

Natural Sciences, Part IB, Introduction to Computing

HANDOUT C++ Programming Tutorial

Christopher Lester and Hrvoje Jasak
Department of Physics

(based on earlier versions by David MacKay, Roberto Cipolla and Tim Love)

February 22, 2021

- The course homepage is : <http://www.hep.phy.cam.ac.uk/~lester/c++2021>

This document explains how to engage with the “Introduction to Computing” course within Part IB NST, Physics B. It explains what you are expected to learn, how you could go about learning it. It suggests a programme of work which you could set yourself (if you know nothing about C++) that could fill the term, but it explains how you can vary tasks if to do so would be sensible for you. The examples provided show how to write, compile, execute and test simple computer programs in C++. In the absence of lectures (due to Covid-19) some videos which support the handout are available on the web, as are instructions of where to get support from demonstrators.

Please read the overview and assessment sections of this handout as soon as possible.

Pages 6–40 are relevant to the first week of the course.

Pages 40–63 are relevant to the second and third weeks of the course.

Pages 63–79 are relate to the fourth week of the course.

The remainder covers what is left.

Contents

1	Outline	6
1.1	The aims of this course	6
1.2	The course has a strange assessment mechanism which expects different students to both do and learn different things from each other.	6
1.3	What you HAVE to do, in this course	7
1.4	Why some of the information you would need to master C++ is (deliberately) not this handout.	7
1.5	What the handout contains	8
1.6	Getting help from demonstrators	8
1.7	Feedback	10
2	Assessment via Self-Assessment	10
2.1	Why self assessment?	11
2.2	Assessment deadlines	12
2.3	Collaboration and team working during the learning process	12
2.4	Copying/re-use of source code	13
2.5	Warning relating to deadlines and zero-mark scores	13
3	Installing a C++ compiler and/or associated tools on your computer.	14
3.1	Apple Mac (OSX / macOS)	14
3.2	Windows	18
3.3	Linux	22
3.4	If programming an Arduino or Arduino clone	24
4	Editing, compiling and executing a simple program	26
4.1	A simple C++ program to add two numbers	26
4.2	Overview of program structure and syntax	26
4.3	The development environment and the development cycle	28
5	SUGGESTION FOR ASSESSMENT 1	30
5.1	Objectives	30
5.2	Getting started	30
5.3	Computing exercise – editing, compiling and executing a simple program that adds numbers	33
5.4	Exploring compiler error messages	35
5.5	Compiler warnings	36
5.6	Practice	36
5.7	Assessment	36
5.8	Instructions for submitting your self-assessed work	37
6	Variables and constants	40
6.1	Variable types	40
6.2	Declaration of a variable	40
6.3	Storage of variables in computer memory	41

7	Assignment of variables	41
7.1	Assignment statements	41
7.2	Arithmetic expressions	42
7.3	Precedence and nesting parentheses	43
7.4	Initialisation of variables	43
7.5	Expressions with mixed variable types	43
7.6	Declaration and initialisation of symbolic constants	44
8	Simple input and output	44
8.1	Printing to the screen using output stream	45
8.2	Input of data from the keyboard using input stream	45
9	Control Statements	46
9.1	Boolean expressions and relational operators	46
9.2	Compound boolean expressions using logical operators	46
9.3	The IF selection control statement	46
9.4	The IF/ELSE selection control statement	47
9.5	ELSE IF multiple selection statement	48
9.6	SWITCH multiple selection statement	50
9.7	The WHILE repetition control statement	52
9.8	Increment and decrement operators	53
9.9	The FOR repetition control statement	53
9.10	The DO... WHILE repetition control statement	55
10	SUGGESTION FOR ASSESSMENT 2	56
10.1	Objectives	56
10.2	Computing Exercises (suggestions)	56
10.3	Tips	57
10.4	Assessment	58
11	SUGGESTION FOR ASSESSMENT 3	59
11.1	Objectives	59
11.2	Computing Exercises	59
11.3	Assessment	62
12	Functions	64
12.1	Function definition	64
12.2	Example of function definition, declaration and call	65
12.3	Function header and body	67
12.4	Function declaration	68
12.5	Function call and execution	68
12.6	Function arguments	69
12.7	Another example	70
12.8	Passing by value or reference	71
13	Math library and system library built-in functions	72
13.1	Mathematical functions	73
13.2	Random numbers	73
13.3	How can I find out what library a function is in?	74

14 SUGGESTION FOR ASSESSMENT 4	76
14.1 Objectives	76
14.2 Computing Exercises	76
14.3 The bisection method	76
14.4 Notes on algorithm and implementation	77
14.5 Assessment	79
14.6 ‘Have I done enough?’	79
15 Arrays	80
15.1 Declaration	80
15.2 Array elements and indexing	80
15.3 Assigning values to array elements	81
15.4 Passing arrays to functions	83
15.5 Character arrays	83
15.6 Multi-dimensional arrays	84
15.7 Structures	87
15.8 Enumerated constants	87
16 Reading and writing to files	88
17 Direct allocation of arrays with “new”	90
18 Modularizing	91
19 Structures and packages	95
20 Formatted output	99
21 Notes concerning the remaining Assessments 5 and 6.	100
22 SUGGESTION FOR ASSESSMENT 5 : “PLANET”	101
22.1 Objectives	101
22.2 Task	101
22.3 Ideas for what to do	102
22.4 What to hand in	104
23 SUGGESTION FOR ASSESSMENT 6 : “BONKERS”	105
23.1 Objectives	105
23.2 Task	105
23.3 Ideas for what to do	106
23.4 What to hand in	107
24 SUGGESTION FOR ASSESSMENT 7 : “RECURSION” (optional!)	109
24.1 Objectives	109
24.2 Recursion exercises	112
A PLANET: step by step guidance	113
B An introduction to object-oriented programming and classes	116

C Further reading	118
D Objectives suggested for each week of assessment	119
D.1 Week 1	119
D.2 Week 2	119
D.3 Week 3	119
D.4 Week 4	119
D.5 Week 5 : “PLANET”	119
D.6 Week 6 : “BONKERS”	119
D.7 Optional week 7 : “RECURSION”	120
E Thirty useful unix commands	121

Part IB Introduction to Computing

1 Outline

1.1 The aims of this course

The aim of this course are:

1. to give you more exposure to `C++` (or occasionally other compiled programming languages) than you had before you started the course, and
2. to reward students who take the course with exam credit for having engaged with it,
3. to make it more likely that, after completing the course, a participant (particularly one who would not previously have considered doing so) might be motivated to write other programs of their own, either to test their understanding of bits of physics they might then be learning, or simply to solve problems they might encounter in life more generally,
4. to award credit to students who demonstrate good time-management and organisational skills.

This course is a part of IB Physics because:

- it is observed that a large number of physics students go on to apply for jobs or research degrees which require a broad experience of both interpreted **and** compiled languages. Exposure to a *compiled* language is therefore intended to complement experience students may already have with *interpreted* languages such as `python` and `matlab`.
- it is recognised that the learning of a new physics concept can often be made faster or easier once a student has the skills needed to write a small simulation to test his or her understanding of the concept, and
- it is observed that Cavendish faculty members and PhD students often require a broad base in programming skills (typically a mix of `python` and `C++`) in order to carry out tasks required in research.

