first object file in order to come up with an executable program. Here's the source, *ir.cpp*:

```
#include <iostream.h>
#include "ir.inc"

/*

This is an example of a file of functions that
could be used from multiple applications.
```

```
*/

int intread ( char * prompt) {
        int rval;
        cout << prompt;
        cin >> rval;
        return rval;
        }

void copyright (char *whom) {
        cout << "copyright " << whom << ", 2006"
                << endl;
        }
```

This defines the **intread** function which we need in our first application, and also a copyright function that we haven't called up at all. It's usual to have a whole group of related functions in a file, and it's usual for such files of functions to be used selectively. In other words, for there to be functions declared which are only used in some of the applications into which they're loaded.

Compile the *ir.cpp* file using:

```
g++ -Wno-deprecated -c ir.cpp
```

to give you an object file called *ir.o*.

You can then load all (or rather both in this early case) object files, plus all the system library stuff, to give you an executable file using the **g++** command again with some different options:

```
g++ -o app1 app1.o ir.o
```

The resulting executable file (specified by the **-o** options) can be run from the command line:

```
-bash-3.00$ ./app1
Enter year of first book 2006
Enter year of second book 2003
First book is newer
Age difference is 3
first book: 2006
second book: 2003
-bash-3.00$ ./app1
Enter year of first book 1998
Enter year of second book 2006
First book is older
Age difference is 8
first book: 1998
second book: 2006
-bash-3.00$
```

Exercise

Write a program to ask the user to enter two prices and report the sum of those prices. Like we did, please use a function in a separate file for the prompt-and-read sequence.

15.6 The make system

You'll have noted that even with two files, there's a lot of work involved in administering the compile and load cycles, and in knowing what goes with what. The make system provides you with an easy route to remember all the commands and to automate tasks such as cleanups, and it also helps you efficiently recompile and reload only elements that are necessary.

The **make** command defaults to a file called *makefile* in the current directory; here's the example that we've used for the earlier application.

```
# makefile - module C231

# Compile / Load instructions for examples in this module

# First application demo

COPTS = -Wno-deprecated

# Header file use to avoid spec duplication
# File of functions used to make common code sharable

ir.o:   ir.cpp ir.inc
        g++ ${COPTS} -c ir.cpp

app1.o: app1.cpp ir.inc
        g++ ${COPTS} -c app1.cpp

app1:   app1.o ir.o
        g++ -o app1 app1.o ir.o

# Tidy up directory / remove all intermediate files

clean:
        @rm -rf app1
        @rm -rf *.o

all:    app1
```

Note:

- Each section is headed by a line describing a target, and what it depends on. Targets are usually files, and will only be regenerated if the dependencies have been timestamped later than the target. In this way files that would not have been changed by being regenerated are left alone.
- Following lines(s) after target/dependencies are the instructions of how to generate the target.
- It is vital that the following instruction lines start with a tab character (a number of spaces will not do), and there's a tab between the target and the dependencies too.
- An **@** character in front of a command suppresses any error.
- Targets "clean" and "all" are conventionally used to have clean up the directory suitable for distribution, and to remake all targets. Thus:

```
-bash-3.00$ make clean
-bash-3.00$ make all
g++ -Wno-deprecated -c app1.cpp
g++ -Wno-deprecated -c ir.cpp
```

```
g++ -o app1 app1.o ir.o
-bash-3.00$
```

Saves a lot of typing, doesn't it?

Exercise

Set up a makefile to allow you to recompile and reload your earlier exercise without having to type in fresh **g++** commands each time you rebuild. Provide clean and all targets, and test it.

Note how when you change only your main application source only limited files will be recompiled. Make is really powerful as your application grows.

16

Object Orientation: Individual Objects

Object Orientation is perhaps the best design approach for medium-size to large applications and systems. In this module (the first of three on Object Orientation that are equally applicable to all the OO languages that we teach), we define object orientation, and study the design of simple objects.

16.1 A History

Unstructured Code

The earliest of computer programs were written to perform a specific task and comprised a series of instructions, like a book starting at the beginning and working through to the end – "glorified calculators" if you like. Very rapidly, conditionals and loops were added.

We'll call programming in this way "unstructured" since the code written by the programmer is placed into a single block rather than being compartmentalised.

To this day, you'll probably use something closely akin to unstructured code if you're writing a little utility which doesn't connect to or share code with other programs. For a short, one-off task it remains the best way to do it!

Two features missing from unstructured code, though:
- The ability to re-use common code
- The ability to split code into manageable blocks

Subroutines, functions, procedures

There are some things you'll want to repeat in your programs. Perhaps you'll want to read a series of values from the user. Easy enough? Yes, but in each case you may want to:
- Issue a prompt
- Read in the value
- Check that it's within the correct range and re-prompt if it isn't ... which is going to be rather more than one line of code

Within a loop, if you're reading into an array, this works fine with unstructured code. If you're issuing a series of very different prompts, accepting different values in, and saving into different variables, it's not so easy! Solution?:
- Split the common code out into a separate block
- Give the block a name through which it can be referenced
- Provide a mechanism to tell the block what variables to use
- Call the separate block from the main program

Separate, named blocks of code which are called up from your main program, and remember where to jump back (return) to, are known variously as "functions", "subroutines", or "procedures", depending on what programming language you're using. You'll also come across the word "method" later on; a method is also a subroutine/function/procedure, but it has additional capabilities and features.

Other important features of subroutines[1]:
- They can be nested, i.e. one can call another, and so on
- Can be held in separate files shared between applications

The ability to use subroutines shared between applications is used heavily by every programming language. The standard, supplied utilities such as you'll already be using to display output on the screen are supplied in the form of subroutines, in a separate library, which your program calls as necessary.

Structured Programming

It's been said that all programmers are lazy. That may or not be true, but it's very unusual to find someone who likes to rework a problem having done something very similar before. A good use of subroutines, with careful design, can make for very reusable, maintainable code.

One recommended way to make a good use of subroutines is to use what's referred to as a "structured programming" technique. You break down your main program

1 We'll stick with that word!

into a series of subroutine calls, each of which performs a single task. You then break down each of these tasks into a further series of sub-tasks, each of which is also written as a subroutine. And so on for as many levels as you need.

Even for a complex application, the main program will now read clearly. Choose your subroutine names carefully, and it will even be self-commenting.

Look at this snippet of code:

Java

```
initialise_server(79,"myfinger");
while (true) {
client_ref = await_connection();
if (client_authorise(client_ref)) {
            fork_handler(client_ref);
} else {
            flag_security(client_ref);
            break;
            }
}
System.err.println("Server died due to security breach");
```

Perl

```
initialise_server(79,"myfinger");
while (1) {
            $client_ref = await_connection();
            if (client_authorise($client_ref)) {
                        fork_handler($client_ref);
            } else {
                        flag_security($client_ref);
                        last;
            }
}
print STDERR "Server died due to security breach\n";
```

Can you guess what it does? How it handles errors? Yes, you probably can, even if you don't recognise all the elements of the language.

Could you decide where to look if you wanted to change or debug the authorisation of connections?

Structured programming provides a good way of sharing code, and indeed it's excellent for small- to medium-size projects. Some problems can come to light, though, as you start working on larger and more complex projects. For example:

- Subroutine name conflicts, where two subroutine authors happen to have used the same name
- Any hierarchy of calling has to be imposed by self-discipline alone.
- Data structures from the lower level subroutines can be seen (and often must be allowed for) by the upper level subroutines.

Object Oriented Programming

For larger projects, especially where code is likely to be shared between a number of programs, you'll probably adopt "object oriented programming". The rest of this module will go on to describe "OO Programming" in more detail.

A note on political, or religious issues

You'll know that if you talk about politics, religion, or football teams, you'll find you're talking to people who have very strong views that may differ from your own. You'll also find that many people are fixed in their views and won't even concede that views other than their own can be valid.

Similar discussions rage between programmers. You'll find some programmers who abhor the whole idea of objects and consider them a complete waste of time. You'll find others who suggest that a program that doesn't use objects is always a bad program. And, as in the area or religion or politics, you'll often find that these views are strongly held by people who really aren't enough of an expert in both structured and object programming as a whole to really make an informed choice.

At the risk of falling into this trap, I'll add my view here. My credentials are that I've programmed for many years, writing live projects in a number of languages. Some of these languages were designed for structured programming, others for object oriented programming. Here goes ...

Each approach has its place.

If you're writing short programs or programs with little complex data (even if they're sharing code between a lot of programs), a structured approach might be the best for you. There's an overhead involved in the setup of objects, and in programs, such as the ones we're talking about, we can do without the overhead.

For larger programs or projects, or those with complex data structures that you wish to compartmentalise away from the top-level programmer, you should be using objects. The extra re-usability, ease of maintenance, additional robustness, etc., mean that the extra overhead is worth the cost.

It's really like choosing your mobile telephone tariff. If you make a few calls, you'll be on "occasional caller" which has a low subscription charge, but the per-minute rate is higher. That's like structured programming. On the other hand, if you're making lots of calls you'll be happy to pay the extra subscription involved with "frequent caller" to get the benefits of cheaper calls. That's like object oriented programming.

The parallel goes further. Just as mobile phone companies add a whole lot of extras to encourage you to use their product, so do the authors of programming languages. This leaves you with a very difficult decision to make!

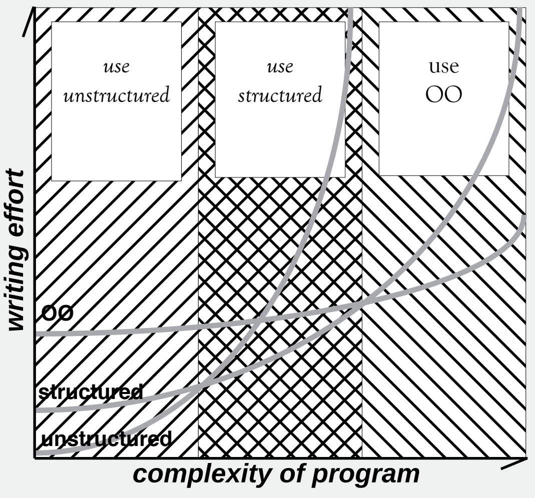


Figure 1 Where OO fits in as "good" and "bad"

16.2 Introduction to OO Programming

Why are we writing programs? To handle data!

We might be writing a whole series of programs to use the same data sets, and we'll want to share a whole load of the subroutines between those programs.

If we're going to use OO techniques, we'll elect one member of our team as the expert on a particular type of data. He'll listen carefully to what all the users want to do, and how they want to do it. He'll think carefully about any additional facilities he can foresee them wanting in the future so that he can allow for that. Only when he's been through the design phase will he actually start to program!

You'll probably notice that I haven't talked of any additional facilities in the language to handle OO yet, and indeed it is possible to follow this philosophy even when using a structured language. OO languages add a range of extra facilities that allow the philosophy to be followed more easily, to be extended, and to impose the encapsulation of data within the sets of subroutines.

Ah – "encapsulation" – the first of many special words that are used in object oriented design and programming! I'll go on and explain those words, using an example as I go through.

Let's take an organisation dealing with the writing and publication of technical manuals/books/courses. That organisation wants to have a whole suite of programs to manage and format the text of each chapter, to keep up-to-date with reviews, to hold copyright information, and much more.

Let's go into some details of object orientation.

16.3 Classes

In object oriented programming, your programming is divided into a series of classes where each class deals with objects (things, if you like) of a particular type.

Typically, each class will be written in its own file so that it can be called into a whole series of different applications as and when needed.

In our example, we'll start with a class called "chapter", which we'll write in a file called:

`chapter.java`	if we're using Java
`chapter.pm`	if we're using Perl

We'll declare the contents of the file to be in the class chapter by writing:

`public class chapter {`	in Java
`package chapter;`	in Perl

What goes into our class files?

Methods

We've introduced the concept of subroutines earlier in this chapter, and within the files that relate to each class, we write a number of subroutines. These subroutines are called "methods".

Some of the methods that we write will be made available to programmers calling our class, but others might be ones that we ourselves call internally. Languages such as Java allow us to set the accessibility of methods from outside the class by using keywords such as

```
public
private
protected
package
```
[1]

In languages such as Perl, the user of a class is trusted to know what he's doing and abide by what he's told he may call.

Static and Nonstatic

It's likely that I'll want to handle a number of objects of the same type within a program. Taking my chapters, for example, I'll want to be able to write a calling program that assembles 15 to 20 chapters into a book.

This means that I'll need to be able to call a method telling it in which particular chapter I'm interested.

[1] Not really a keyword ... it's the default!

The syntax (in my calling program) will be something like:

```
pscript = chap12.getps(297,210);                 (Java)
$psript = $chap12 -> getps(297,210)              (Perl)
```

which requests that a method called **getps** is run on a particular chapter.

Within my class, the actual methods will be defined:

```
public String getps(int height, int width) {     (Java)
sub getps {                                      (Perl)
```

There will be other times in my calling program that I want to find out information about a class as a whole. For example, I might want to enquire how many chapters my program knows about. A method that works on the class as a whole is known as a static method, and is called:

```
numchap = chapter.getcount();                    (Java)
$numchap = chapter::getcount();                  (Perl)
```

Within my class, the method definitions will start:

```
public static int getcount() {                   (Java)
sub getps {                                      (Perl)
```

Instances

Remember that our class can handle multiple chapters at the same time? We refer to that as "multiple instances".

Details of each chapter, each instance, will need their own unique set of variables to hold relevant information. Working out what particular information needs to be held is one of the early jobs for the author of a class.

In the case of our chapters, we may decide to hold the following pieces of information:

– The text of the chapter
– The author
– The year written
– Whether or not it's been published

For each instance, then, we need a number of variables. In Java, we might declare them as:

```
String chaptext;
String author;
int year;
boolean published;
```

Perl is a language in which we don't declare variables; they're created as our program runs.

It's vital to understand that the **int year** declaration doesn't refer just to a single integer. Rather, it refers to a different integer for each instance of a chapter. If we have 12 chapters, then we'll have 12 variables, all called "year".

In Java, the instance variables are declared within the class, but not within any method therein. You can also declare variables that relate to the class as a whole at that point:

```
static int nchaps = 0;
```

declares a static or class variable called `nchaps`, which is initialised to zero when the class is loaded. There's only one **nchaps**, even though there are 12 "year"s in our example.

Constructors

So how do we know which particular instance, which particular chapter, we're working on at any particular time? And how do we set up a chapter object in the first place?

New objects are created by a special type of method known as a "constructor". Although the syntax differs greatly between Perl and Java, when we call a constructor, the following happens:

- Storage – a set of variables are set aside to hold the instance variables
- A "reference" is created – a key by which the object can be referred later – and passed back to the calling program.
- Constructors are called using the new keyword in Java. In Perl it's usual to call the subroutine we use "new".

Here are examples of the calls:

```
chap12 = new chapter("Programming Tools","Graham Ellis",2000);       (Java)
$chap12 = new chapter("Programming Tools","Graham Ellis",2000);      (Perl)
```

You'll notice that the constructor call doesn't have to contain data for all the instance variables. Indeed, it may have no parameters at all. Because the data is going to be encapsulated in the class, the caller need not be aware of the internal structure at all.

The variable returned by the constructor merits further comment. It's the variable we saw a few minutes ago, the one which was used in further calls to methods to identify which particular object we're talking about. It is important to realise that this variable is just a reference; it doesn't contain the actual data. If you've programmed in C before, they're rather like pointers.

Another parallel to these reference variables is file handles. No doubt you're used to referring to files by name. Of course you wouldn't dream of referring to the block and sector numbers on the disk, would you? A file handle or file name is the equivalent to the reference variable; the actual block and sector number, and the rest, are encapsulated within the routines that you're used to calling to handle files.

Destructors

When a new instance of a class is called, you've seen how the constructor sets aside memory for it. It can also do other setup operations. For example, it could grab the data out of a relational database.

What happens when our main program no longer requires the instance? How are we going to write any changes back to the database?

When a variable containing a reference gets overwritten,[1] what happens to the data in our object?

If the reference variable has been copied to another variable, the data is retained. The data has several names, and deleting one of them doesn't release the data. If it's the last reference to an object that's being overwritten, then a special method called the "destructor" is called.

The destructor is a method with a particular name – **`DESTROY`** in Perl, **`destroy`** in Java. Its purpose is to clean up the object. Taking our example where the data was initially loaded from a database, and might then have been modified, the destructor will complete the cycle by writing the changes back to the database before it releases the memory occupied by the object's variables.

You don't call destructors explicitly; they're called automatically for you as you release objects. If you have any objects which have not been released as you leave your program, the destructor is called on each remaining object in turn for you. If you don't believe me, put a `print` statement in your destructor and see it in action.

You don't have to provide a destructor; if you don't, no action except the releasing of memory is done as you finish with an object.