1.2 The course has a strange assessment mechanism which expects different students to both do and learn different things from each other.

The course recognises that there is no such thing as a ‘typical’ student. Some students taking the course will have no prior experience of `C++` or compiled programming languages at all, while other students will already know both it and many other programming languages well. A highly prescriptive course which said “do this on week one, and then this on week two ...” would end up wasting the time of students who have a lot of experience, while going over the heads of those who have none.

To avoid falling into the above trap, this course is structured around something called ‘**self assessment**’. More details about **self assessment** are given in Section 2, but in short this model

sees **you** setting targets for **yourself**, based on **your own** level of familiarity with C++, and the time allotted to the course. You are then awarded credit not on the basis of your final level of attainment, but on your level of engagement with the aims of the course.

Through the use of **self assessment**, every student should find that the time spent on the course is of benefit to them. Students who are already high skilled can use the time spent on the course to gain even higher skills, or to stretch themselves in other useful ways. A student who is a complete beginner can use the time to make progress on just the basics.

A detailed framework of tasks and ideas for the six self assessments is provided in this document (including suggested physics objective for the extended physics tasks of the last two assessments) but students are given freedom to set themselves task that are different from those in this manual if they wish (or need) to do so.

99% of students will do the course in C++, but any student who believes that he or she might have a good reason to depart from using C++ is encouraged to contact the course organiser to see if permission to use another language can be granted.¹

No student should find the course taking up more time than is manageable, since the part of the definition of appropriate engagement is that the tasks students set themselves should not take more than two and a half hours. If a student finds his or her tasks take too much time, then that student is setting inappropriate tasks and should adjust them appropriately.

1.3 What you **HAVE** to do, in this course

The only thing you should have to do precisely in the manner described in the handout, is to hand in six pieces of self assessed work. The rules dictating how this must be done are explained in Section 2. Everything else in the handout is, to some extent, only ‘advisory’ or ‘suggestive’. For example, Sections 5, 10, 11, 14, 22 and 23 make suggestions of what you might do to structure the work to be handed in for each of those six assessments. Their content remains, however, a suggestion only. Many students end up choose different objectives (both harder or simpler) for full credit, provided they persuasively argue in favour of the choices they made when handing in the pieces of assessed work. Therefore: **do not be intimidated by the length of this handout – not all of it will be used by everyone!**

1.4 Why some of the information you would need to master C++ is (deliberately) not this handout.

It is a simple fact that majority of people who using programming skills to help them achieve research goals do not acquire those skills simply by following a set of sequential steps in a single prescriptive course which aimed to teach them how to program. In contrast, most programmers acquire (at least their initial) knowledge of how to program from a wide variety of sources which can include:

- friends, colleagues, supervisors and demonstrators,
- tips, instructions, tutorials and videos found online,
- google searches of:

– stackoverflow.com,

¹This be the case for students who believe they already know all there is to know about C++, or know so much about it that their time could be better spent learning about some other compiled language.

- cppreference.com,
 - cplusplus.com, and any
 - other programming question-and-answer sites,
- by generalising knowledge relating to a programming language already understood to another less familiar one.

In the long term it is far more useful to have acquired the ability to **teach yourself** any new programming language, than it is to learn a particular programming language in a particular teaching framework. Having such a skill allows you to acquire other languages in future, and to modify one's knowledge of a language as it (or its use cases) evolve with time. Accordingly, this course (and the handout in particular) does not expect to provide every piece of information you need to establish yourself as a C++-programmer. It is assumed that many people will find better resources on the web – and rightly so. New tutorials are being created all the time, and many different styles by many different authors – with at least one suited to every possible preference. **Google** will help you find them for C++ just as it will help you find them for any other language you need to learn in the future.

Human-based assistance is very important, and is provided by demonstrators in the department for person in this course. Undoubtedly, a person can provide the best help when you are thoroughly stuck, even if all the human does it tell you the right sort of **google** search string to do to get you out of trouble! But human-based assistance to one side, it is external online resources that you will need use to teach yourself future programming languages, long after you have left the university, so you may as well start to use them now, and get credit for having done so.

1.5 What the handout contains

In addition to explaining the **aims and assessment format**, the handout provides a tutorial introduction to computing and the C++ computer programming language, **aimed at a beginner with no prior knowledge of compiled languages or shell/terminal usage**. It aims to help such a person **install a C++ compiler** on their own personal computer, before showing them how to **write, compile, execute and test simple computer programs** in C++. It makes suggestions of how to display the results of those calculations graphically using **gnuplot**. The beginners whom the handout is aimed at are encouraged to work on the command line and with simple code editors (not large Integrated Development Environments, or IDEs) so as to get a better idea of where the boundaries between source code, compiler, debug messages, executables, input and output are.

1.6 Getting help from demonstrators

1.6.1 When can I get help?

There are five demonstrating/help sessions per week differentiated by day of week and morning and/or afternoon. Each lasts two and a half hours. Each student on the course has been assigned to one session per week based on his or her college and **a list of the days and times of days at which the sessions run has been published on the [course website](#), together with an indication of which colleges are attached to which days**. The first help session begins on Friday 29th January 2021. Morning sessions have been timetabled to start at 9:30am, and

afternoon sessions at 2:00pm. It is unlikely that those times will change during the term, but if they do then the times shown on the [course website](#) will be updated.

1.6.2 Can I change the day/session to which I have been allocated ?

It is possible to swap sessions if you find someone else to swap with - we aim to keep all sessions of similar size. Please contact the course organiser (see front of this handout) if you would like to do this as action will be required on their part to activate the swap in Microsoft Teams.

If you have swapped sessions, your session may no longer correspond to the one that most of those in your college go to. If you become unsure which session you are in, it is possible to rediscover the answer by looking in Microsoft Teams to see which *channel* you have access to in the IB Computing *team*.

1.6.3 How should I used the help sessions?

For each session (e.g. 'Friday PM') there is a dedicated Microsoft Teams *channel* of which you have been made a member, within the The IB Physics B Scientific Computing *team*, of which everyone on the course is a member.²

We recommend that if you attend a help sessions that you do so actively collaborating in a group of around two to five other students. Most of you would organise those groups within your college, but they could span multiple colleges if you want them to. **Those of you entering that way should create a group chat within Teams for yourselves.**³ During the help session, you should discuss your progress and problems with your peers within the private chat of the local group session you started. This chat will be visible only to the small number of those of you in it. You can use text messages in the chat, share screenshots, including video calls, screen screens, or collaborate any way you wish. If you encounter a problem that you cannot overcome after having discussed it with your peers privately, you can then go to the main (global) chat for that channel to ask for a demonstrator to join your private group to chat with you there. **Simply type a short message such as 'we could use help' in the global channel and wait.** As soon as a demonstrator is available, they will tell you who they are (there are more than one demonstrator per session), and then **YOU can then add that demonstrator to your private call or chat group.** They will join, discuss your problem with you in the privacy of your own group, and then when done will leave your personal group chat and return to the global one to see if others need help. The main (global) chat therefore acts as a queuing system, and the demonstrators should mark the messages they have dealt with, so everyone sees where they are in the queue.

1.6.4 What if the procedures need to change?

These procedures are new – mostly prepared to work around coronavirus restrictions, so they might need to be amended. In the event of changes, the course website will be the first to be updated. The demonstrators might add further instructions, and answers to common questions in the main chat of the session. If in doubt, simply post questions to the demonstrators there.