Overloading

It might be that you (as the author of a class of objects) want to provide your users with various different ways of calling the same basic functionality. A constructor that

[1] In our earlier example, if the chap12 variable is assigned a new value.

either takes its data from parameters, or takes a file name as its parameter and then reads that file, or just constructs an empty object – a skeleton.

Yes, you can do this. Unlike languages where each subroutine must have a unique name with a unique sequence of calling parameters, both Perl and Java allow you to "overload". Here are examples of how you might call the constructor for a chapter:

```
chap12 = new chapter("Programming Tools","Graham Ellis",2000);       (Java)
chap13 = new chapter("pt.txt");
chap14 = new chapter();

$chap12 = new chapter("Programming Tools","Graham Ellis",2000);      (Perl)
$chap13 = new chapter("pt.txt");
$chap14 = new chapter();
```

This works very differently within the classes – in Java, a separate method is provided for each possible configuration of parameters (number and type) that the class programmer wishes to allow. In Perl, the new method simply examines how many parameters it was called with, and what they are.

16.4 Accessing members of a class

Having seen and learnt what a class is, it's worth our while stopping to consider what the users of our class need to know and do to use it.

Loading

Users need to know where to load a class from (or if it's going to be available automatically).

Use

Users need to know what methods are available to them within the class:

- method names
- parameters to those methods
- what they do
- what they return

You'll notice that the user does not need to know how the methods actually work; that's back to encapsulation.

Direct access to variables

The object oriented paradigm says that the class programmer should provide methods to perform every action that his user wants – an excellent rule, but sometimes a little frustrating to the class programmer who has a very large number of different variables to set and get for each object. It is possible in both Java and Perl for the paradigm to be broken, and for variables to be accessed directly. In Java, the class programmer needs to set the permission; in Perl, we trust the user to know what he's doing!

Testing

Because a class will be shared by a number of programs, it's worthwhile testing it thoroughly.

A simple test program will probably suffice at first, but as your classes get more complex, these will grow into more complex testing suites. Such a test program or suite is often referred to as a "test harness", since it harnesses the power of the class and exercises each element in turn.

Although test harnesses are often written in classes of their own, it's quite practical to include a test harness within the source code of your class in such a way that it's only going to be executed when the file as a whole is executed, and not when it's loaded as a called class.

Encouraging class use

Both Java and Perl provide you with the ability to document your classes via information contained within the class, and such information is of two types:

- Comments – ideal for the maintenance programmer, not seen by the user
- Documentation – seen as a special type of comment by the maintenance programmer, but also accessible by, and visible to, the user.

"Documentation comments" start `/**` and end `*/` in Java. In Perl, there's a whole special format used, called "POD", using an `=` character in the first column.

But then Java goes a stage further.

If you're an author of classes, you'll be advised to adopt certain conventions in the naming of methods, basically starting the names with "get" if you're retrieving values and "set" if you're setting values. As well as helping the user understand what a method does, this allows special tools provided with Java[1] to look inside the classes and provide templates for their use, test harnesses so that the calling programmer can experiment with them without having to write special programs, etc.

A Java class which adheres to the naming rules is known as a "bean". Further classes can be provided by the class programmer, as well as *chapter.java*, you would provide:

chapterBeanInfo.java	further information for the special program
chapterCustomEditor.java	Special editors to validate parameters, etc

1 Or can be purchased separately

Exercise

17

Defining and Using Classes in C++

In this module, you'll learn how to define and use simple classes in C++.

17.1 Your first C++ class definition and use

Although you'll separate out your main program and your classes into different files for any substantial live application, let's have a look at a minimal class definition and use all in the one file.

This is file *hmain.cpp*:

```
#include <iostream.h>

// First definition and use of C++ class

/*  This is file is a minimal class definition
and use in C++ - showing the most basic of
structure.
*/

class Hotel {
        public:
                int roomcount;
                float occrate;
};

int main () {
        Hotel manor;
        Hotel beechfield;
        manor.roomcount = 6;
        beechfield.roomcount = 18;
        manor.occrate = 0.85;
        beechfield.occrate = 0.35;

        int totrooms = manor.roomcount + beechfield.roomcount;

        cout << "Total rooms listed: " << totrooms << "\n" ;

        return 0;
        }
```

Class Declaration

The class is declare as follows:

```
class Hotel {
        public:
                int roomcount;
                float occrate;
};
```

The script says that each object of type Hotel is going to have two member variables – an integer one called "roomcount" and a float one called "occrate" – associated with it. Since they're declared to be public, they'll be visible and useable from anywhere inside or outside the class.

Unusually, we've not declared any accessor methods at all so our code will make direct reference to these variables.

Program code to use the class

The main program code starts by declaring that there are to be two objects of type Hotel, and it then assigns values directly in to their member variables. You may be familiar with the "." operator from structs and unions in C; at this level and in this example, a class is just a glorified structure:

```
        Hotel manor;
        Hotel beechfield;
        manor.roomcount = 6;
        beechfield.roomcount = 18;
        manor.occrate = 0.85;
        beechfield.occrate = 0.35;
```

The code then proceeds to make use of the data held within this object and report on its results:

```
        int totrooms = manor.roomcount + beechfield.roomcount;
        cout << "Total rooms listed: " << totrooms << "\n" ;
```

When the code is compiled and run, you'll see:

```
-bash-3.00$ make hmain
g++ -Wno-deprecated hmain.cpp -o hmain
-bash-3.00$ ./hmain
Total rooms listed: 24
-bash-3.00$
```

17.2 Declaring your class in a separate file and accessing it via methods

That first example has got us into the whole world of OO programming, but we've not yet encapsulated any logic into the class, nor have we provided any accessor methods nor separated the class out into a separate file that can be maintained by another team member, and tested separately. Let's take those steps now.

To start off, here's a file that declares what the class looks like, and defines what it does:

```
/*
This will compile but not load on its own ...
there's no main program provided. You can
find a sample main program in afuncs.cpp

*/

class Hotel {
        public:
                void SetRoomcount(int num);
                int GetRoomcount();
        private:
                int roomcount;

};

void Hotel::SetRoomcount(int nrooms) {
        roomcount = nrooms;
        }

int Hotel::GetRoomcount() {
        return roomcount ;
        }
```

That's file *hcd.cpp*. The class declaration is as follows:

```
class Hotel {
        public:
                void SetRoomcount(int num);
                int GetRoomcount();
        private:
                int roomcount;

};
```

That tells us that there are two methods you can call on each member of the class – a method called SetRoomcount that takes a single integer parameter, and a method called GetRoomcount which takes no parameters but returns an integer.

The declaration also calls for a variable called roomcount within each member of the class (different hotels have different numbers of rooms), but declares it to be private to indicate that it cannot be accessed directly from outside. Clearly, it's going to be accessed through SetRoomcount and GetRoomcount, which we know are methods (named pieces of code that run on an object) since they're declared with brackets **()** after the member name.

Here's the code that actually declares just what those methods do:

```
void Hotel::SetRoomcount(int nrooms) {
        roomcount = nrooms;
        }

int Hotel::GetRoomcount() {
        return roomcount ;
        }
```

The code is simply saving and returning the value from the roomcount variable that's already been declared. Note you can have:

- Parameter variables - e.g. **nrooms**
- Local variables (none here)
- Instance variables - e.g. **roomcount**
- Static class variables (none here)

The use and scope of each of these is paramount to understanding C++.

In file *afuncs.cpp*, we have:

```
#include <iostream.h>

/*

This will COMPILE correctly since we're correctly defining a Hotel
and making references to that definition, but it will NOT load on
its own since we have not provided the code for methods
GetRoomcount and SetRoomcount.

You can find the code that implements the class
methods in hcd.cpp

*/

class Hotel {
        public:
                void SetRoomcount(int num);
                int GetRoomcount();
        private:
                int roomcount;

};

int main () {
        Hotel Manor;
        Hotel Beechfield;
        Manor.SetRoomcount(6);
        Beechfield.SetRoomcount(18);

        int totrooms = Manor.GetRoomcount() +
                           Beechfield.GetRoomcount();
        cout << "Total rooms listed: " << totrooms << "\n" ;

        return 0;
        }
```

Yes, we have repeated the definition of the class member in that file (we'll put it into an include file soon to avoid being so wasteful – all programmers are lazy!), and we've also provided a main method that calls up the methods.

It compiles and runs as follows:

```
-bash-3.00$ make hcd
g++ -Wno-deprecated -c afuncs.cpp
g++ -Wno-deprecated -c hcd.cpp
g++ -o hcd hcd.o afuncs.o
-bash-3.00$ ./hcd
Total rooms listed: 24
-bash-3.00$
```

17.3 Adding an include file for headers

Let's extend that application a little further to bring in an include file for the headers. That way, you'll be saved the effort of duplicating the code into both files, and you'll not run the risk of them getting out of step.

Header file *hotel.inc*

```
/*

This is a header file and defines the class Hotel and its members -
a "template" that's available for use in both the file that
defines the code for each class member, and also for all
applications that use members

*/

class Hotel {
        public:
                void SetRoomcount(int num);
                int GetRoomcount();
        private:
                int roomcount;

};
```

Hotel definition file *hotel.cpp*

```
/*

The code that defines a hotel

*/

#include "hotel.inc"

void Hotel::SetRoomcount(int nrooms) {
        roomcount = nrooms;
        }

int Hotel::GetRoomcount() {
        return roomcount ;
        }
```

Application in *melksham.cpp*

```
#include <iostream.h>
#include "hotel.inc"

int main () {
        Hotel Manor;
        Hotel Beechfield;
        Manor.SetRoomcount(6);
        Beechfield.SetRoomcount(18);

        int totrooms = Manor.GetRoomcount() + Beechfield.GetRoomcount();

        cout << "Total rooms listed: " << totrooms << "\n" ;

        return 0;
        }
```

Appropriate lines from *makefile*:

```
# class definition and use via accessor methods
# Header file use to avoid spec duplication

hotel.o:        hotel.cpp hotel.inc
        g++ ${COPTS} -c hotel.cpp

melksham.o:     melksham.cpp hotel.inc
        g++ ${COPTS} -c melksham.cpp

melksham:       melksham.o hotel.o
        g++ -o melksham melksham.o hotel.o
```

That's been three examples to build up to there; the smallest really useful and practical class. Time for an exercise?

Exercise

Create a class of object of type Chronometer, with a weight and a colour variable (also known as a property or attribute) for each chronometer. Provide accessor methods to allow the weight to be set and read back, but set up the colour through a variable in each member which can be accessed from outside the class.

Write a small application program in which you define two chronometers – one is a brown replica station clock of the type you may see in our training room, and weights 650 g, the other is a 70 g silver pocket watch. Having defined your two time pieces, please write code to call back each of their attributes so that you display details as follows:

```
$ ./clocklist
A 650g clock coloured brown
A 70g clock coloured silver
$
```

17.4 Constructors, destructors, consts and inline methods

In our previous example, when our application declared an object of type hotel, an empty hotel was created and we had to go through a series of calls to populate it with data. In practice, we'll usually want to declare and populate at the same time, and we can do this by declaring our own constructor, a method with the same name as the class name which is automatically called to provide extra functionality as the object is created.

As well as constructors, we'll have destructor methods. A destructor method is automatically called when an object is no longer needed. The default destructor simply releases memory, but you can also take other actions such as flushing database information out to disk if you need to.

Here's the header fill for a class of object of type book:

```
class Book {
        public:
                Book(int year, int pages);
                ~Book();
                void setPages(int num);
                int getPages() const;
                int getYear() const;
        private:
                int itsyear;
                int itspages;
};
```

Note the const keyword, which tells the compiler that this method only uses the member variables and does not change them. It's done to help make the code more efficient.

And the actual code for the class:

```
/*

The code that defines a book

*/

#include <iostream.h>
#include "book.inc"

// Constructor - code to run when a book is created

Book::Book(int year, int pages) {
        itsyear = year;
        itspages = pages;
        cout << "Creating book\n";
        }

// Destructor - code to run when a book is eliminated

Book::~Book() {
        // Null method - just f.y.i.
        cout << "Destroying  book\n";
        }

// Accessor methods
// Pages property

void Book::setPages(int nrooms) {
        itspages = nrooms;
        }

inline int Book::getPages() const {
        return itspages ;
        }

// Year property

int Book::getYear()  const{
        return itsyear ;
        }
```

The final source code file is an application that uses the book class:

```
#include <iostream.h>
#include "book.inc"

/*

# Main application calling constructors and destructors

*/


int main () {

        Book Coatimundi(1996,207);
        Book Chipmunk(2003,791);

        int b1y = Coatimundi.getYear();
        int b2y = Chipmunk.getYear();
        int diff = 0;

        if (b1y < b2y) {
                cout << "First book is older\n" ;
                diff = b2y - b1y;
        } else if (b1y == b2y) {
                cout << "Books are of same age\n" ;
        } else {
                cout << "First book is newer\n" ;
                diff = b1y - b2y;
        }

        cout << "Age difference is " << diff << "\n" ;

        return 0;
        }
```

And the complete makefile for all the examples in this module:

```
# makefile - module C232
# First class definition and use via member variables
```

```
COPTS = -Wno-deprecated

hmain:  hmain.cpp
        g++ ${COPTS} hmain.cpp -o hmain

# class definition and use via accessor methods

afuncs.o:       afuncs.cpp
        g++ ${COPTS} -c afuncs.cpp

hcd.o:  hcd.cpp
        g++ ${COPTS} -c hcd.cpp

hcd:    afuncs.o hcd.o
        g++ -o hcd hcd.o afuncs.o

# class definition and use via accessor methods
# Header file use to avoid spec duplication

hotel.o:        hotel.cpp hotel.inc
        g++ ${COPTS} -c hotel.cpp

melksham.o:     melksham.cpp hotel.inc
        g++ ${COPTS} -c melksham.cpp

melksham:       melksham.o hotel.o
        g++ -o melksham melksham.o hotel.o

# class definition and use via accessor methods
# Header file use to avoid spec duplication

book.o: book.cpp book.inc
        g++ ${COPTS} -c book.cpp

mylib.o:        mylib.cpp book.inc
        g++ ${COPTS} -c mylib.cpp

mylib:  mylib.o book.o
        g++ -o mylib mylib.o book.o

# Tidy up directory / remove all intermediate files

clean:
        @rm -rf hcd hmain melksham mylib
        @rm -rf *.o

all:    hmain hcd melksham mylib
```

Compiling and running:

```
-bash-3.00$ make clean
-bash-3.00$ make all
g++ -Wno-deprecated hmain.cpp -o hmain
g++ -Wno-deprecated -c afuncs.cpp
g++ -Wno-deprecated -c hcd.cpp
g++ -o hcd hcd.o afuncs.o
g++ -Wno-deprecated -c melksham.cpp
g++ -Wno-deprecated -c hotel.cpp
```

```
g++ -o melksham melksham.o hotel.o
g++ -Wno-deprecated -c mylib.cpp
g++ -Wno-deprecated -c book.cpp
g++ -o mylib mylib.o book.o
-bash-3.00$ ./mylib
Creating book
Creating book
First book is older
Age difference is 7
Destroying  book
Destroying  book
-bash-3.00$
```

Exercise

Modify your chronometer class by adding in a constructor that allows you to create a chronoment and also to pass the initial settings in to it all in a single declaration.

Provide a separate method that returns the weight of the chronometer in ounces (divide the grams by 28). This is the first time you've encapsulated calculation and logic within the class rather than simply using it as a property container.

18

Object Orientation: Composite Objects

This is the second of three modules on Object Orientation which are equally applicable whether you're programming in Java or Perl, PHP or Python. Topics covered include Polymorphism, Inheritance and Overloading.

18.1 Revision

We're writing programs to handle data, and so "data is King".

An object is a group of data elements, and a number of methods that are used to access that data.

There will often be multiple objects of one type (multiple chapters, multiple pens, etc), and each object is known as an instance. Objects of a particular type are referred to as a whole as a class.

In order to create a member of a class, you call a special method called a "constructor" which sets up a set of all the variables you need to hold the data and returns a reference to the instance, also known as an instance variable.

Subsequent calls to methods in the class are told which particular object you want them to work on by running them on the particular instance. You can also call static methods which refer to the class as a whole rather than to individual members.

When an object is no longer required, a destructor is called to tidy up and release memory. Usually the destructor is called implicitly by your program when you reassign or delete your instance variable.

In many Object Oriented languages there can be several methods of the same name in the same class. The system decides which particular method the user called by looking at the number and type of parameters. This is known as "overloading".

18.2 Inheritance

We've spent some time working with our example of a chapter, and by now you'll understand a little of how I can do things that relate to chapters by using the class. But let's go on and extend the example.

As it turns out, I don't just write books, I write training courses as well.

When I come to write my first programs that deal with training course modules, I look with some envy at the "chapter" class already written. I need most of what's there, but I also need a few other things. What can I do? Taking a copy of the source code and making changes or additions is not a good idea since it will lead to a maintenance problem in the future. Instead, I use inheritance.

Base classes and subclasses

My original class –"chapter" in the example that I'm using – is going to be used as the basis of my new class "module".

We describe "chapter" as the base class and "module" as the subclass.

We leave the code of chapter untouched, but within the code of "module" we declare that a module is a chapter. We then have to specify only the extras that we need in our "module" class.

Declaring that "module" is a subclass of "chapter":

Java

```
public class module extends chapter {
```

Perl

```
use chapter;
@ISA = qw(chapter);
```

There's a little more to it than that, but what we've achieved is the ability to write common code only once, and have it used by a whole series of related classes. As well as extending our base class "chapter" into "module", we could extend it into "bookchapter", "lawsection", "diaryentry" and a host of other mutually exclusive subclasses.

What's inherited?

The author of any class can't prevent others subclassing his objects, but in Java he can control what is (and what isn't) visible to the subclass. In Perl, of course, the user is trusted with everything if he wants it.

In any case, methods and direct access variables can be inherited by the subclass, with the exception of the constructor method. You do have to ask for a "new module" rather than a "new chapter", so the author of the subclass must provide a constructor. Usually, his first action will be to call the base class constructor: "To make a module, make a chapter and then add ..."

Each instance of an extended class will contain all the variables defined for an instance of the base class, and all the extra variables defined for an instance of the extended class.

Overriding

It could be that on some occasions a method provided in the base class is inappropriate to the extended (or derived, or sub) class. In such a situation, the author of the extended class simply provides a method of the same name as the method that he wants to override, and his new method will be called in preference to the one in the base class on objects of his extended type.

As an example, let's say that the `getpublished` method returns the contents of a boolean variable in the base class book. In our "module" extended class, though, `getpublished` must check through a database of courses presented and see if the particular module was actually used on any of them. Easy; we just provide a `getpublished` method in both our base class and in the subclass.

Abstract classes

Our original chapter class probably contains a great number of useful methods, but in practice we'll probably find that we don't want to create any plain "chapters"; after all, such documents are written with a particular purpose in mind.

Methods such as `getps`, `setauthor` and `getauthor` can be inherited throughout the class.

Methods such as `getpublished` can be overridden in awkward subclasses.

But there may be a number of other methods which, whilst they apply to all chapters, are going to differ in every subclass. An example might be a `getvalue` method; value calculations for diary entries, books, and training modules will vary widely.

If we have a requirement for methods such as `getvalue`, we'll describe the class as being abstract[1]and we'll then have to provide a `getvalue` method in every subclass.

Polymorphism

Let's continue our example further.

I'm now writing an application to handle book chapters, training course modules, diary entries, and more. I have a class for each, extended from "chapter" and I'm wanting to call a number of routines, some of which are in the base class, others of which have been overridden in some of the subclasses, and others which might be abstract. Can I do so? Yes, absolutely.

Let's say that `chaptab` is an array of chapters, and I'm looking to find out the total value of all my writings.

Java

```
for (int k=0;k<chaptab.length;k++) {
valtot += chaptab[k].getvalue();
}
```

[1] Java must be told!

Perl
```
for ($k=0;$k<@chaptab;$k++) {
$valtot += $chaptab[$k] -> getvalue();
}
```

Which **getvalue** is called? Possibly a different one each time around the loop. Whichever OO language that you're using, it looks at the reference held in each element of `chaptab` in turn, and decides as it runs which method to call.

Having one name that causes different things to happen depending on what object you call it on is known as "polymorphism" – many things depending on how it's called. You don't explicitly program for polymorphism, it's just there, and if you have methods in different subclasses that all call the same method, you can use it.

Inheritance structure

So far, we've looked at an example with a base class and a number of subclasses, but the concept extends.

Although our base class was described as such, it's only a base class as far as the extended classes are concerned. It could be that a "chapter" is a subset of "asset", where an "asset" object has its own variables and methods, and so on.

Eventually, all classes in Java inherit from a base class called **Object** and in Perl all classes inherit from a base class called **UNIVERSAL**. This means that whenever you write a class of objects, there turn out to be a number of methods available already. They'll let you do things like copy (clone) an object, see if two references point to the same object, etc.

Calls that the user makes to the methods in any class might actually result in methods in the class being called, or perhaps methods in the base class being called. The model goes on; a program will only say it can't find a method if it can't fine one in any subclass.

Multiple Inheritance

C++ and Perl (but not Java[1]) can subclass from several classes at the same time. In C++, most experienced programmers recommend that you don't use the facility, but does turn out to be quite useful in Perl. We could say that a "module" is both a "chapter" (to give us methods such as `getps`) and a "coursecomponent" (to give us methods such as `getcourse`).

An alternative in Java is an "interface" which does not allow multiple inheritance, but it does allow the enforcement on the programmer who's implementing the interface to provide certain methods, rather like an abstract class.

If you defined:
```
public class module extends chapter implement coursecomponent{
```
then you could hold modules in arrays of **chapter**s, and in arrays of **coursecomponent**s.

18.3 Class structure

With all these classes and subclasses that we've been talking about, it's likely that you'll be imagining directories filled with many, many files. Yes, that's easily possible, and so a directory structure can be imposed onto classes.

Hierarchy

In Java, we can arrange our classes into groups that logically fit together, and call the group a package. We then keep the package in a separate directory and can call classes therein up explicitly, or by using an "import" statement to short-cut that need.

1 design decision

In Perl, the work package refers to a class, a package is a class. A similar hierarchy is, however, available.

It's worth stressing at this point that the groups that we choose to save into a single directory will include a number of classes and their subclasses, but it will also include a series of classes that are used, perhaps, as part of the other objects. Although we referred earlier on to having `Strings` and `ints` within our chapters, it could well be that we had a "title" object as a member of each instance of the class, and the title class would, most likely, be held in the same directory.

Visibility

With many object oriented languages, there are keywords that the programmer uses to control how near or how far his objects and their individual methods and features can be seen – the public, protected, package, private feature of Java.

There's also the issue of where classes may be held on the computer for them to be callable when programs are run.

Java uses environment variables called **`CLASSDIR`** or **`CLASSPATH`**[1] as its list of where programs should look for classes. Perl uses a list called **`@INC`**.

By default, most languages and installations will look for classes

- In directories reserved for use by the language itself, then
- In site-wide and system-wide directories, then
- In the user's own directory.

18.4 Designing Objects

You've learnt about the buzz words of object orientation, and seen how they fit in with one particular example. You'll probably be a little apprehensive about how they'll fit together when you jump in with both feet.

Some comments.

Designing your objects correctly is vital. You should always plan before you start coding, and there are a number of methodologies that are there to help you. If you get the design wrong, then OO will be a burden rather than an asset. Be warned!

Although we've presented Structured and OO programming as being poles apart, in practice that's not strictly so.

If you're a top-level programmer, i.e. if you're bolting together objects written by others, your style of coding will differ between Structured and OO, but you won't have to be heavily involved in many of the intricacies of inheritance, for example.

You can even use someone else's class of objects in your structured program.[2] Very practical, even if it leaves any purists who see your code up in arms!

Remember, for the right applications, OO is great. But please don't follow it as a dogma.

[1] depending on which release you're using

[2] Perl is especially good at this!

Exercise

19

OO in C++ — Beyond the Basics

If you want to define a new type of object that's similar to one you've already defined, you use "inheritance" ... and you can build up a complete hierarchy tree of objects if need be. If you use an array of different objects, you can arrange for each of them to behave slightly differently, using "polymorphism".

19.1 Inheritance

In an earlier module, we defined a class of Hotel, and very nice it was too. Let's assume that we've done a lot more work on that class and it's now a quite valuable piece of code, when some chap comes along with a new applications and say "great class Hotel, can I have a copy because I want to define a Melkshamhotel"?

Being nice folks, sure, we give him a copy. But then when we have to modify the whole Hotel class to take account of some new legal requirements that come in and let him know, he's f-ing and b-ing because he's got to modify his code too. Wouldn't it have been much better if he's defined a file that only called up the differences, and was stated as being based on the hotel class? That's inheritance.

Let's see how it works in practice. Here's the original class hotel, known as a base class since other classes are to be based on it:

```
/*

The code that defines a hotel

*/

#include "hotel.inc"

void Hotel::setRoomcount(int nrooms) {
        roomcount = nrooms;
        }

int Hotel::getRoomcount() {
        return roomcount ;
        }
```

If you ask "what's the difference, what's changed?", the answer is "nothing". My hotel class as it previously was implemented can form the base for other classes. Here's my **melkshamhotel** class:

```
/*

The code that defines derived class MelkshamHotel's methods

*/

#include "melkshamhotel.inc"

void MelkshamHotel::setDistance(float kms) {
        distance = kms;
        }

float MelkshamHotel::getDistance() {
        return distance ;
        }

void MelkshamHotel::setAll(float kms, int nrooms, char *about) {
        distance = kms;
        roomcount = nrooms;
        describe = about;

        }
```

```
bool MelkshamHotel::isWalkable() {
        if (distance <= 1.0) return true;
        return false;
        }
```

Again, just a file of methods with no real clue that it's a derived class; the extra relationship is in fact defined in the header file *melkshamhotel.inc*:

```
#include "hotel.inc"

/*

Definitions of the members that will be added
to a Hotel object to make it into a Melkshamhotel
object

*/
class MelkshamHotel : public Hotel {

        public:
                void setDistance(float num);
                float getDistance();
                void setAll(float kms, int nrooms, char *about);
                bool isWalkable();
        private:
                float distance;

};
```

The crucial declaration here is the **`:public Hotel`** on the end of the class declaration that tells the compiler to cause Melksham hotel to inherit all public members from Hotel, which means it will have the members from that class available to it in addition to the methods just declared. Here's what's in the *hotel.inc* file now:

```
/*
This is a header file and defines the class Hotel and its members -
a "template" that's available for use in both the file that
defines the code for each class member, and also for all applications
that use members
*/

#ifndef HOTEL
#define HOTEL 1

class Hotel {
        public:
                virtual void setAll(float a, int b, char *c) {};
                virtual bool isWalkable() {} ;
                void setRoomcount(int num);
                int getRoomcount();
                char *describe ;
        protected:
                int roomcount;
};

#endif
```

The important thing to note here is the preprocessor directives that ensure that this file's declarations are only included once in any calling code, even if there are several calls to include the file through different paths on the hierarchy.

Finally, let's have an application to create and use some melkshamhotel objects:

```
#include <iostream.h>
#include "melkshamhotel.inc"

/*

# Class using classes and derived classes with inheritance

*/


int main () {
        MelkshamHotel Manor;
        MelkshamHotel TheSpa;
        MelkshamHotel Beechfield;

        Manor.setRoomcount(6);
        Manor.setDistance(0.0);

        Beechfield.setRoomcount(18);
        Beechfield.setDistance(2.5);

        TheSpa.setRoomcount(3);
        TheSpa.setDistance(0.9);

        int totrooms = Manor.getRoomcount() +
                TheSpa.getRoomcount() +
                Beechfield.getRoomcount();

        cout << "Total rooms listed: " << totrooms << "\n" ;

        if (Manor.isWalkable())
                        cout << "Walkable from Well House Manor\n";
        if (Beechfield.isWalkable())
                        cout << "Walkable from Beechfield\n";
        if (TheSpa.isWalkable())
                        cout << "Walkable from The Spa\n";

        return 0;
        }
```

You'll notice that although there's no reference at all in this code to the fact that a MelkshamHotel is a special type of hotel, I can call up the methods from the Hotel class as if they were in the Melkshamhotel class – inheritance into a derived class in action!

Running the code:

```
-bash-3.00$ make accomchooser
g++ -Wno-deprecated -c hotel.cpp
g++ -Wno-deprecated -c accomchooser.cpp
g++ -Wno-deprecated -c melkshamhotel.cpp
g++ -o accomchooser hotel.o accomchooser.o melkshamhotel.o
-bash-3.00$ ./accomchooser
```

```
Total rooms listed: 27
Walkable from Well House Manor
Walkable from The Spa
-bash-3.00$
```

Exercise

Take your Chronometer class, and create a new class called Analog, which is the same as a Chronometer but has an extra parameter which represents the number of minutes gained or lost per week. Turn both of your existing Chronometers into Analog devices, and set them up so that one gains 5 seconds per week, and the other loses 1.5 seconds per week. Provide a method that returns the gain or loss made by an analog timepiece over a year .... for example:

```
Silver --- 70g --- 80.5 seconds slow
Brown --- 680g --- 262 seconds fast
```

19.2 Storing Strings in an object, and other enhancements

A minor enhancement. Let's add some text strings into our objects; a char * member in the *hotel.inc* file that we had earlier can be used.

```
#include <iostream.h>
#include "melkshamhotel.inc"

/*
# Class storing strings within an object
*/

int main () {
        MelkshamHotel Manor;
        MelkshamHotel TheSpa;
        MelkshamHotel Beechfield;

        Manor.setRoomcount(6);
        Manor.setDistance(0.0);
        Manor.describe = "Well House Manor";

        Beechfield.setRoomcount(18);
        Beechfield.setDistance(2.5);
        Beechfield.describe = "Beechfield House Hotel";

        TheSpa.setRoomcount(3);
        TheSpa.setDistance(0.9);
        TheSpa.describe = "The Spa B&B";

        int totrooms = Manor.getRoomcount() +
                TheSpa.getRoomcount() +
                Beechfield.getRoomcount();

        cout << "Total rooms listed: " << totrooms << "\n" ;

        if (Manor.isWalkable()) cout << "Walkable from "
                        << Manor.describe << "\n";
        if (Beechfield.isWalkable()) cout << "Walkable from "
                        << Beechfield.describe << "\n";
        if (TheSpa.isWalkable()) cout << "Walkable from "
                        << TheSpa.describe << "\n";

        return 0;
        }
```

That's much cleaner in its final display, and we're now using the class as a container for pointers to the strings. We need to be very careful to be sure where the memory is actually allocated; setting up a char array within the class would be liable to leave us short on space if somewhere happened to have a very long name!

```
-bash-3.00$ make accom2
g++ -Wno-deprecated -c accom2.cpp
g++ -o accom2 accom2.o melkshamhotel.o hotel.o
-bash-3.00$ ./accom2
Total rooms listed: 27
Walkable from Well House Manor
Walkable from The Spa B&B
-bash-3.00$
```

Note the limited compile; *hotel.cpp* and *melkshamhotel.cpp* haven't changed since the previous example, and the make system knows this and saves itself the effort and time of recompiling.

An Array of objects

Have you noticed that the setup and access code is getting long-winded and will grow further if we add more hotels? Better for us to use an array of objects; as we're hardcoding the data we'll still have to set them up by hand, but at least we can access them in a loop ;-)

```
#include <iostream.h>
#include "melkshamhotel.inc"

/*

# An array of objects

*/


int main () {

        int nhots = 3;
        int totrooms = 0;

        MelkshamHotel recommend[nhots];
        for (int k=0; k<nhots; k++) {
                recommend[k] = MelkshamHotel();
        }

        recommend[0].setAll(0.0,6,"Well House Manor");
        recommend[1].setAll(0.8,3,"The Spa B&B");
        recommend[2].setAll(2.5,18,"Beechfield House Hotel");

        for (int k=0; k<nhots; k++) {
                MelkshamHotel current = recommend[k];
                totrooms += current.getRoomcount();
                if (current.isWalkable())  {
                        cout << "Our training center is walkable from "
                        << current.describe << "\n";
                } else {
                        cout << "If you stay at "
                        << current.describe
                        << " you'll need transport\n";
                }
        }

        cout << "Total rooms listed: " << totrooms+0  << "\n" ;

        return 0;
        }
```

We've now got the main body of our code in a loop, so the number of lines there will be the same for 3 or 30 objects! Note too the use of a "setAll" method to make multiples settings all at the same time, and if you refer back to the original *hotel.inc* file at the top of this module, you'll notice that we're now using a protected member to hold the room count.

- private members are available within a class only
- protected members are available within a class and its derived classes
- public members are available within any class

Exercise

Put all your time pieces into an array and add a string that tells you who the owner is to each of them. Enhance your application to use these new methods and to loop through all the analog timepieces you have, reporting a full description of each.

19.3 Polymorphism and overloading in C++

The case for polymorphism

Let's move on with inheritance and derived classes. From our hotel we derived a class called "melkshamhotel" with a method that works out whether or not a hotel is within walking distance of our place. But there are other hotels as well – hotels in the region where all the same named methods could be called as on a Melkshamhotel, but the answer will always be that it's not walkable.

In other words, we could have a method of the same name as that we used for a melkshamhotel, that did the "moral equivalent". It's just that the internal code would be different.

Wouldn't it be nice if the we could take this a stage further and have an array of hotels made up from a mixture of different derived classes, a heterogeneous array if you like, and have the program decide at run time which particular flavour of "isWalkable" is was to run depending on the object type?