²Yes, the *team* was mis-named: it should have been 'IB Physics B Introduction to Computing'. But you know what it means ...

³See [here](#) for instructions on how to create a group chat.

1.6.5 For demonstrators

Demonstrators are encouraged mark the messages they have dealt with, so that everyone sees where they are in the queue. Demonstrators are also asked to note the instructions on how to leave a chat given [here](#).

1.7 Feedback

Your feedback on all aspects of this new course is welcome. Feedback given *early* is more valuable than feedback delivered through the end-of-term questionnaires. Early feedback allows any problems arising to be fixed immediately rather than next year!

Dr Christopher Lester Tel: +44 (0)1223 337232
Room 952, Department of Physics, Cavendish Laboratory
lester at hep.phy.cam.ac.uk

2 Assessment via Self-Assessment

As the term progresses, you are expected to hand in six pieces of work as as part of the assessment process of this course. A summary of what is expected is given in the bullet points below, while the rest of Section 2 provides more detail on those of the bullet points which need further explanation.

- In total there are six pieces of work which are to be handed in for assessment.
- The first four pieces of work (one per week) are worth **one mark each**.
- The last two pieces of work (covering the last two weeks of the course) are worth **two marks each**. The extra mark for each of these is to reflect the expectation they must have a *physics learning goal* in addition to the *programming* component which is present in all submissions.
- The maximum score for the entire course is therefore 8 marks ($8 = 1 + 1 + 1 + 1 + 2 + 2$).
- It expected that students will manage their workload by doing do one piece of work per week, making one submission in each of weeks 2, 3, 4, 5, 6, and 7, however there is nothing to prevent people submitting work early if they wish.
- The content to be handed in each week consists only of:
 - The usual metadata one would expect (CRSID, email address, name, etc).
 - A one to two hundred word description of the goal or learning objective the student set him/herself for that week, and any comments on how things went.
 - The source code (and or supporting material) that the student wrote or used when achieving the self-imposed objective.
- It is neither possible nor necessary to submit output (e.g. images or videos) generated by the programs which have been written.
- Each assessment task is set by the student for him/herself. Though ideas for self assessment tasks are given throughout the handout, ultimately the student is given a lot of freedom to set tasks that are appropriate to them. Nonetheless, each week's self-assessment task should meet the following minimal criteria:

- The task should take the student about two and a half hours to complete.
- Every task should set a programming-based learning objective. Each of the last two tasks should additionally set a physics-based learning objective .
- The task should be handed in by the relevant deadline.⁴ No work that is handed in late, or after an extension granted, is permitted to count for assessment.
- The task should be set at a level which is appropriate for the student in question, given his or her pre-existing skills and familiarity with C++ (or on rare cases some other a compiled programming language).⁵
- The nature of the task should clearly indicate that the student is trying to further their knowledge/experience in a compiled programming language.
- The task should be consistent with the reasons for self-assessment given in Section 1.2
- The handout (and lectures) provide examples of tasks which student could set themselves. Students may follow them exactly, make them harder, make them easier, or do things which are altogether different. If in doubt, ask the course organiser.

2.1 Why self assessment?

The self-assessment process is based on a desire to treat you as adults, and on the assumption that you are interested in learning, capable of choosing goals for yourself, and capable of self-evaluation.

There are different types of programmers. Some people like to get into the nuts and bolts and write programs from scratch, understanding every detail – like building a house by first learning how to make bricks from clay and straw. Other people are happy to take bricks as a given, and get on with learning how to assemble bricks into different types of building. Other people are happy to start with an existing house and just make modifications, knocking through a wall here, and adding an extension there.

I don't mind what sort of programmer you become. All these different skills are useful. I encourage you to use this course to learn whatever skills interest you. The exercises give opportunities for various styles of activity. Indeed, a small number of microcontrollers (arduino clones) are available to loan to students in case some of you would like to turn the course into a hardware-based one.⁶

Programming is an essential skill, just like graph-sketching. If you don't know how to sketch graphs, what do you do? – practice, read books, talk to supervisors and colleagues, practice some more. As second-year physicists, graph sketching should be second nature, and you should use it as a tool in all physics problems you study. Similarly, programming is a valuable tool for thinking, exploring, investigating, and understanding almost any scientific or technical topic. You should find that programming can help you with your 2nd year, 3rd year, and 4th year courses. This course exists to get you up to speed on programming, for the benefit of all your courses.

As you progress through the course, ask yourself “what have I not mastered?” and take appropriate action. Think, read, tinker, experiment, talk to demonstrators in the help sessions, talk to colleagues. Make progress, stretch yourself, try to have fun, and get credit for it.

⁴There are six assessment deadlines. (one for each of the six pieces of assessed work). Work must be submitted by the deadline to count. For use in emergencies only, there is a mechanism (see description elsewhere) which can be used to apply for an extension of 168 hours — which is a factor of 67.2 times greater than the 2.5 hours required to do the work.

⁵Contact the course organiser for permissions if you think this route might apply to you.

⁶Contact the course organiser if this extension appeals to you.

The point of the course should be that you gain useful skills you don't already have, not that you jump through hoops!

All self-assessed submissions will be made available to the IB examiners.

2.2 Assessment deadlines

The assessed component of the course begins on Thursday 28th January 2021, and then runs for seven successive weeks (from each a Thursday to the Wednesday following). Every student has access to demonstrators for at least one two-and-a-half hour help session in each of those weeks (albeit sharing that access with other students) as explained in Section 1.6. Attendance at the help sessions is not compulsory, but is recommended for those of you who feel you would benefit most. Regardless of whether students choose to go to help sessions or not, the assessment deadlines are chosen such that a person who *did* attend one help session per week could theoretically submit their work for assessment at the end of each of their help sessions.

The **nominal deadline** for the task in “Assessment 1” (pages 30-39) is therefore chosen to be 11pm on Wednesday the 3rd February 2021, as this is the last day of the 1st week of help sessions. Similarly, the work for Assessment 2 is due at 11pm on the next Wednesday, and so on. This is summarised in the following table.

Name	Pages	Help Sessions	Nominal Deadline	(extended deadline)
Assmnt 1	30-39	28th Jan - 3rd Feb	11pm 3rd Feb	11pm 10th Feb 2021
Assmnt 2	56-58	4th Feb - 10th Feb	11pm 10th Feb	11pm 17th Feb 2021
Assmnt 3	59-63	11th Feb - 17th Feb	11pm 17th Feb	11pm 24th Feb 2021
Assmnt 4	76-79	18th Feb - 24th Feb	11pm 24th Feb	11pm 3rd Mar 2021
Assmnt 5	101-104	25th Feb - 3rd Mar	11pm 3rd Mar	11pm 10th Mar 2021
Assmnt 6	105-108	4th Mar - 10th Mar	11pm 10th Mar	11pm 17th Mar 2021
(spare)	109-112	11th Mar - 17th Mar	NONE – optional!	NONE – optional!