Code can be written in that way, and the effect is known as "polymorphism". You don't declare anything to be polymorphic; you just set things up right and it happens. I'll show you how on different types of hotels, then ask you to do the same thing on your analogue clocks which can gain and lose time, and on a class of radioclocks with do not lose or gain time. And you'll soon see that the same approach that I've applied to hotels and watches can equally apply to your own application, and to more abstract things like a data stream that may be read from a keyboard, a file, or a network connection. It's called in just the same way, but internal elements vary considerably but silently.

Implementing polymorphism and overloading in C++

The first question, what type of variable do you declare for an object that can be both a melkshamhotel and a regionhotel? You declare the variable as holding a member of its base class, i.e. a simple hotel.

That will make all the methods of that base class available when called on any type of hotel, and if you've redefined (i.e. overloaded) any of the methods in the derived classes, then those alternative methods will in fact be invoked in priority.

What if some methods simply can't be defined on a basic hotel? That applies to methods like the isWalkable example I've been quoting. Until you know what type of hotel we're talking about, we can't even start to make the decision. **So we define virtual members in the base class**. By that we mean we declare that a function will exist to operate on every hotel, to keep the compiler and loader happy, but we leave the definition of the true code to the derived class.

Here are the extra declarations (highlighted) in my hotel class:

```
class Hotel {
        public:
                virtual void setAll(float a, int b, char *c) {};
                virtual bool isWalkable() {} ;
                void setRoomcount(int num);
                int getRoomcount();
                char *describe ;
        protected:
                int roomcount;
};
```

We'll then add the extra methods into the melkshamhotel class, and we'll also define our new regionhotel class and make sure that it too has methods to meet the profile specified in the virtual declarations:

```
#include "hotel.inc"

/*

Definitions of the members that will be added
to a Hotel object to make it into a Regionhotel
object

*/

class RegionHotel : public Hotel {

        public:
                void setDistance(float num);
                float getDistance();
                void setAll(float kms, int nrooms, char *about);
                bool isWalkable();
        private:
                float distance;

};
```

And here's the code for *regionhotel.cpp*

```
/*
Derived class member code
The code that defines derived class Regionhotel's methods
*/

#include "regionhotel.inc"

void RegionHotel::setDistance(float kms) {
        distance = kms;
        }

float RegionHotel::getDistance() {
        return distance ;
        }

void RegionHotel::setAll(float kms, int nrooms, char *about) {
        distance = kms;
        roomcount = nrooms;
        describe = about;

        }

bool RegionHotel::isWalkable() {
        return false;
        }
```

Some of those methods should arguably be in the base class, but certainly the isWalkable method is truly polymorphic – different code being called up at different times.

It just remains to write an application to use one of these heterogeneous arrays that we've been talking about.

An application uses polymorphism

I'll show you the source code first, and how the program runs, then I'll point out some of the new features.

```
#include <iostream.h>
#include "melkshamhotel.inc"
#include "regionhotel.inc"

/*
 An array of objects of slightly different types
*/


int main () {

        int nhots = 5;
        int totrooms = 0;

        Hotel *recommend[nhots];

        recommend[0] = new MelkshamHotel();
        recommend[0]->setAll(0.0,6,"Well House Manor");
        recommend[1] = new RegionHotel();
        recommend[1]->setAll(3.5,6,"New House Farm");
        recommend[2] = new MelkshamHotel();
        recommend[2]->setAll(0.8,3,"The Spa B&B");
        recommend[3] = new RegionHotel();
        recommend[3]->setAll(8.0,12,"The Bear");
        recommend[4] = new MelkshamHotel();
        recommend[4]->setAll(2.5,18,"Beechfield House Hotel");

        for (int k=0; k<nhots; k++) {
                Hotel *current = recommend[k];
                totrooms += current->getRoomcount();

                if ((current)->isWalkable())  {
                        cout << "Our training center is walkable from "
                        << current->describe << "\n";
                } else {
                        cout << "If you stay at "
                        << current->describe
                        << " you'll need transport\n";
                }

        }

        cout << "Total rooms listed: " << totrooms+0  << "\n" ;

        return 0;
        }
```

When I run that:

```
-bash-3.00$ ./accom4
Our training center is walkable from Well House Manor
If you stay at New House Farm you'll need transport
Our training center is walkable from The Spa B&B
```

```
If you stay at The Bear you'll need transport
If you stay at Beechfield House Hotel you'll need transport
Total rooms listed: 45
-bash-3.00$
```

- You're told you'll need transport from The Bear and New House Farm because they're Region Hotels and you always need transport from Region Hotels.
- You'll also need transport from Beechfield because it's over 1 km from us, even though it's a Melksham Hotel.
- You'll not need transport from Well House Manor nor from The Spa because they're both Melkshamhotels that are with the 1km range that's the walking limit for that class of accommodation.

Good. Tested, works. Further derived classes could easily be added, and other applications written to use the same family of classes.

Notes on the code

Look at this:

```
        Hotel *recommend[nhots];

        recommend[0] = new MelkshamHotel();
        recommend[0]->setAll(0.0,6,"Well House Manor");
        recommend[1] = new RegionHotel();
        recommend[1]->setAll(3.5,6,"New House Farm");
```

You'll declare an array to hold pointers to members of the base class as you really don't want to define an array of "nhots" actual hotels. (I hope you remember your C Pointers.)

You then set up each member of the class that you want using the **`new`** keyword. New explicitly calls the constructor (if you've not defined one, a default one that created all the elements you've defined for each member, but does not populate them, is called).

`new` returns a pointer to an object, be it a Melkshamhotel or a RegionHotel, and that can be stored in our array of hotel pointers. However, the syntax for accessing the members through the **`.`** operator then looks much more complex; I have to write:

`(*(recommend[3])).setAll(8.0,12,"The Bear");`

A new operator, the **`->`** operator, is provided so that you can simply write:

`recommend[3]->setAll(8.0,12,"The Bear");`

instead

heap v Stack variables

Another big advantage of using pointers and creating the objects with **`new`** is that they're stored in a memory are known as the **`heap`**, an area which is retained as long as the members are referenced there in some way.

The alternative store for variables is on the **`stack`**, an area which builds up as you enter blocks of code, and shrinks back to lose that area as you then leave the block of code. So a variable created on the stack is lost when you exit the block in which it was created, even if it's still referred to from elsewhere.

Placing a variable on the stack when it should be on the heap leads to broken pointers, and broken pointers lead to indeterminate results when you run your code, and the dreaded **segmentation fault** message that's basically C and C++'s way of saying "oops, that can't be right".

Two extra pieces of terminology – a class that contains the declaration of virtual functions and is never instantised itself is known in some OO languages as an **abstract class**. And an extra level of class that lies between a base class and some of the final derived classes, with the purpose of holding code that's needed by some but not all of the derived classes is known as a **mixin**. It gets even more interesting when you learn that a class can actually inherit from several other classes if you want it to. Multiple inheritance that we had better leave for a later chapter.

Exercise

We’ve covered a lot of topics there in Polymorphism. Following our example quite carefully, add another derived class called radio as a derivative of your Chronometer class, and ensure that you have the appropriate virtual methods in Chronometer.

Create an array that contains (pointers to) both Analog and Radio objects, and ensure it’s done in such a way that you don’t get memory fault problems.

Write an application that calls up both Analog and Radio chronometers, and reports back to you on how much each of them gains and loses in a year; now you, too, are using Polymorphism.

20

Further C++ Object Oriented Features

Multiple inheritance, references and pointers, optional parameters, multiple constructors and other methods of the same name ... even redefining of operators. C++ supports them all, and although there are some features here that you won't use in every function, you had better, at the least, be aware of them

20.1 References and pointers

C++ Supports **pointers** in the same way that C does.

- Precede a variable with an **&** on the right of an expression, and you're asking for the "address of" a variable, i.e. a pointer to it
- Precede a variable with an * on the right of an expression, and you're asking for the contents of a variable, i.e. what is pointed to.

C's pointers and their uses have been around for a long time, and they're not easy to pick up at first, but they're extremely powerful.

In addition to pointers, C++ lets you define references. A reference is, in essence, an alternative name or alias for a variable. You declare references using an **&** character in the declaration, and then you can simply use the reference name as a pseudonym for the target variable. Thus:

```
#include <iostream.h>

/*
References and Pointers

A pointer is a variable that holds the address of another
variable (object).

A reference is an alias to another variable (object)

*/


int main () {

        int weight = 96;
        int *pWeight = &weight;
        int &rWeight = weight;

        int another = 123;
        int athird = 770;

        cout << "weight " << weight << endl;
        cout << "pWeight " << pWeight << endl;
        cout << "*pWeight " << *pWeight << endl;
        cout << "rWeight " << rWeight << endl << endl;

// Note - following line does NOT reassign rWeight; it
// assignse a new value to the variable that rWeight references

        rWeight = another;

        cout << "weight " << weight << endl;
        cout << "pWeight " << pWeight << endl;
        cout << "*pWeight " << *pWeight << endl;
        cout << "rWeight " << rWeight << endl << endl;


// Following line changes pWeight only; it's NOT an alias
// so it leaves weight and rWeight alone

        pWeight = &athird;
        cout << "weight " << weight << endl;
        cout << "pWeight " << pWeight << endl;
        cout << "*pWeight " << *pWeight << endl;
        cout << "rWeight " << rWeight << endl << endl;

        return 0;
        }
```

When we run that, we get

```
-bash-3.00$ ./refdemo
weight 96
pWeight 0xfee9d294
*pWeight 96
rWeight 96

weight 123
pWeight 0xfee9d294
*pWeight 123
rWeight 123

weight 123
pWeight 0xfee9d284
*pWeight 770
rWeight 123

-bash-3.00$
```

You'll notice that when I *appear* to reassign my reference (i.e. change it to point at another variable), I don't in fact end up doing so. I end up changing the variable that it points to. Thus, I change rWeight and I end up changing weight and *pWeight too.

In contrast, when I change pWeight, I'm changing what it points to and although *pWeight prints out with a new value, with the base value containing a slightly different memory address, I have not changed weight nor rweight in this case.

There's a convention to precede reference variables with an "r" and pointers with a "p" in some circles; it's certainly an approach which, if used consistently, can help readers of your code no end.

A practical use of references

Although you cannot change which variable a reference refers to in C++, you can assign a new reference variable using a name that's previously been used but has gone out of scope at run time. Thus, you can use a reference as a shorthand for an array member in a loop. Here's an example that shows this technique in action; I've added the equivalent code using a pointer to so that you can, once again, compare and contrast.

```
#include <iostream.h>

/*
 References and Pointers for each array member

*/

int main () {

        int weights[] = {96,98,100,99,98,95};
        for (int k=0; k < 6 ; k++) {
                int &rWeight = weights[k];
                int *pWeight = &weights[k];
                cout << "rWeight " << rWeight << endl;
                cout << "*pWeight " << *pWeight << endl << endl;
                }

        return 0;
        }
```

And when we run that:

```
-bash-3.00$ ./r2
rWeight 96
*pWeight 96

rWeight 98
*pWeight 98

rWeight 100
*pWeight 100

rWeight 99
*pWeight 99

rWeight 98
*pWeight 98

rWeight 95
*pWeight 95

-bash-3.00$
```

C++ programmers tend to use references when they can, but they need to fall back onto pointers from time to time; pointers have an extra complexity, but extra power too.

20.2 Comparing objects

If you want to compare two objects, do you pass them both into a function? Well, sort of. You would have passed them in to a function in C, something like **strcmp**, and that would have worked perfectly well. But now that we're using the power of objects and polymorphism, it's no longer adequate. We need to pass in the second object to a method that's run on the first object.

In other words - structured style:

```
        biggest = larger(testcase, recommend[k]);
```

C++ (object oriented) equivalent:

```
        biggest = testcase->larger(recommend[k]);
```

With the big gain in the C++ case being that it can dynamically call up a different method called larger each time it's run, depending on what type of objects happens to be in the testcase variable. A larger hole is rather different to a larger calorific value, after all!

Within my comparator method, I need to refer to two different objects, but both of which have the same members, and it can get very confusing. The **this** keyword is provided to allow you to refer to the object on which the method has explicitly been called, and I've chosen to declare the variable called **that** as the second object instance's name; it keeps a symmetry of sorts and helps on understanding what's going on. Here's the code of larger:

```
Hotel *  Hotel::larger(Hotel *that) {
        if (this->getRoomcount() > that->getRoomcount()) return this;
        return that;
        }
```

When you come on to compare a whole array of objects of the same sort, you could specify one object as the one on which the method is run, then pass in the array of objects to the method. In practice, you would probably use a static method.

20.3 Static members

A static member of a class (also known as a class member rather than as an object member) is one that refers to the class as a whole; it's not dynamic in that it does not change the data it's using as you look at different objects in your class.

Static members are declared as static in the headers; I've added some static members to my hotel class, and the declaration of the class now reads:

```
class Hotel {
        public:
                virtual void setAll(float a, int b, char *c) {};
                virtual bool isWalkable() {} ;
                Hotel *larger(Hotel *that);
                void setRoomcount(int num);
                int getRoomcount();
                char *describe ;
                static int counter();
                static int nhotels ;
        protected:
                int roomcount;
};
```

This means that although there are separate methods called **`getRoomcount`** for each Hotel, there's only one method called **`counter`** and it applies to the class as a whole. That's sensible when you think about it; each hotel, potentially, has a different number of rooms, but there's only one number that tells you how many hotels there are.

Within the code that defines our hotel methods, we provide code for the static methods and also initialisation code for the static variables. Thus:

```
int Hotel::nhotels = 0;
```

and

```
int Hotel::counter() {
        return Hotel::nhotels;
        }
```

Note that a static method can only refer to static variables within the class (there's not this pointer, no concept of a current object), but a dynamic or object method can refer to the variables within its own (this) object and to static variables.

You typically don't have a lot of static variables in your classes, but from time to time they're useful.

The complete source code of the final hotel demonstration is in a file called *bighotels.cpp*, and running the comparator and static methods on the final demonstration, I get:

```
-bash-3.00$ ./bighotels
The biggest hotel is: The Bear with 19 rooms
Total hotels built - 5
-bash-3.00$
```

The full source code of all the files is available via our website - see
http://www.wellho.net/resources/C234.html

Exercise

Add a loop to your chronometer handler, using references to quickly and easily refer to each individual timepiece as you report how much time it gains or loses through a month.

Add code to tell you how many radio chronometers, you've created as you run your program.

20.4 Overloading methods, multiple constructors

You've already seen references to overloading, where a method in a derived class overrides a method that was defined in the base class. In C++, you can go further and define multiple methods of the same name in the same class. They're distinguished only by the number, type and sequence of parameters that they're called with.

This feature is perhaps most commonly used with constructor methods. There's often a requirement when a new object is created to have various options available to provide some of the possible information about it, but not everything.

Optional parameters

As well as providing multiple methods of the same name, C++ allows you to default the latter parameters in a method call. For example, if you've got a method that takes (up to) five parameters but you only supply four, the final parameter can be set to a default value. And if you only supply three, the final two can be defaulted.

The syntax for multiple constructors and optional parameters is shown in this new example, *pet.cpp*

```
#include <iostream.h>

/*

Demonstration of additional facilities such as:
* defaulted parameters
* multiple constructors

*/

class Pet {
        public:
                Pet::Pet();
                Pet::Pet(char *breed, char *name = "Anon");
                char * getBreed();
                char * getName();
        private:
                char *itsname;
                char *itsbreed;
};

Pet::Pet() {
        itsname = "undefined";
        itsbreed = "unknown";
        }

Pet::Pet(char *breed, char *name) {
        itsname = name;
        itsbreed = breed;
        }

char * Pet::getBreed() {
        return itsbreed;
        }

char * Pet::getName() {
        return itsname;
        }
```

```
int main () {
        int npets=5;
        Pet *menage[npets];

        menage[0] = new Pet("Hamster","Hammie");
        menage[1] = new Pet();
        menage[2] = new Pet("Dog");
        menage[3] = new Pet("Cat","Charlie");
        menage[4] = new Pet();

        for (int k=0; k<npets; k++) {
                cout << "Pet " << k << " is "
                << menage[k]->getBreed() << " called "
                << menage[k]->getName() << endl ;
        }
}
```

Running that:

```
-bash-3.00$ make pet
g++ -Wno-deprecated pet.cpp -o pet
-bash-3.00$ ./pet
Pet 0 is Hamster called Hammie
Pet 1 is unknown called undefined
Pet 2 is Dog called Anon
Pet 3 is Cat called Charlie
Pet 4 is unknown called undefined
-bash-3.00$
```

20.5 Overloading operators

What do you mean by "addition"? 2 plus 2 is four, but bread plus cheese is a ploughman's lunch. In C++, you can define what you mean by addition on objects that you create by overloading the addition operator, and indeed all the other operators if you wish.

Really, this is just syntactic icing; you're actually defining a method which you then shortcut to the "+" operator. Nice, though.

Here's an example that makes good use of an overloaded "+" operator. I want to be able to add cubes together for packing purposes. I've two laptop computers, a Powerbook and a Compaq, and I want to find the overall containing volume for them. So that's going to be:

- The larger of the two widths
- The larger of the two depths
- The sum of the heights.

So, here are the results wanted:

```
-bash-3.00$ ./cube
Compaq Evo N1050v 33 by 17 by 3
17" Powerbook G4 39 by 16 by 1.8
Both computers together 39 by 17 by 4.8
-bash-3.00$
```

From this main program:

```
int main () {
        Cube Compaq = Cube(33.0,17.0,3.0);
        Cube Powerbook = Cube(39.0,16.0,1.8);
        Cube Combo = Compaq + Powerbook;

        Compaq.report("Compaq Evo N1050v");
        Powerbook.report("17\" Powerbook G4");
        Combo.report("Both computers together");

}
```

The declaration of the + operator on a cube:

```
Cube operator+ (const Cube &);
```

The definition of how it's done

```
Cube Cube::operator+ (const Cube & second) {
        float newx = (x > second.x) ? x : second.x;
        float newy = (y > second.y) ? y : second.y;
        float newz = z + second.z;
        return Cube(newx,newy,newz);
        }
```

You can redefine most operators in C++, although the syntax varies a little for those such as ++ which are monadic (+ is diadic since it takes two parameters). Here's the complete source code of the cube example:

```
#include <iostream.h>

/*

#%% Operator overloading

Demonstration of operator overloading in C++

*/

class Cube {
        public:
                Cube::Cube(float inx, float iny, float inz);
                Cube operator+ (const Cube &);
                float Cube::getX();
                float Cube::getY();
                float Cube::getZ();
                void Cube::report(char *descr);
        private:
                float x;
                float y;
                float z;
};

Cube::Cube(float inx, float iny, float inz) {
        x = inx; y = iny; z = inz;
        }

Cube Cube::operator+ (const Cube & second) {
        float newx = (x > second.x) ? x : second.x;
        float newy = (y > second.y) ? y : second.y;
        float newz = z + second.z;
        return Cube(newx,newy,newz);
        }

float Cube::getX () {
        return x;
        }
float Cube::getY () {
        return y;
        }
float Cube::getZ () {
        return z;
        }

void Cube::report(char *descr) {
        cout << descr << " "
                << this->getX()
                << " by " << this->getY()
                << " by " << this->getZ()
                << endl;
        }

int main () {
        Cube Compaq = Cube(33.0,17.0,3.0);
        Cube Powerbook = Cube(39.0,16.0,1.8);
        Cube Combo = Compaq + Powerbook;

        Compaq.report("Compaq Evo N1050v");
        Powerbook.report("17\" Powerbook G4");
        Combo.report("Both computers together");

}
```

Makefile for this module

Here's the makefile for this module:

```
# makefile - module C234
# Compile / Load instructions for examples in this module

COPTS = -Wno-deprecated

r2:     r2.cpp
        g++ ${COPTS} r2.cpp -o r2

refdemo:        refdemo.cpp
        g++ ${COPTS} refdemo.cpp -o refdemo

pet:    pet.cpp
        g++ ${COPTS} pet.cpp -o pet

cube:   cube.cpp
        g++ ${COPTS} cube.cpp -o cube

# Passing two and more objects to a method. Static, etc

hotel.o:        hotel.cpp hotel.inc
```

```
        g++ ${COPTS} -c hotel.cpp

melkshamhotel.o:        melkshamhotel.cpp hotel.inc melkshamhotel.inc
        g++ ${COPTS} -c melkshamhotel.cpp

regionhotel.o:  regionhotel.cpp hotel.inc regionhotel.inc
        g++ ${COPTS} -c regionhotel.cpp

bighotels.o:    bighotels.cpp hotel.inc melkshamhotel.inc
        g++ ${COPTS} -c bighotels.cpp

bighotels:      hotel.o bighotels.o melkshamhotel.o hotel.o regionhotel.o
       g++ -o bighotels melkshamhotel.o regionhotel.o bighotels.o hotel.o


# Tidy up directory / remove all intermediate files

clean:
        @rm -rf bighotels refdemo pet cube r2
        @rm -rf *.o

all:    bighotels refdemo pet cube r2
```

Exercise

Either ...

At this point, you'll have learnt a substantial amount about Object Orientation and how it's implemented in C++, and your chronometer code is probably getting less and less easy to enhance.

Now that your tutor has taken you through the mechanisms of OO, he'll take you though some appropriate design techniques and will then set an appropriate exercise for the particular delegates on this course to pull everything together.

Or ...

Modify our cube example, adding in a multiply method that lets you multiply a cube by a number and returns a new cube that's a stack of that number of cubes.

Further modify the cube example so that it will construct with 1, 2, or 3 parameters. If specified with just one parameter, that same value should be used for the second and third values. With two parameters, the second parameter should be used for the third as well. *Hint - a cube with a negative dimension is clearly not right.*

21

Object Orientation: Design Techniques

This is the third and final module that introduces object orientation, and is applicable to any language with OO support. It is vital that applications are correctly designed from the beginning. Starting from first principles, we take students through the steps of modelling a system using informal methods, and we also introduce more formal methods such as UML.

Before you start to write code ... or classes ... or methods ... you need to give some thought to what you're going to write, every bit as much as you need to plan a holiday away from home before you actually set off on your travels.

The more your travels vary from the norm, the more planning you have to do ahead of time. Even if you're away for just a few days, some trips can be weeks or months in the planning; once the planning's done, the actual going is the easy bit!

Any new object oriented project needs a great deal of careful planning and review before a single line of code is written. The end user requirements need to be learned and considered, and indeed it's up to the analyst(s) doing the planning to have more foresight than the end user – to ask those searching questions, to think about what the future may hold, etc.

As you might imagine, there are a number of informal and formal techniques for planning and maintaining object oriented projects, and a number of tools of various sorts that are available to help you implement the techniques. Some are as simple as pencil and paper, others are complex and expensive pieces of software.

21.1 OO Design — some basics

Start with a good understanding of OO

If you're taking this module because you'll be working on a major OO project, remember that you do need to have a thorough understanding of Object Orientation, to know your encapsulation from your polymorphism from your privates, before you start. Otherwise you'll probably end up throwing out your first attempt!

Find a small training project. Design it, implement it. Learn OO that way first before you jump in with both feet. Then get a good understanding of the project.

One of the many OO authorities [Coad] came up with three development cycles:

Waterfall –

- Analysis
- Design
- Programming

Spiral –[1]

- Analysis, prototyping, risk management
- Design, prototyping, risk management
- Programming, prototyping, risk management

Incremental –

- A little analysis
- A little design
- A little programming
- Repeat

Although it may seem counter to what you're used to, and managers may worry at the lack of coding until late in the project, the author of this course has found the waterfall cycle very effective in some circumstances, with even the user manual written before a single line of new code exists!

And also a good understanding of the buzz words and design cycle.

From a web site .... Useful text, once you're a little way there! ...

OOA and OOD stand for Object-Oriented Analysis and Object-Oriented Design, respectively. OOA strives to understand and model, in terms of object-oriented concepts (objects and classes), a particular problem within a problem domain (from its requirements, domain and environment) from a user-oriented or domain expert's perspective and with an emphasis on modeling the real-world (the system and its context/(user-)environment). The product, or resultant model, of OOA specifies a

[1] The spiral model is often incremental and may waterfall if called for

complete system and a complete set of requirements and external interface of the system to be built, often obtained from a domain model (e.g. FUSION, Jacobson), scenarios (Rumbaugh), or use-cases (Jacobson).

A new Unified Modeling Language (previously Unified Method) is now being worked on by Grady Booch, James Rumbaugh, and Ivar Jacobson at Rational Software which should be made into a public standard, perhaps to be adopted by the OMG. The latest docs can be found online from the Rational home page.

The usual progression is from OOA to OOD to OOP (implementation) and this Universal Process Model roughly corresponds to the Waterfall Model [Royce 70]. See [Humphrey 89] and [Yourdon 92] for a few of many discussions on software life-cycle models and their use. Humphrey also details Worldy and Atomic Process Models for finer grained analysis and design in the Defined Process (see below) and discusses other alternatives to the task oriented models.

21.2 Informal Techniques

This section talks you through some of the things you need to think of as you design your Object Oriented system. We'll then go on to have a look at some of the more formalised techniques, concentrating on UML.

Micro or Macro?

You have to start somewhere. Do you start your design with the detail of the smallest data objects that you'll be using, or with the overall system and the classes that you'll be using in your main application?

Since there's an interface between each of the classes that you'll be using, some degree of flexibility if called for. Starting at the Micro level, you need to specify your user interface by analysing the next level as you specify the current level! Starting at the Micro level does mean that you can write test classes at each level as you write them. A series of thorough test harnesses, level by level, should ensure that problems are located in the detail before they can fester and spread.

A disadvantage of the Micro approach, though, is that you need to keep your eye on the overall target. It's very easy to specify and write classes, build them into other classes, but then discover that the classes you've written can't actually be bolted together to do the job that your end user needs.

Specifying classes and methods

What do you need to consider?

- Variables per instance
- Variables per class
- Method calling specifications

At the very least, write out lists and draw a diagram such as shown in Figure 2 and this will form a basis for you to then go on to some of the more formal methods later.

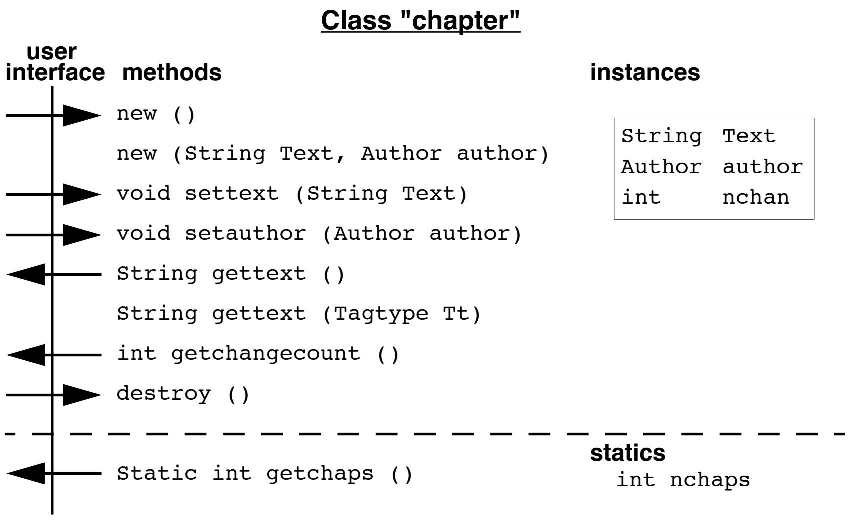


Figure 2 Within this diagram, we show: on the left, the user (caller's) interface to the class; on the right, the variables associated with each instance; at the top, instance methods and variables; at the bottom, static variables and methods.

Having drawn the basic diagram, the way is open for me to produce other diagrams; I can add lines showing which methods use which variables, for example as shown in Figure 3.

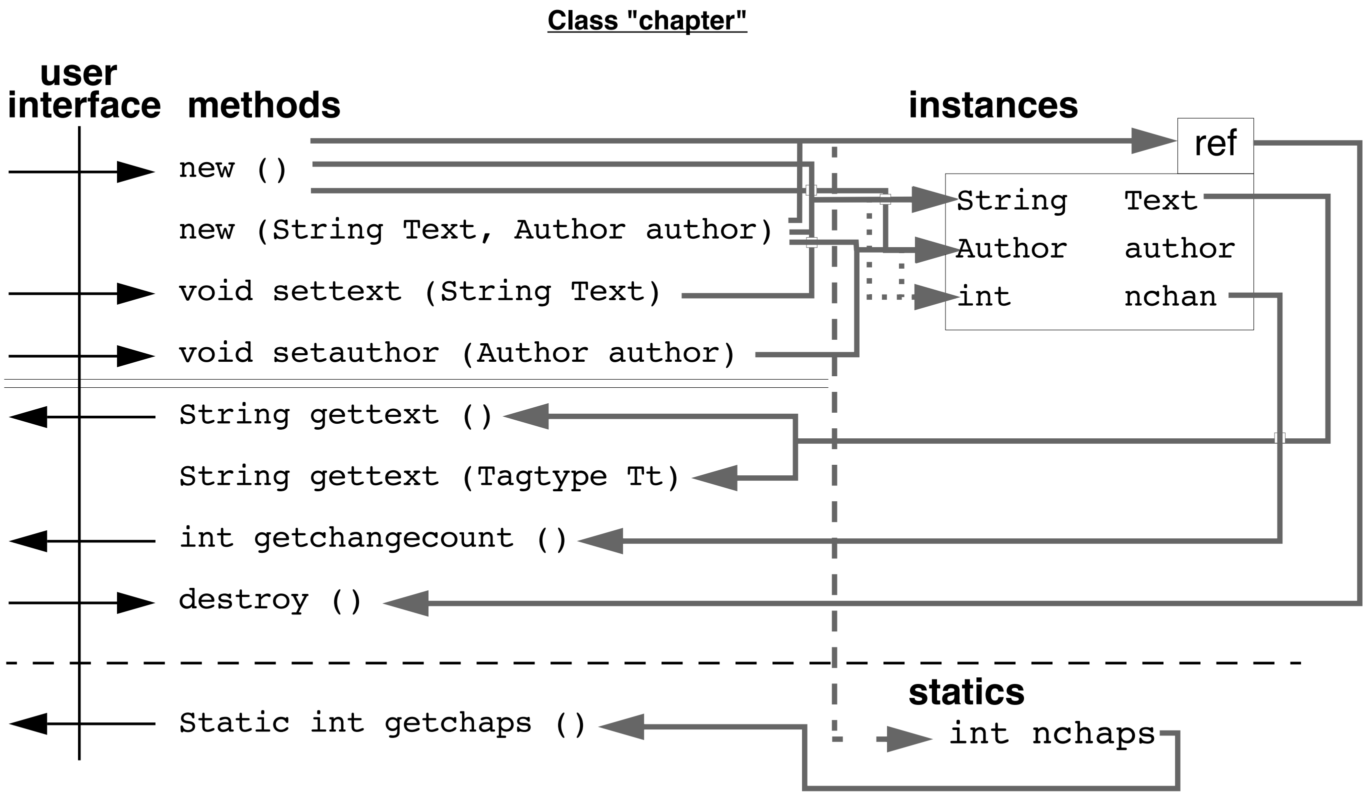


Figure 3 Solid lines on this diagram show which methods are directly reading or writing which particular instance variables, and dotted lines show how we're proposing to access other variables that are held within each instance or within the class.

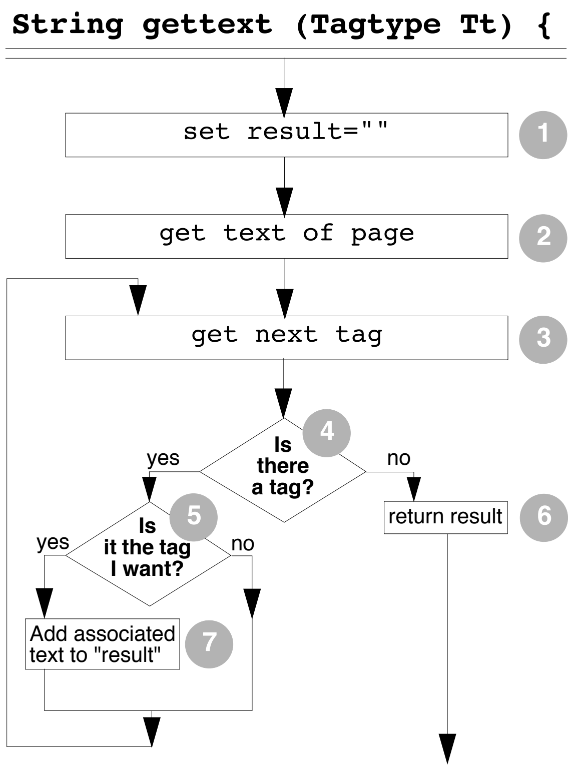
We're designing from first principles here, and if I follow this approach with a more complex class, showing extended information on the chart, my diagram will become overcomplex. That's when I need to start using a tool that lets me highlight certain features, or show certain aspects of the (as yet!) informal model I've started to build.

– What else can I see from the diagram already?

I can very clearly pick out certain flaws in my class design. In the example shown, my user interface allows me to set the author, but no methods have been provided to access that data!

If I want to see how a particular variable is used within the class, I can see all the influencing methods.

Figure 4 A flowchart of one of the `gettext` methods



In this particular example, I can see how the design neatly divides the methods into those which set variables and those which retrieve information – the **`set`** and **`get`** approach that's used by Java Beans. I've added a double-ruled line to my diagram to show you this division.

What can't I see from the diagrams I have?