You will note the additional column headed “extended deadline” which is always at least one week later than the nominal deadline. Very occasionally people are seriously ill, or affected by an unavoidable and unforeseeable event that makes it not possible to meet a deadline. If such a situation arises, an extension will automatically be granted to the “extended deadline” for no loss of credit.⁷

2.3 Collaboration and team working during the learning process

How you learn to program is up to you, but let me make a recommendation:

If possible, and particularly you don't currently like programming, try to work with someone else (socially distanced if necessary). *Try to set things up so that the weaker programmer does as much of the typing as possible*, and the stronger one mainly advises.

⁷It is not necessary to apply for this extension! Don't mail me to ask for one! Handing work in between the Nominal and Extended deadlines is *itself* treated as an application for extension, and each such application is automatically granted. Note that extensions beyond the extended deadline, even by seconds, will *never* be granted. Two weeks is more than enough time to find to attend a session and do the two hours of work. Similarly, acts of God (power outages, the room was locked, my computer died, my college network was down) in the period of the extended deadline are immaterial. The extension facility is there to allow for the possibility that such things happened prior to the *nominal* deadline. Going beyond the nominal deadline means you are on borrowed time. Good time management is a skill that this course rewards. By being given the prospect of “automatic” extensions, you are being given freedom that could allow you to shoot yourself in the foot. Don't allow this to happen.

Working in pairs greatly reduces programming errors. Working in pairs means that there is always someone there to whom you can explain what you are doing. Explaining is a great way of enhancing understanding. It's crucial to have the weaker programmer do the typing, so that both people in the pair understand everything that's going on.

Nonetheless: **at the end of the day, you must all self-assess *individually*, and you must submit *individual* electronic records of your work via the link on the [course webpage](#).** In your submission, acknowledge any collaboration you engaged in.

2.4 Copying/re-use of source code

When programming in real life, copying or re-use of source code written by others is strongly encouraged.

- Copying saves time;
- Copying avoids typing mistakes;
- Copying allows you to focus on your new programming challenges while making use of work previously done by others.

Similarly, in this course, copying may well be useful. For example, copy a working program similar to what you want to do; then modify it. Feel free to copy programs from the internet, so long as you explain what your source was and why you copied it. The bottom line is: “do you understand how to solve this sort of programming problem?” Obviously, copying someone else's perfect answer verbatim (or just with minor cosmetic changes) does *not* achieve the aim of learning to program. Always self-assess. If you don't understand something you've copied, tinker with it until you do. If necessary, set yourself further additional programming goals.

You should not copy anyone else's “additional programming goal”. You should devise this goal yourself, and solve it yourself. If you substantially copy some other piece of code but do not acknowledge its source, then the assessment submissions may be voided.

2.5 Warning relating to deadlines and zero-mark scores

Late work, or work that suggests the student did not engage with the spirit of the self assessment, or a submission which fails to justify why the nominated self-assessed task was appropriate for the student in question, scores nothing.

Historically, failure to submitting the self-assessment exercises by the relevant deadlines (see section 2.2) has been the single biggest cause of loss of marks for students taking this course. Do not leave your work to the last minute ... it is not unusual for late work to be invalidated through being late by mere seconds. You have been warned!⁸ The reason that the course is strict in this regard is that demonstration

⁸Everyone who follows the instructions and engages with the course sympathetically should expect to achieve full marks, and in a typical year approximately 80-90% of people taking the course do get full marks. This is because, by construction, anyone taking longer than two and a half hours per week on this course has misunderstood the self-assessment instructions, and every student should have two and a half hours per week to devote to it. Nonetheless, there is always a sizeable number of students who lose marks through late submissions or through failing to justify why their self-assessed task was appropriate for them. In addition, each year a small number of student spend far too much time on this course, and then complain in feedback about the time it consumed, presumably having failed to note the almost binary nature of the mark scheme, and that it is up to them to set themselves tasks of an appropriate difficulty. Do not end up as one of them, No one who follows the course instructions carefully, should need this course to consume more than two and a half hours of their time per week.

of time-management and organisational skills is a stated aim of the course (see Section 1.1) and because there is a clear and automated mechanism for applying for extensions (to account for Acts of God, *etc.*) (see Section 2.2). A late submission therefore demonstrates that a key learning objective for self-assessment was not met.

3 Installing a C++ compiler and/or associated tools on your computer.

There now follow some instructions that will help a student get his or her laptop or PC ready to start this course.

- Students having an Apple Mac should read Section 3.1.
- Students having a Windows PC should read Section 3.2.
- Students having a linux system should read Section 3.3.

All students should read Section 3.4 for instructions on how to set themselves up to use an arduino, if they decide to work with one. [It is not compulsory to do so!]

3.1 Apple Mac (OSX / macOS)

Quickstart

1. Open a *terminal session* using one of the following:
 - Open Finder, go to Applications, go to Utilities, double click Terminal.
 - Open Launchpad, search for Terminal, click it.
 - Open Launchpad, click Other, click Terminal.
2. Check whether you have a *compiler*:
 - (a) In the terminal, type *exactly*:

```
which g++
```

and press enter.
 - (b) If the terminal responds `/usr/bin/g++`, you have a compiler.
 - (c) If the terminal says `g++ not found`, you do not have a compiler. Type *exactly*:

```
xcode-select --install
```

press enter and follow all instructions.
 - (d) If the terminal responds with something else, double check your typing. If the problem persists, consult Google or a demonstrator.
3. Install a *package manager*. You have a few choices, but it is not recommended to install more than one.

- For Homebrew (more popular nowadays), go to <https://brew.sh>, paste the suggested command into your terminal session. Press enter and follow all instructions.⁹
- For MacPorts (older/more mature), follow the instructions at <https://www.macports.org/install.php>.
- Some other options exist, e.g. Fink, but the above two are supported in this course.

4. Install gnuplot using the appropriate command:

```
(homebrew)    brew install gnuplot
(macports)    sudo port install gnuplot
```

Detailed instructions

Apple provides a C++ compiler as part of the its so-called ‘`command line tools`’ package which supports (or can be provided by) Apple’s ‘`Xcode`’ application which can be freely downloaded from Apple’s ‘App Store’. If possible, you should probably aim to install just the ‘`command line tools`’ without first installing ‘`Xcode`’ (even though installing the latter will give you the former) because at time of writing¹⁰ the `Xcode` application is 12 GB in size!¹¹ You are recommended, therefore, to proceed cautiously, and to only install the minimum you need, and only to do that if it turns out that you don’t already have a C++ compiler installed. You may therefore wish to do the following:

3.1.1 Checking first if you already have a compiler installed

First determine whether you already have a C++ compiler installed. Do this by opening a terminal¹² and once your terminal is open, try typing either

```
g++ --version
```

or

```
c++ --version
```

into it. When I do the first of the options on my mac I see the following:

```
CGLMAC2-2:~ lester$ g++ --version
Configured with: --prefix=/Library/Developer/CommandLineTools/usr --with-gxx-include-dir=/Library/Developer/CommandLineTools/usr/include/c++/
Apple clang version 12.0.0 (clang-1200.0.32.2)
Target: x86_64-apple-darwin20.1.0
Thread model: posix
InstalledDir: /Library/Developer/CommandLineTools/usr/bin
```

⁹Thank you to Will Early for adding Homebrew instructions to this handout.