– How the individual methods operate

I can't see how the individual methods work nor what algorithms are involved. For such work, I might use a traditional flowchart, I might write pseudocode, or I might even start writing real code in my target language. By using or calling a number of private methods within the class, such target code should be virtually self-commenting.

In the pseudocode examples that follow, the comment numbers apply to the box on the flowchart.

Pseudocode - near-Java

```
public String gettext (Tagtype Tt) {
            Stringbuffer result = new Stringbuffer(""); // 1
            Stringbuffer pagetext = new Stringbuffer(Text); // 2
            Taggedtext ttex;

                        while ((ttex = pagetext.gettagged()) != null) { // 3,4
                                    if (ttex.gettype().equals(Tt)) { // 5
                                                result.append(ttex.getchars()); // 7
                                    }
                        }
            String rr = toString(result);
            return (rr);  // 6
}
```

Pseudocode - near-Perl

```
sub gettext {

            my ($inst,$tt) = @_;
            my $result;                         # 1
            my $ttex;
            my @ttex_list = gettagged($inst -> "text");# 2
            foreach $ttex(@ttex_list) {         # 3, 4
                        my ($ttthis,$charsinthis) = splittagsection($ttex);
                        $result .= $charsinthis if ($tt eq $ttthis);# 5,7
                        }
            $result;                            # 6
            }
```

My earlier diagram could be extended to show private methods too. They would be placed to the right of the public methods and arranged in a hierarchy similar to the calling hierarchy.

– The State of an object

The diagrams that I've drawn so far are both views on my perception - my model - of the class that I'm writing. But there are also other aspects to consider that might be implicit prom or partly answered by what we've written so far, but are not obvious from either view or diagram.

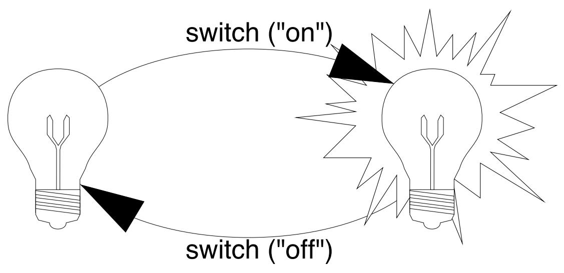


Figure 5 A state diagram to show a light bulb on and off

Let's consider one more feature of an object – an object can be in a particular condition, or have a particular state (or a series of states). A light bulb can be lighted or not lighted. I can draw a state diagram to show each of these states, and then add lines to my diagram to show how it moves from one state to another.

It's now easy to see how how you can move from one state to another, and also to spot any states that you can get stuck in. Indeed, a state diagram is an excellent tool for designing web sites and for using to follow visitors through the site from your logfiles.

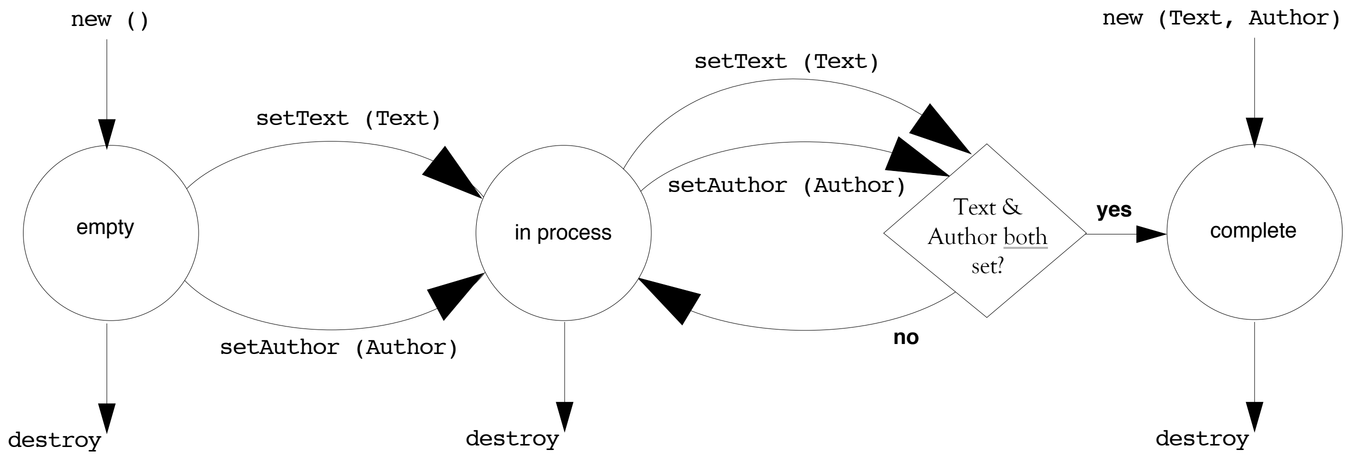


Figure 6 States of chapter object

Even with our "chapter" class, where we either have a chapter or we don't, a state diagram can be drawn (see Figure 6).

Specifying extended classes

What goes into the base class, and what's extended, and what's abstracted?

Your model needs to reflect the real life situation ...

Once again, you can extend the diagrams that you drew earlier for extended classes. Let's take our objects of type chapter and extend them into the diaryentry class.

The following diagram shows the original class in grey and the extended information in black.

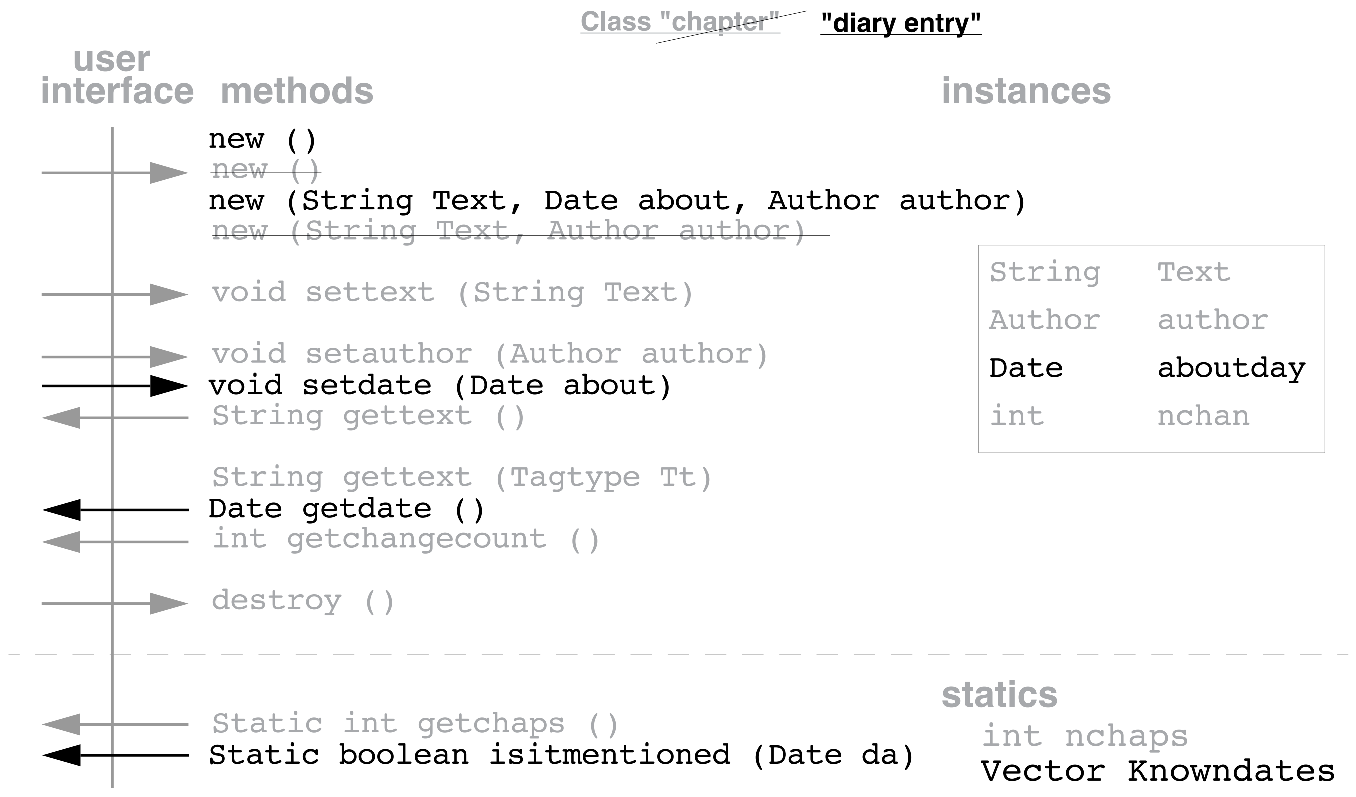

~~Class "chapter"~~ "diary entry"
user interface
methods
instances
new ()
~~new ()~~
new (String Text, Date about, Author author)
~~new (String Text, Author author)~~
void settext (String Text)
void setauthor (Author author)
void setdate (Date about)
String gettext ()
String gettext (Tagtype Tt)
Date getdate ()
int getchangecount ()
destroy ()
String Text
Author author
Date aboutday
int nchan
statics
Static int getchaps ()
Static boolean isitmentioned (Date da)
int nchaps
Vector Knowndates

Figure 7 Specifying extended cases

We've added the extra instance variables necessary to the diagram, and additional methods. Constructors are not inherited, so we've crossed them out (in reality, we wouldn't show the constructors of the base class at all on this diagram).

Methods of the base class which are overridden should be shown with the original method crossed through on the diagram and the new replacing it.

So why did the date object go into the extended class rather than the base class? Because it's something that relates to a diary entry but not to other types of chapter. "What date applies to this chapter?" (of this course) is a nonsense question. There might be other date members, such as date written, date first presented, date most recently presented, date next revision due, etc ...

If you find yourself adding (nearly) identical class members (methods and variables) to a whole series of extended classes, then you should consider moving those members into the base class.

If you find yourself writing a series of (base) classes which have nearly identical members, then you should consider creating a single base class and extending it if and when necessary to create each of what started off as independent classes.

If you find yourself overriding a method in every extended class of a base class, but needing to vary how you do the overriding, then you should be looking at abstract classes [Java].

You're not going to get your design right the first time you put pen to paper. You'll be changing designs, looking at different views, etc. Good reason to be using tools of the type we'll describe later, and more formal modelling languages so that you can work alongside others on projects and the whole team can have a clear understanding of what other team members are doing purely by looking at their models.

Clusters of classes

It's very rare for a single class to do everything you require. Within one type of object, you'll be using objects of another type.

If I'm writing a class of "chapter" objects (that could be extended to coursemodule, bookchapter, diaryentry, etc), I'll also need and call author objects, date objects (for when written, when published, etc), which will logically be separate objects rather than a part of the actual class. Indeed, the actual text of my "chapter" will be (in my example) a String object from the standard library of the OO language that I'm using, and I've used "Author" and "Tag" objects too!

The diagrams earlier in this section actually included the Author and Tag objects; not obvious to see from our black and white paper diagram, but using computerised tools, or using colour, such additional classes in the same cluster or elsewhere can be noticed, appropriate diagrams can be linked to, if need be all the variables within (say) the *Author* object class could be added to the diagram for a chapter, etc.

Generalise it out

"The best OO projects get simpler rather than more complex as they proceed". Amazingly, this has proved true many times over. And the more generalised your code is, the more it can be reused. Whoever wants to write two similar classes on two similar projects?

Programming and method standards

Before we look at some of the more formal methods which are around, a short comment on programming and design standards.

Perl especially, but also Java to some extent, allow you a flexibility over and above what you'll need within your particular organisation. You'll probably want to adopt particular programming styles and standards throughout your work or your team's work.

Programming standards as a whole are a quite separate topic beyond this module, but you should be giving thought to aspects such as ...

– Construct and populate separate?

As a matter of course, are we going to write constructors to set up an empty object, which we'll then initialise through a separate call, or are we going to write the two combined?

In our chapter example, we could elect to have all the work done by constructors:

```
    thischap = new chapter(String Text,Author author);
or  thischap = new chapter(File loaditfrom);
```

OR we could contruct an empty chapter:

```
    thischap = new chapter();
```

which we then initilased or populated as follows:

```
    thischap.init(String Text, Author author);
or  thischap.fromfile(File name);
```

Which is best? We could argue about this. Some authorities will tell you that the creation and initialisation are different jobs and should be done in separate calls. They'll point out that doing both in a single function means duplicated code if the object can be initialised in several ways.

Others will suggest you should always initialise objects as you create them, to avoid you accidentally calling inappropriate access methods on objects which haven't been initialised yet. In other words, if you combine the create and initialise, you save yourself a "state" that you have to consider in each of your called methods.

– **`get`** and **`set`**?

When accessing object methods, it's often sensible to separate out those which store information into the objects, from those which analyse the information in the objects, from those which retrieve information. Usually the last two groups are combined so that you have:

- methods that store information and
- methods that retrieve information.

It's sensible to name to a convention.

The human reader who knows the convention can instantly tell which group a method belongs to.

Tools that read existing programs and produce models from them can have that same knowledge.

Within some languages (and this applies to both Java and Perl) there's a facility called introspection which lets the language find out the names of all the methods.

The "de-facto" standard is to use words like:

`set`	to start method names that store information
`get`	to start method names that retrieve values
`isit`	to start method names that pass back boolean states.

and tools exist that make use of this convention; we're talking "beans" in Java.

Let's say that we're setting the author of a chapter. What set methods could we use?

Java

```
   setAuthor(authorobject);
or set("Author",authorobject);
```

and you'll find authors that use either (or both) of these standard conventions.

If you have a class that includes a large number of variables for each instance, many of which are just stored and retrieved, the second approach has some merit; the code author is saved the need to write or generate a very large number of methods, and indeed the single set method can be used to set several values at the same time:

```
set("Author",authorobject,"Text",contentstring);           Java
set("Author"=>$authorobject,"Text"=>$contentstring);       Perl
```

On the other hand, a series of explicit methods imposes a more controlled user interface onto the authors of the classes calling your methods. The more explicit that interface, the more robust the code is likely to be and the less change there is of the user managing to call something in a way not intended by the author.

Formal Methods

Objects need to be designed, and somehow those designs written down. It's often done in a visual language, which should be

- Accurate
- Consistent
- Easy to communicate to others
- Easy to change
- Understandable

We've come across the following:

Berard
BON
Booch
Coad/Yourdon
Colbert
de Champeaux
Embley
EVB
FUSION
HOOD
IBM
Jacobson

Martin/Odell
OOram
OMT
OOSE/Objectory
ROOM
Rumbaugh
Shlaer and Mellor
OPEN
UML
Wasserman
Winter Partners (OSMOSYS)
Wirfs-Brock

What a long list. And there have been many deeply held views about which are "good" and "bad" – the "method wars" within the object oriented community, if you like.

Problems that can occur if you use some of the OO methods described above. Very often coding starts too early, partly because programmers feel happier writing code than papers, partly because managers feel things are going well if code's being written. Also partly because some of the methods don't translate too well into the "small matter of programming", and those in the know can start coding early to sidestep problems!

UML is an attempt to solve some of these problems, and has the potential to become the de-facto standard.

21.3 Unified Modelling Language (UML)

UML is used to model systems. It comprises element which are categorized as Views, Diagrams, Model Elements and General Mechanisms, and can be extended to include other elements such as Stereotypes, Tagged Values and Constraints.

A number of tools that use UML and help you implement the mass of graphics, design, inter-relationships, etc., exist. One example is Rational Rose, now an IBM product since IBM completed the purchase of the Rational company in early 2003. Such tools will comprise some (or all) Drawing Support, Model Repository, Navigation, Code Generation, Configuration, Version Control and other associated tools.

Let's look at the elements of UML

Views

A model comprises:

- The use-case view which describes the functionality that the system should deliver as perceived by the user of the system – the external actors. It's central to the system; the objective of the OO project as a whole is to provide the functionality of this view.
- The logical view describes how the functionality shown in the use-case view is provided; it describes the structures such as classes and objects, and also shows how they collaborate with each other. This view is primarily of use to the designers and developers.
- The component view is a description of each of the implementation modules, (i.e. sections of code; classes; clusters) primarily used by developers. It also includes ancillary information, like who's responsible for what.
- The concurrency view. Addresses issues that are of concern with multithreaded systems. Divides the system into processes and sections that can run in parallel, and deals with synchronisation of resources and what can be done asynchronously.

• The deployment view shows the physical layout of the system, the computers and other devices concerned, and which components are deployed onto which devices.

Diagrams

Each view comprises a number of diagrams.

– use-case diagram

The use-case diagram shows the external actors, and how they connect to the cases that the system provides. Our earlier example using a chapter object didn't actually have a use-case diagram, as we were talking about chapters in isolation from any real use of those chapters. But we do need such a diagram to create a reason for us writing the chapter in the first case!

Using our chapter class in a complete system, we might see something like Figure 8.

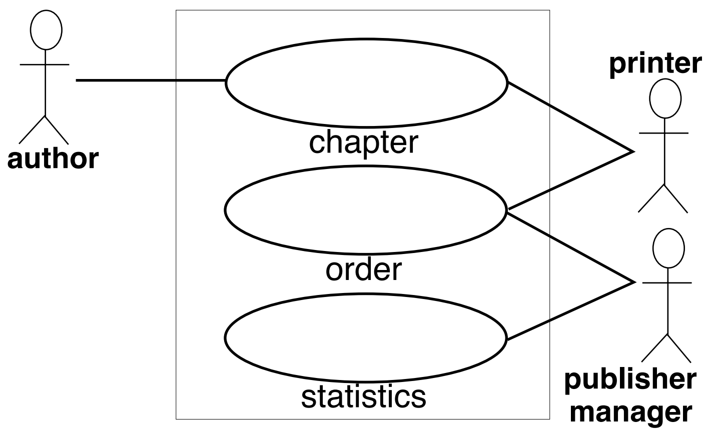


Figure 8 Our chapter class in a complete system

The author is purely concerned with the chapters. The printer is concerned with orders to print, and with the chapters; he needs to combine the two in order to undertake the functionality of his task. The publication manager isn't concerned with the actual chapter content; instead, he's concerned with orders for books, and order statistics.

– class diagram

A Class diagram shows how various classes relate to each other, and there are typically a number of class diagrams in a system. Very often, an individual class will occur on more than one class diagram.

Classes relate to each other in different ways. Conventionally, one class can extend another and then inherit from the base class.

Classes can be grouped into a cluster or a package. But also, Instances of one class can contain instances of another class.

An object of type chapter contains an object of type String and an object of type Author.

The new method (the constructor) for a chapter refers to 0 or 1 object of type File. Such an object of type File may be used by one or more objects of type chapter.

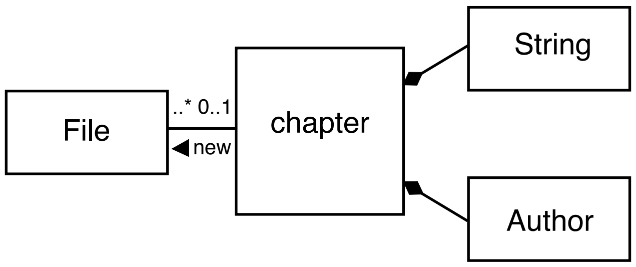


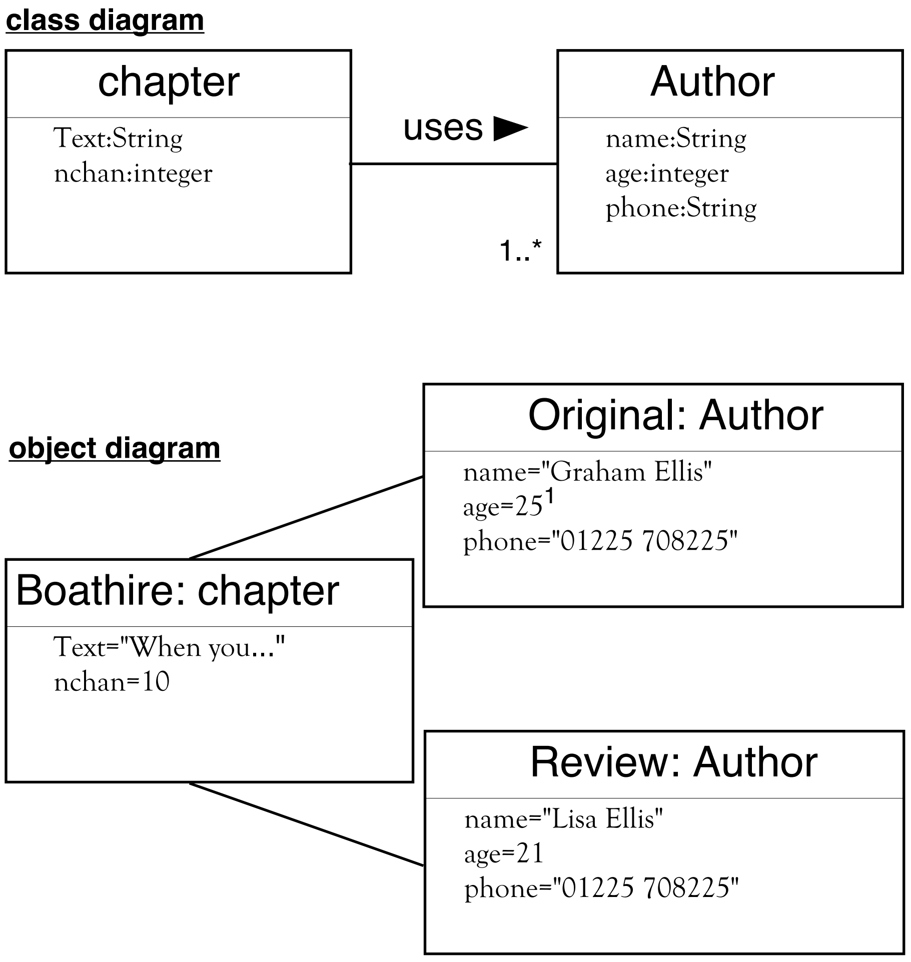
Figure 9 Partial class diagram for our chapter class

To this diagram, we could add our Tag and TagType objects as we developed our model as a whole ... and class diagrams can also include much more information such as names and types of primitives and system objects used within each class..

– Object diagram

An object diagram shows instances of a class, rather than the class as a whole. They can be used to clarify the design by showing a series of objects, or where a class contains multiple objects of another type. Our example *chapter* class contains a single Author. if we had it contain two authors, the we might have the class and object diagrams as shown in Figure 10.

Figure 10 class diagram, above, and object diagram, below, for our chapter class



It's no coincidence that these diagrams are similar to the informal diagrams we were drawing above. After all, UML is a modelling language, a way of specifying the topics that we were discussing in a formal language.

– State Diagram[1]

UML's State diagrams are a formalised version of the state diagram we saw earlier in this section, Figure 5 and Figure 6.

– Sequence and collaboration Diagrams

A sequence diagram is used to show a dynamic collaboration between a number of objects.

1 He lies.

Our chapter example would need considerable extension for us to find you an example, so shown in Figure 11 is an example of a sequence diagram (this one uses informal notation rather than the UML conventions, by the way).

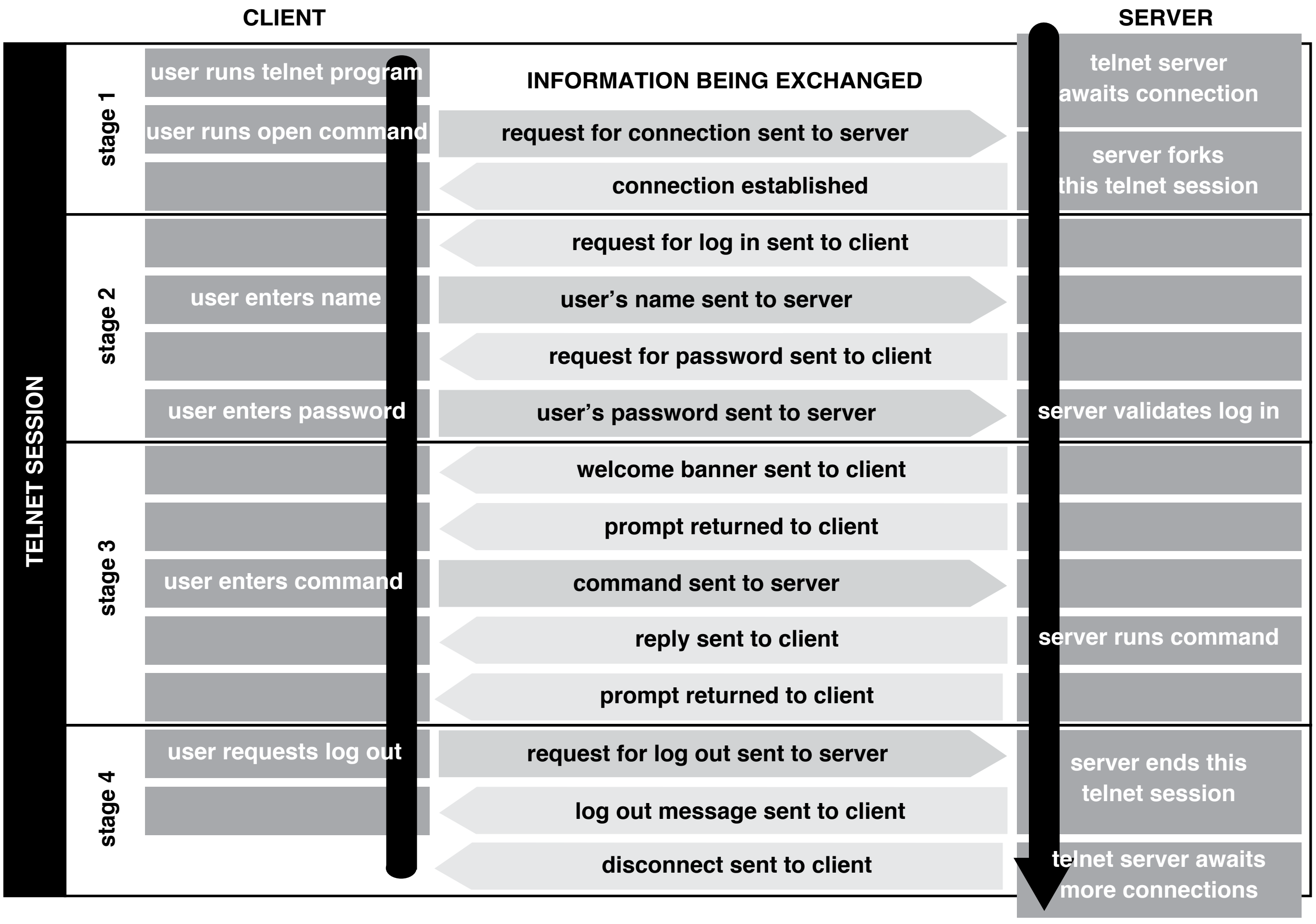

CLIENT
SERVER
TELNET SESSION
stage 1
stage 2
stage 3
stage 4
INFORMATION BEING EXCHANGED
user runs telnet program
user runs open command
user enters name
user enters password
user enters command
user requests log out
request for connection sent to server
connection established
request for log in sent to client
user's name sent to server
request for password sent to client
user's password sent to server
welcome banner sent to client
prompt returned to client
command sent to server
reply sent to client
prompt returned to client
request for log out sent to server
log out message sent to client
disconnect sent to client
telnet server awaits connection
server forks this telnet session
server validates log in
server runs command
server ends this telnet session
telnet server awaits more connections

Figure 11 sequence diagram using informal notation

A collaboration diagram is often an alternative to a sequence diagram; it shows how things work together. If the timing is important, you'll use a sequence diagram. If it's the nature of the relationship between the two things collaborating, use a collaboration diagram.

– Activity Diagram

A formalised flowchart, but using a differing notation and terminology to the informal one we used earlier.

Figure 12 An activity diagram

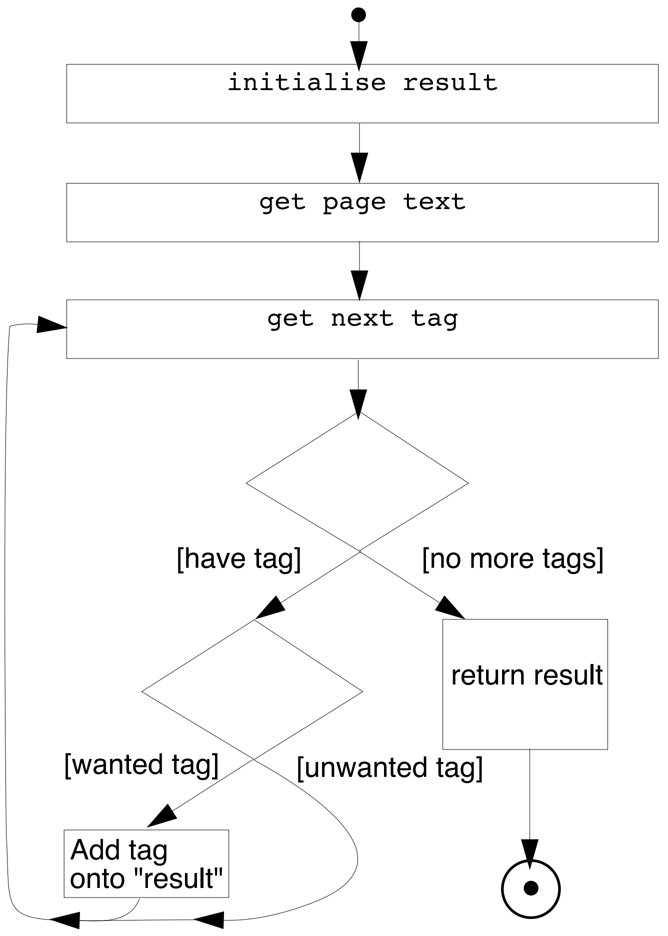
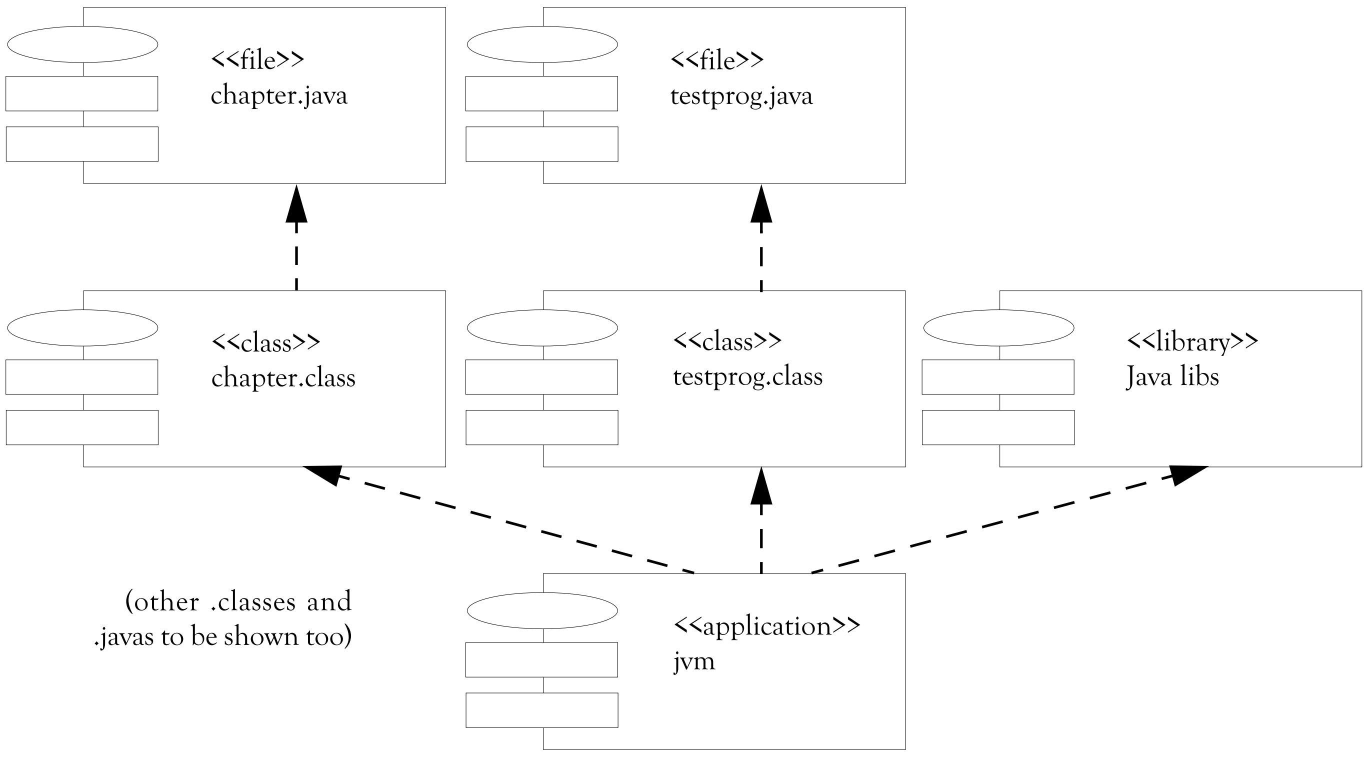


Figure 13 Our chapter class and test harness in Java

– Component Diagram
The layout of source, binary and executable files; shows dependencies, file types…



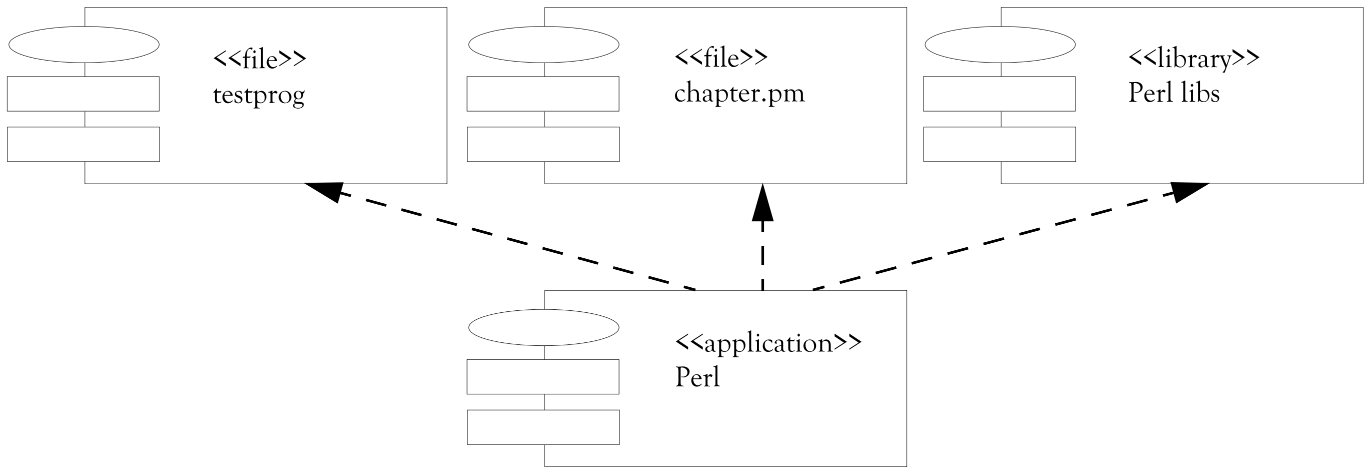


Figure 14 Our chapter class and test harness in Perl

Component diagrams often include documentation files too:

`<<page>>`	for HTML documents
`<<documentation>>`	for other forms of documentation

– Deployment Diagram

Physical architecture of hardware and software. In the case of our chapter class and test harness, everything will be running on the one computer and the diagram would comprise a single box. If we go the more complex use-case view we saw earlier, or to our telnet session that we showed in the sequence diagram, things can get to be more interesting, as shown in Figure 15.