¹⁰November 2020

¹¹This is because it includes not just a C++ compiler, but also an entire software suite to allow users to develop applications for iPhones, iPads, and every other device in the Apple ecosystem. As a consequence of all this additional functionality which is not needed for this lecture course, a full X-code install can take a few hours, will take up a lot of valuable space on your laptop, and may even ask you to upgrade to the latest version of OSX.

¹²Google ‘**how open terminal osx**’ if you don’t know how to do this.

which shows me that I already have a C++ compiler installed. If you see something similar (or at least something not disheartening) then probably you already have a C++ compiler installed and can skip to Section 3.1.4.

However, if you find that both of those commands report ‘`command not found`’, or give some other discouraging messages, then you probably don’t yet have a C++ compiler and will have to get one in the manner about to be described.

3.1.2 Installing the ‘command line tools’ to get a C++ compiler

The instructions for installing the ‘command line tools’ can differ subtly between different versions of OSX, but most of them start off typing something like:

```
xcode-select --install
```

into a terminal, and then following whatever on-screen instructions then appear. These subsequent instructions usually appear in a dialog box that pops up, but in some systems they may appear in the terminal. Follow whichever instructions look like they will install the ‘command line tools’. If you are presented with a choice between (i) a way of proceeding that installs all of Xcode, or (ii) one which installs *just* the ‘command line tools’, then go with the latter, at least first, for as it is a lot less space hungry.

Once you have finished the above, check if you successfully managed to get a C++ compiler by trying the steps in Section 3.1.1 again. If you did succeed, you are done. If not, consider trying `xcode-select --install` again, to check you didn’t miss something, and maybe google ‘**install command line tools XXXX**’ where XXXX is replaced by your mac OSX version.¹³ Even better, ask a compsci or a friend (or even a compsci friend) to help you. If all such attempts fail, then try the next step.

3.1.3 Installing ‘Xcode’ if all else fails

The fall-back (but slow) option is to visiting the App Store and searching for (and then installing) the whole of the Xcode application. You can do this in just the same way as you would install any application from the App Store. Xcode is free. Once you’ve done it, try the `xcode-select --install` line again – and check that it indicates that the tools are now installed, and then check if you successfully managed to get a C++ compiler by trying the steps in Section 3.1.1 again.

3.1.4 Installing other tools

Once you have a C++ compiler you can code in C++ as explained elsewhere in this handout. However, some of the later parts of the handout recommend that you gain experience in plotting outputs of your C++ programs. The course is happy for you to make such plots in any plotting package you like, but the handout is written assuming you will do it in gnuplot. On a mac, to install gnuplot you could do one of the following three alternatives, according to your preference:

1. Follow the gnuplot installation instructions on [this web page](#)¹⁴; or

¹³(e.g. Big Sur, or Catalina or El Capitan, *etc.*)

¹⁴If you are looking at a printed copy of this handout, the url is https://csml-wiki.northwestern.edu/index.php/Binary_versions_of_Gnuplot_for_OS_X. If that web page no longer exists, please email the course organiser so that the handout can be updated. In the interim, try to find replacements by googling ‘gnuplot binaries for osx’.

2. (i) install MacPorts (home page is <https://www.macports.org/index.php>, installation instructions on <https://www.macports.org/install.php>), and then (ii) after MacPorts is installed, open a terminal and type:

```
sudo port install gnuplot
```

or;

3. (i) install Homebrew (home page and installation instructions at <https://brew.sh>), and then (ii) after Homebrew is installed, open a terminal and type:

```
brew install gnuplot
```

The first option is simpler than the second/third. However, the advantage of the latter options (installing a package manager, i.e. MacPorts or Homebrew) is that once you have MacPorts or Homebrew installed, installing any other package becomes trivial. For example, you could install `vim` with one of the two commands (for MacPorts and Homebrew respectively):

```
sudo port install vim
brew install vim
```

or install a large number of L^AT_EX packages with

```
sudo port install texlive-latex texlive-latex-recommended texlive-latex-extra
brew install --cask mactex
```

N.B. It is *highly* unrecommended to attempt to install both MacPorts and Homebrew simultaneously!

If you don't know the specific name of a package you want to install, but you are fairly sure it is related to, say, the word 'ruby' you could list all the potential package matches with:

```
port search ruby
brew search ruby
```

On my system the MacPorts command gives many lines including the following:

```
rb-xslt @0.9.9 (ruby)
  A Ruby class for processing XSLT

rb-zentest @3.10.0_1 (ruby)
  Automated test scaffolding for Ruby

rb-zip @0.5.7 (ruby, archivers)
  rubyzip is a ruby module for reading and writing zip files

rb19-abstract @1.0.0 (ruby, devel)
  define abstract methods in Ruby

rb19-activerecord @3.0.5 (ruby, databases)
  Ruby ORM
```

rb19-archive-tar-minitar @0.5.2_1 (ruby)

Ruby library and utility for POSIX tar(1) archive files

rb19-arel @2.0.9 (ruby, devel, databases)

Relational Algebra for Ruby

The output above might be sufficient to remind me that the name I was searching for was ‘rb-zip’ after which I could type

```
sudo port install rb-zip
```

to get the package I wanted.

3.2 Windows

While Windows is a “visual environment” by design, for the purpose of the course we will set up an environment with a terminal and command-line compilers. To do this, we will use the following packages:

- MinGW – Minimalist GNU for Windows;
- Gnuplot – a portable command-line driven graphing utility
- Notepad++ – a free source code editor

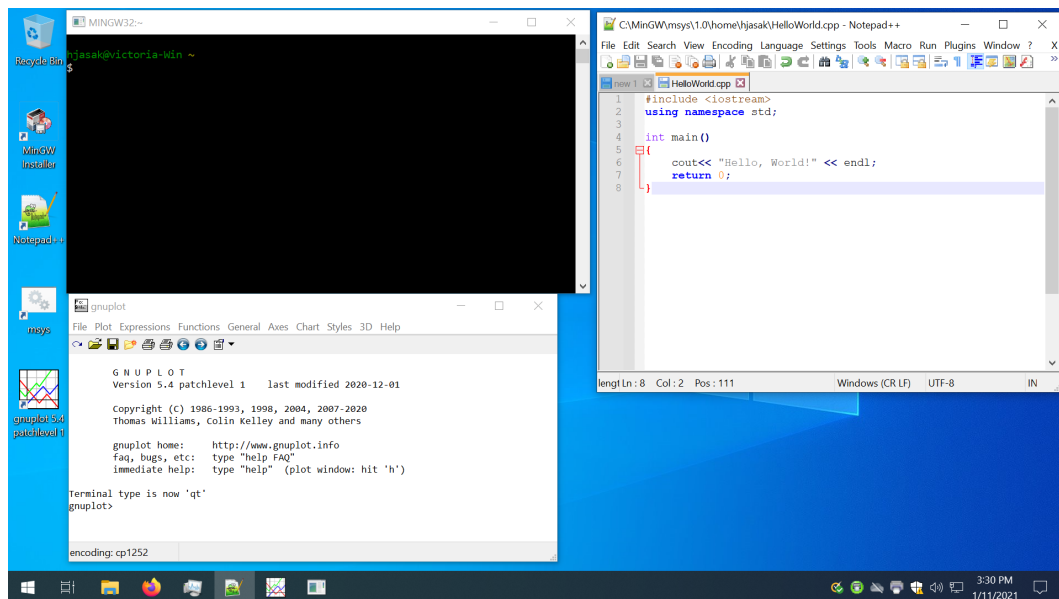


Figure 1: Windows environment.

For installation, you will need a Windows computer and an account with **Administrator** privileges. Assuming you are working on “your own computer”, this will be a default account you created. To check;

1. Open the Control Panel;
2. Click the User Accounts option’

3. In User Accounts, you see your account name listed on the right side. If your account has admin rights, it will say "Administrator" under your account name.

If you are an experienced software developer using Windows, you are welcome to use alternative setup.

3.2.1 Installing MinGW

Using your web browser, go to: <https://sourceforge.net/projects/mingw/files/Installer/> and download `mingw-get-setup.exe` file, Figure 2. instructions.

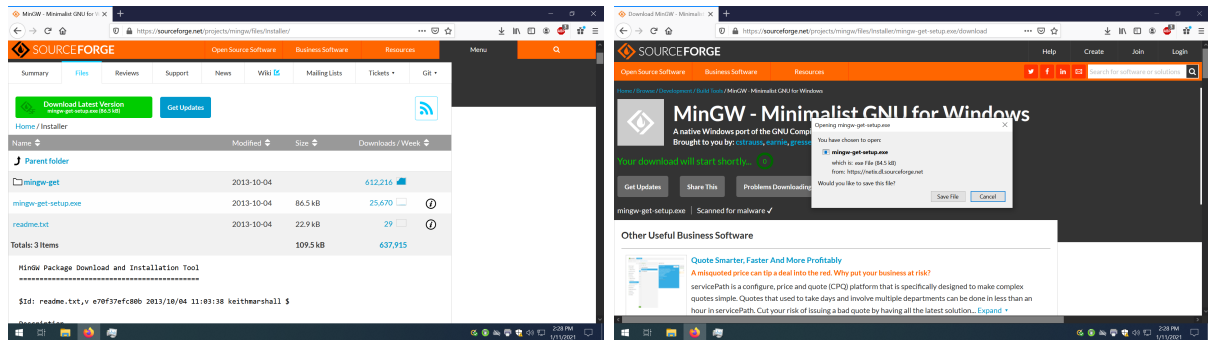


Figure 2: Downloading MinGW.

Open the file you downloaded, and follow the instructions, Figure 3.

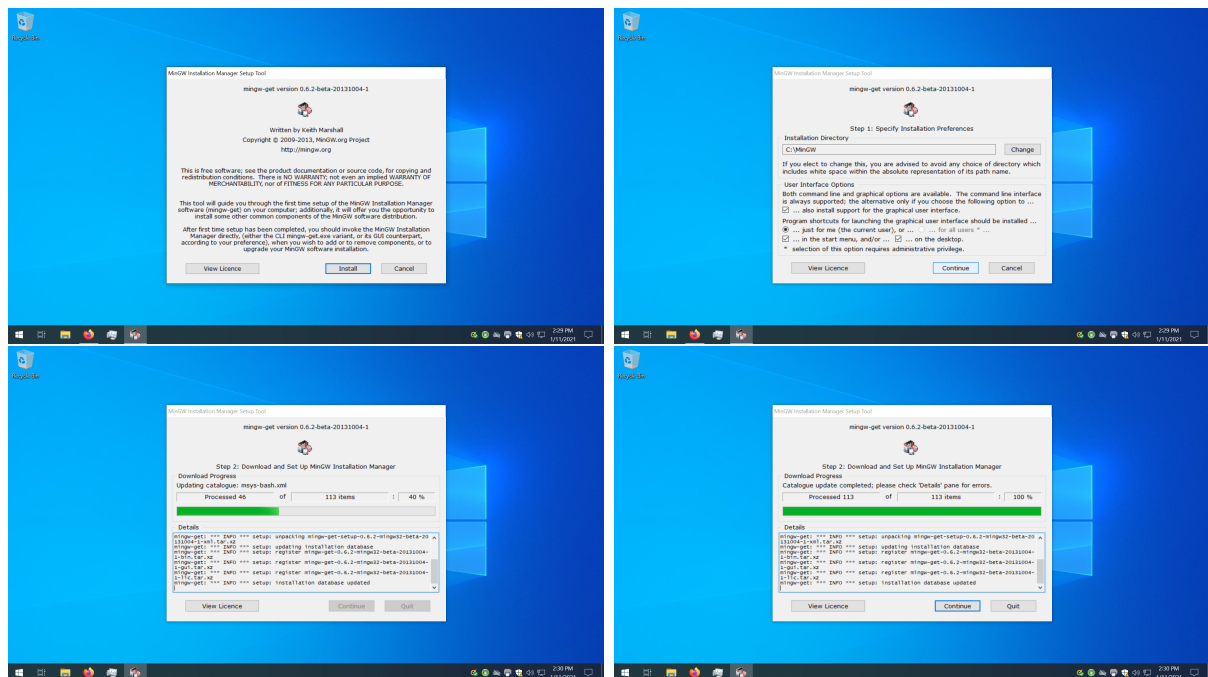


Figure 3: Installing MinGW.

When the choice of components comes up, Figure 4, please choose the following

- `mingw-developer-tools`
- `mingw32-base` (pre-selected)

- mingw32-gcc-g++
- msys-base (pre-selected)

Click on Installation -> Apply Changes to execute the installation.

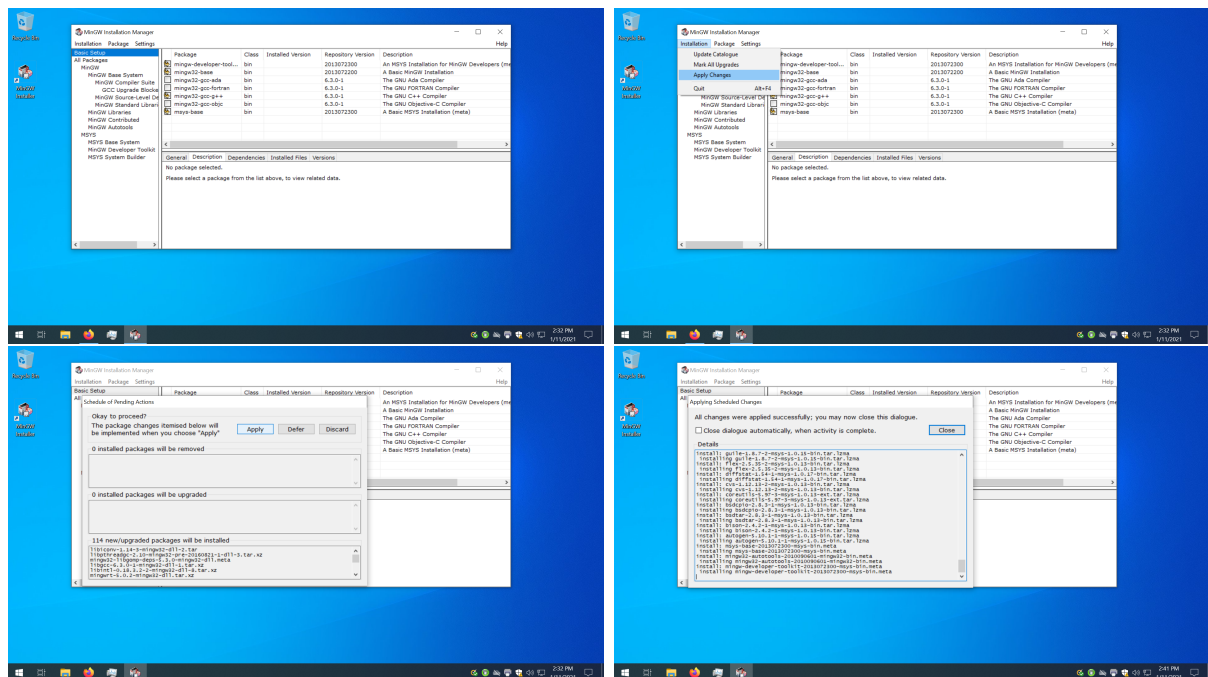


Figure 4: Selecting and installing MinGW components.