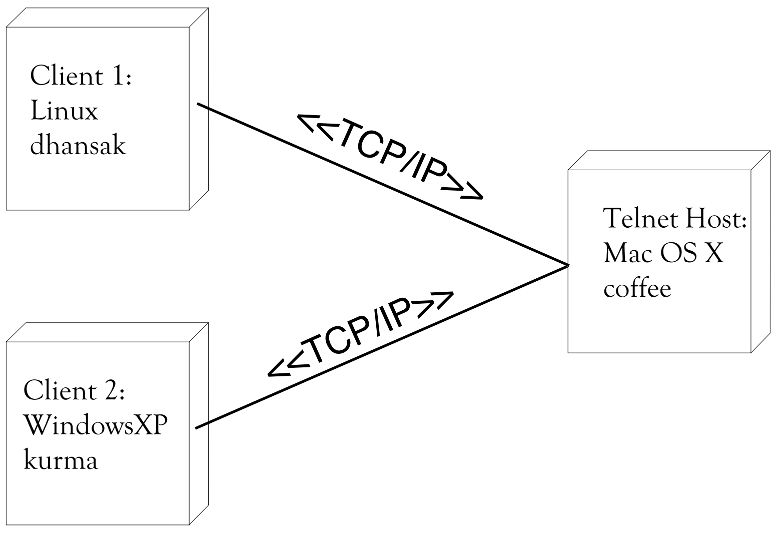


Figure 15 A sequence diagram of our telnet session

Model Elements

We've seen a number of types diagrams, and they are related. Model elements are shared between and used on multiple diagrams of different types, but there are rules concerning which elements can be used on which diagrams.

As well as the formal model elements, there are a number of general mechanisms which can be used in all diagrams. These general mechanisms are subdivided into types such as adornments, notes and specification.

UML can be extended or adapted by the addition of further facilities, such as stereotypes (a mechanism that allows a modelling element to be based on an existing modelling element), tagged values (where properties are associated with an element) and constraints (where restrictions are placed on values that are held in elements).

UML Summary

UML is just a way of formalising design and avoiding design pitfalls; whilst it's a help, you still need to be aware of overall design principles, of what your target language is capable of, and of what your user requires in the end!

21.4 Tools

UML is a tool that's designed to assist in the formalising of design over a number of designers. It certainly makes the designer think through his design, as you'll appreciate from what we've already covered in this module. But how practical is it?

For a simple system, diagrams can be drawn and correlated easily. But no system is that simple and with a series of diagrams to describe your system, a change will mean going through a pile of papers. There must be an easier way of doing it.

Yes, there are commercial tools available which let you define your model as a whole and then extract individual views and diagrams as required. Rational Rose[1] is perhaps the best known.

What's offered by such tools?

- Drawing support
- Model Repository
- Navigation
- Code Generation

And going beyond the the strict implementation of UML, there are many other facilities available in such tools:

- Configuration,
- Version Control,
- Prototyping tools
- Testbed tools (e.g. beans)

21.5 Project management and design issues

In this module, we've looked at some of the fundamental concepts of Analysis and Design of object oriented systems, in preparation for programming those systems in Object capable languages such as Java or Perl.

As well as informal techniques, which we studied so that you're aware of the basics, we had a brief look at UML, which lets you add a formality to the design process; in turn leading to advantages if you're automating the design through design tools, or if you have a project task.

Software design, though, is not easy and it needs careful planning and management. You need to get it right, you need to look ahead, you need to assess any risks you're taking and plan ahead of time for them.

Through the life of the project, you need to keep one eye on your goal and make sure you're not going off track, and one ear open to your user community to ensure that your implementation is fit for the purpose.

Two quotes from Grady Booch, Chief Software Engineer at Rational and often regarded as one of the fathers of OO Design:

"Remember that the class is a necessary but insufficient means of decomposition. For all but the most trivial systems, it is necessary to architect the system not in terms of individual classes, but in the form of clusters of classes."

"To be successful, an object-oriented project must craft an architecture which is both coherent and resilient and must then propagate and evolve the vision of this architecture to the entire development team."

[1] Company – Rational (now part of IBM); product – Rose

21.6 Extreme programming

Extreme programming is a design and programming methodology that's becoming increasingly popular. Its unit coding and unit programming approach fits well with an object oriented approach, but in extreme programming you're going to be encouraged to redesign and refactor the objects as the project progresses. And you're encouraged to involve the customer on day 1, and day 2 and every other day too.

Here are some of the keys ...

- User stories. Typically customer requirements of just a few sentences each, taking a couple of weeks each to complete. There may be between 50 and 100 user stories in a typical XP project
- Spike Solutions. Not wasting a lot of time writing code that's not effective in the end, and experimenting and trying out code in such a way that there's no huge loss if it has to be junked, then
- Refactoring. Redesigning or altering aspects in the light of experience. You are encouraged to refactor early and often, and not continue on down a less than ideal development route.
- Unit tests. Testing each element of the system in turn to see how it works before integration to ensure a series of robust components.
- Frequent Sequential Integration. Adding the units together frequently to ensure that they work correctly together, and adding them one at a time.
- Acceptance tests. Ensuring that each user story is met to the satisfaction of the user through rigorous testing.
- Iteration and release plans. A series of integrations and releases as the work proceeds, with feedback.

Planning

User stories are written.
Release planning creates the schedule.
Make frequent small releases.
The Project Velocity is measured.
The project is divided into iterations.
Iteration planning starts each iteration.
Move people around.
A stand-up meeting starts each day.
Fix XP when it breaks

Design

Simplicity.
Choose a system metaphor.
Use CRC cards for design sessions.
Create spike solutions to reduce risk.
No functionality is added early.
Refactor whenever and wherever possible

Coding

The customer is always available.
Code must be written to agreed standards.
Code the unit test first.
All production code is pair programmed.
Only one pair integrates code at a time.
Integrate often.
Use collective code ownership.
Leave optimization till last.
No overtime.

Testing

All code must have unit tests.
All code must pass all unit tests before it can be released.
When a bug is found tests are created.
Acceptance tests are run often and the score is published.

Exercise

You've got a database of golf courses containing the name of each course and a series of numbers – the length and par of each hole. Your application is to write a piece of code that goes through each of the golf courses in the database and reports back on the par and total length of each course.

Produce a series of diagrams (use case, object, class, activity, component, deployment) for this scenario. If useful, produce sequence and state diagrams.

22

IO in C++

Input and Output in C++

22.1 Standard input and output using cin and cout

```
#include <iostream.h>

/*

#%% Stream demonstration

* Input using cin

*/

int main () {
        float realnum;
        int inum;
        char firstword[10];
        char secondword[10];
        char myline[256];
        char ch;

        cout << "Integer please: ";
        cin >> inum;

        cout << "Float please: ";
        cin >> realnum;

        cout << "Integer was " << inum
                << " and float was " << realnum
                << endl;

// Note - with a string, separates at spaces
// Note - string to receive MUST be long enough

        cout << "String please (two words) ";
        cin >> firstword;
        cin >> secondword;

        cout << "Second word was " << secondword << endl;
        cout << "First word was " << firstword << endl;

// Reading character by character
// Can also use EOF to read to end of file

        cout << "Please enter a sentence ";
        while ((ch = cin.get()) != '.') {
                cout << "igot .. " << ch << endl;
                }
        while ((ch = cin.get()) != '\n') {}; // Tidy to line end

// Read in line by line mode

        cout << "Please enter three lines:\n";

        for (int k=0; k<3; k++) {
                cout << k+1 << ": ";
                cin.get(myline,256);
                ch = cin.get(); // Get termination from buffer
                cout << "You said: " << myline << endl;
                }
}
```

```
-bash-3.00$ ./ioex
Integer please: 33
Float please: 12.4
Integer was 33 and float was 12.4
String please (two words) The test.
Second word was test.
First word was The
Please enter a sentence igot ..

The test.
igot .. T
igot .. h
igot .. e
igot ..
igot .. t
igot .. e
igot .. s
igot .. t
Please enter three lines:
1: First line
You said: First line
2: Second line
You said: Second line
3: third
You said: third
-bash-3.00$
```

22.2 Reading and writing files

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

/*

#%% File handling with C++ streams

* Opening input and output files
* Reading from / writing to files

*/

int main () {
        char myline[256];
        int lc = 0;

// Opening file - with an extra "append" flag

        ofstream outfile("demo.txt",ios::app);

// Checking that a file open has worked
```

```
        ifstream infile("stdcodes.xyz");
        if (! infile) {
                cerr << "Failed to open input file\n";
                exit(1);
                }

// Parsing a file line by line

        while (1) {
                infile.getline(myline,256);
                if (infile.eof()) break;
                lc++;
                outfile << lc << ": " << myline << "\n";
                }
        infile.close();
        outfile.close();

        cout << "Output " << lc << " records" << endl;

}

-bash-3.00$ ./file01
Output 110 records
-bash-3.00$
```

23

C++ — Exceptions

Exceptions are a mechanism through which run time errors can be trapped easily.

23.1 The case for exceptions

When you run a program, things can go wrong – referred to as run time errors. And no amount of coding by the programmer can prevent these things because it's a user entering a string of text when a number's required, or a needed file having been deleted, or a network connection that's broken that causes problems.

In traditional coding, it's standard practice to check for as many of these errors as you can throughout your code, and this often results in a few lines of live code being wrapped in four or five times that number of lines of error checking. These catch most but still not all of the errors that may occur.

Exceptions are provided in many modern OO languages and they're in more recent C++ compilers, for example, as well as in languages like Python and Java. They let you write code where you don't write the detail of checking for each possible error yourself. Rather, you code for the working case and you enclose anything that may go wrong into a try block. Then you provide one or more catch blocks to set up actions that are to be taken if the try block failed to complete.

Great system; often less coding, and with a tendency to fail safe if error conditions that you've not explicitly coded for crop up.

Here's a short program – *prob.cpp* – which fails if our user enters a character string less than 20 characters long:

```
#include <iostream>

using namespace std;

/*
Exception handling - why we need it

This program deleted characters 10 to
20 of a string.  If the string isn't long
enough ...

*/

int main () {
        string demo ;

        cout << "Please enter your details ";
        cin >> demo;

        demo.erase(10,10);

        cout << "Reduced to " << demo << endl;

        return 0;
}
```

Here's the problem:

```
-bash-3.00$ ./prob
Please enter your details ajhsjhdfshjsdfjh
Reduced to ajhsjhdfsh
-bash-3.00$ ./prob
Please enter your details ssdfdf
terminate called after throwing an instance of 'std::out_of_range'
  what():  basic_string::erase
Aborted
-bash-3.00$
```

23.2 Catching Exceptions

It's much neater to use exculpation than to try to fix the problems of that example by doing every check that would be necessary before any operation on – in this example – my string object.

Here are the sort of results that I want to see:

```
-bash-3.00$ ./except
Please enter your details uhsdf
Caught exception basic_string::erase
We could go on if we wanted
Although in this demo we'll exit nice
-bash-3.00$ ./except
Please enter your details jsdjksdfjksdjfkkjdsfsdf
Reduced to jsdjksdfjksdf
-bash-3.00$
```

(For sure, in a live example I would take some alternative action and not simply report the problem).

Here's the code:

```
#include <iostream>

using namespace std;

/*
#%% Exception handling - catches
*/

int main () {
        string demo ;

        cout << "Please enter your details ";
        cin >> demo;

        try {
                demo.erase(10,10);
        }
        catch (exception &why) {
                cerr << "Caught exception " << why.what() << endl;
                cerr << "We could go on if we wanted" << endl;
                cerr << "Although in this demo we'll exit nice" << endl;
                return 1;
                }

//
        cout << "Reduced to " << demo << endl;

        return 0;
}
```

24

Templates

If you're defining a whole family of similar classes or functions, you can save yourself a great deal of repetition by using C++ templates.

24.1 The case for templates

When you're defining classes, you'll often want to define a whole series of classes that are similar to each other, or a whole set of methods that vary just in a tiny detail; for example, a whole series of classes where just the input type varies.

Templates can be used to define a whole series of classes, or a whole series of methods and functions, based on a pattern.

24.2 Function template example

```
#include <iostream.h>

/*

Function templates

Function templates are a way of defining a whole
family of functions which are essentially the
same except that they work on different types

*/

// Define the template

template<typename T>
 void swap(T& x, T& y)
 {
   T tmp = x;
   x = y;
   y = tmp;
 }

// Write some code to make use of the template

int main () {

        int x = 6;
        int y = 12;

        char *left = "To the West";
        char *right = "To the East";

        swap(x,y);
        swap(left,right);

        cout << "x is now " << x << endl;
        cout << "y is now " << y << endl;

        cout << "left is now " << left << endl;
        cout << "right is now " << right << endl;
}
```

Running that, we get

```
-bash-3.00$ ./template
x is now 12
y is now 6
left is now To the East
right is now To the West
-bash-3.00$
```

Exercise

Index

Behind Learning to Program in C and C++: the author

Graham Ellis

Graham Ellis has a vast background in the computer industry. After receiving his degree in Computer Science in1976 from City University, London, he hasn't looked back.

Graham spent seven years leading a team developing cross-platform products (running on both PCs and Unix systems), from which he brings an appreciation of product specification, portability, standards and security.

Many highly technical staff prefer to remain "back room boys", but Graham has always enjoyed writing and presenting training courses. Since the first course he wrote and presented (on the Fortran programming language) in the late '70s, training has gradually accounted for more and more of his working time. He does, though, take care to leave time aside for outside interests, family, and for time to undertake "real work" in the subject areas in which he teaches. Graham believes in practising what he preaches!

Graham was a user of the Internet before it ever hit big, and, with his training and support roles, was ideally placed both to make practical use of the technology and to help others do the same.

Well House Consultants, Ltd

In 1995, Graham founded Well House Consultants Ltd to present training courses for others under contract and to develop his own training material. Well House Consultants specialises in "niche" courses, the sort of topics that other training organisations can't provide economically themselves.

Well House Consultants is a partnership between Graham and his wife Lisa, whom he first noticed on the Internet. Lisa comes from a graphic arts background and looks after much of the web site maintenance work, as well as the production of sales material and manuals, such as this one, and sales enquiries and bookings. Lisa can be reached by email at *lisa@wellho.net*, via our main phone number 01225 708 225, or by fax on 01225 707 126.

In spite of being busy, Graham aims to remain approachable. If you have any questions on this material that you can't resolve through normal channels, please feel free to post to his forum (*http://www.opentalk.org.uk*) where he aims to reply within a day, or send him an email (*graham@wellho.net*) and you should hear back within two days. If you're interested in booking places on courses, or seeing if we can help with onsite requirements, please email *enquiries@wellho.net* (that will get you a quicker response if Graham's out training), or visit our *web site at http://www.wellho.net.*

Thank you for attending this course!