The files belonging to the MinGW installation are in `C:\MinGW`. MinGW will provide an interactive shell called `msys`, which can be found in

`C:\MinGW\msys\bin\msys`

For ease of use, I have made a shortcut to this file on my Desktop. Your own files created within MinGW will appear in the Windows filing system under

`C:\MinGW\msys\bin\msys`

both shown in Figure 5.

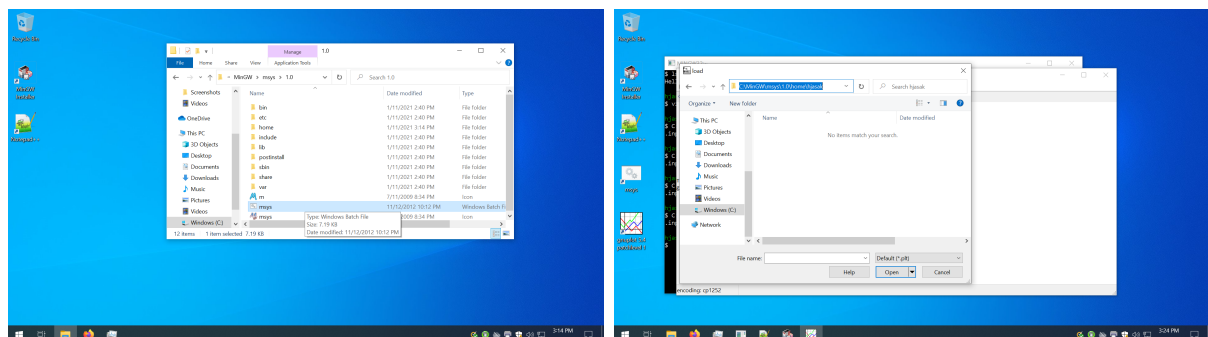


Figure 5: msys shell and home directory.

When you open `msys`, it will feel just like a Unix shell or terminal. To test things, type


```
which g++
g++ -v
```

to make sure the compiler can be found and it is working.

3.2.2 Installing Gnuplot

Gnuplot can be downloaded from <https://sourceforge.net/projects/gnuplot/files/gnuplot/>. Follow the usual Windows installation procedure, Figure 6.

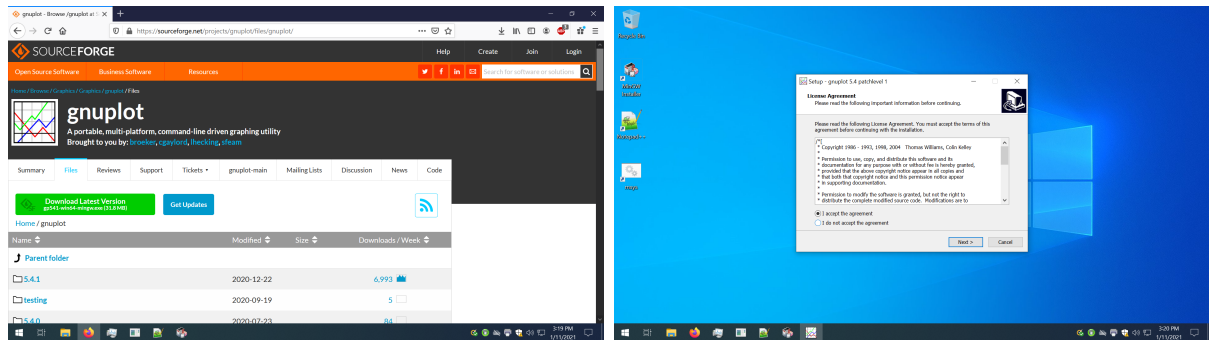


Figure 6: Download and install Gnuplot.

A few snapshots of Gnuplot in action are shown in Figure 7. Note the location of your files from msys when selecting data from files.

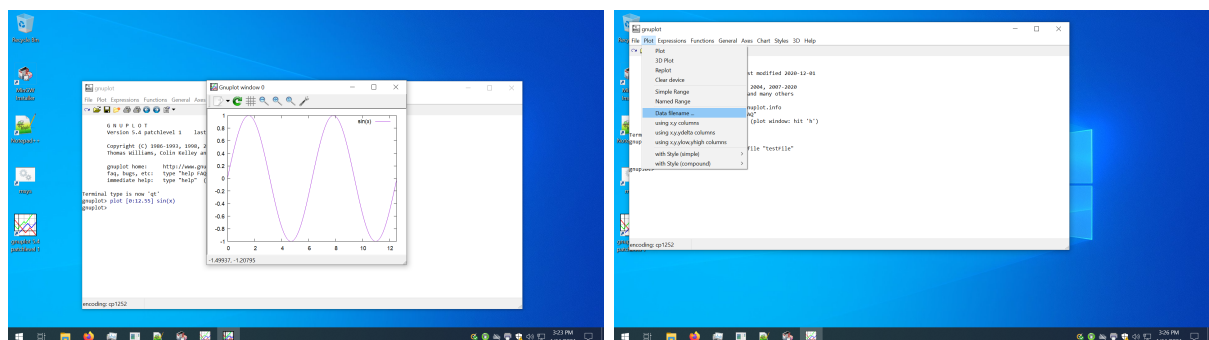


Figure 7: Gnuplot in action.

3.2.3 Installing Notepad++

One would assume that Microsoft Windows provides a usable text editor that can understand and edit the source code. Regrettably, most Office tools are useless for this purpose: instead, we shall install Notepad++. It can be downloaded from <https://notepad-plus-plus.org/downloads/v7.9.2>.

Please do not worry if your version is slightly different: for software under active development it is always best to use the latest version. The installation procedure follows the Windows pattern, Figure 8.

The easiest way to open files with Notepad++ is to drop them onto the Notepad++ icon; when you are finished with them, please close the files to avoid clutter.

This completes the environment you need – enjoy!

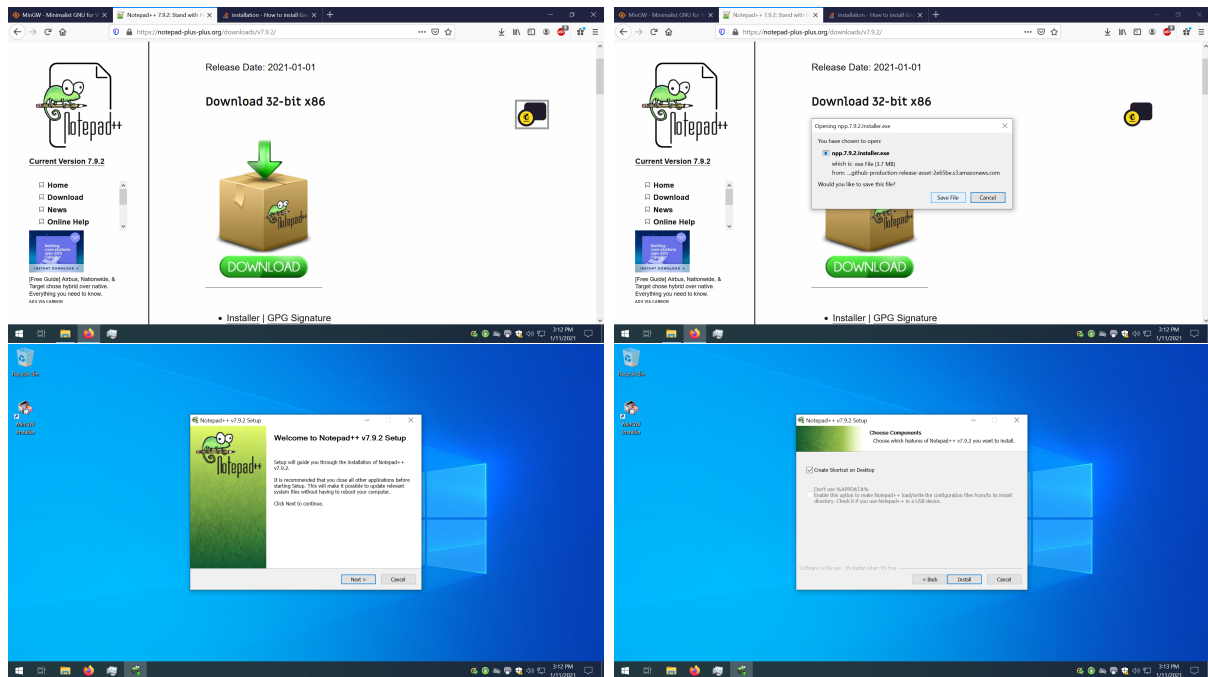


Figure 8: Download and install NodePad++.

3.3 Linux

3.3.1 Checking first if you already have a compiler installed

Many linux-based systems already have a C++ compiler installed by default. If you open a terminal, and can get some interesting looking output by typing either

```
g++ --version
```

or

```
c++ --version
```

into it, then probably you already have a C++ compiler installed. When I do the first of the options above I see the following on my linux machine:

```
pclu--bash$ g++ --version
g++ (GCC) 4.8.5 20150623 (Red Hat 4.8.5-44)
Copyright (C) 2015 Free Software Foundation, Inc.
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

If, however, both of those commands report 'command not found', or some other error, then you probably don't have a C++ compiler and will have to install one. Fortunately, installation is usually very simple on linux.

3.3.2 Installing a C++ compiler on linux

The most popular systems (Ubuntu, Debian, Mint, Elementary OS, ...) use the 'apt' command for package management, and so if you run one of those you will probably be able to install a C++ compiler by opening a terminal, and entering the following command:

```
sudo apt install g++
```

[Note: in some older systems you may need to replace ‘apt’ with ‘apt-get’.]

Other linux systems (such as Redhat or CentOS) likely use either the `yum` command (if they are older) or the `dnf` command (if they are newer). On those systems you will probably be able to install a C++ compiler by opening a terminal, and running either:

```
### On CentOS/RHEL 7/6/5 and systems prior to (approx) 2019
sudo yum update
sudo yum groupinstall "Development Tools"
```

or

```
### For Fedora 28/27/26/25/24/23 and systems after (approx) 2019
sudo dnf update
sudo dnf groupinstall "Development Tools"
```

depending on the age of your system. If in doubt, try the newer command (`dnf`) before the older command (`yum`).

If none of `apt`, `yum` or `dnf` work then consult Google for your particular distribution. If, for example, you are using Arch Linux, then the package manager will be `pacman`. Use the command:

```
sudo pacman -Syu gcc
```

3.3.3 Installing other tools on linux

Once you have a C++ compiler you can code in C++ as explained elsewhere in this handout. However, some of the later parts of the handout recommend that you gain experience in plotting outputs of your C++ programs. The course is happy for you to make such plots in any plotting package you like, but the handout is written assuming you will do it in `gnuplot`. On a linux, to install `gnuplot` you would proceed in a similar method to the above, using whichever of

```
sudo apt install gnuplot
```

or

```
sudo apt-get install gnuplot
```

or

```
sudo yum install gnuplot
```

or

```
sudo dnf install gnuplot
```

or

```
sudo pacman -Syu gnuplot
```

is appropriate for your system. You could do the same for any other packages you might need, but providing the appropriate package name in place of ‘`gnuplot`’. For example. If you don’t know the specific name of a package you want to install, but you are fairly sure it is related to, say, the word ‘`ruby`’ you could list all the potential package matches with whichever of the following commands is appropriate for your system:

```
apt search ruby
apt-cache search ruby
yum search ruby
dfn search ruby
pacman -Ss ruby
```

On my system a command similar to one of those above gives me many lines of output which include the following:

```
rb-xslt @0.9.9 (ruby)
  A Ruby class for processing XSLT

rb-zentest @3.10.0_1 (ruby)
  Automated test scaffolding for Ruby

rb-zip @0.5.7 (ruby, archivers)
  rubyzip is a ruby module for reading and writing zip files

rb19-abstract @1.0.0 (ruby, devel)
  define abstract methods in Ruby

rb19-activerecord @3.0.5 (ruby, databases)
  Ruby ORM

rb19-archive-tar-minitar @0.5.2_1 (ruby)
  Ruby library and utility for POSIX tar(1) archive files

rb19-arel @2.0.9 (ruby, devel, databases)
  Relational Algebra for Ruby
```

The output above might be sufficient to remind me that the name I was searching for was ‘rb-zip’ after which I could type

```
sudo yum install rb-zip
```

to get the package I wanted.

3.4 If programming an Arduino or Arduino clone

Arduinos <https://www.arduino.cc> and similar clones or compatible devices are small micro-controller boards which can be bought online for prices ranging between £3 and £30 depending on the level of functionality you desire in them. An example of an ‘arduino uno’ is shown in Figure 9. These are typically connected to your computer via a USB cable which also powers them, and they can be programmed over that cable. Less commonly they can be powered by batteries and connected to wirelessly either by bluetooth or WiFi.

Strictly speaking, Arduinos are not programmed in C++. They are programmed using proprietary language of their

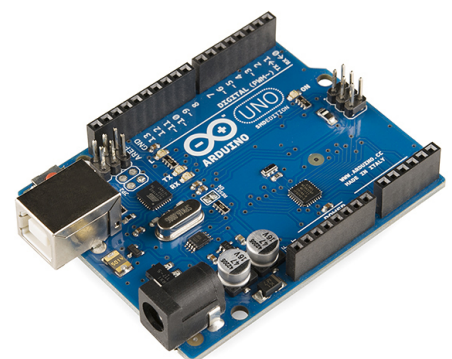


Figure 9: An arduino uno

own. However, that language is sufficiently close to C++ that many students reading this handout might feel confident that, with a bit of help from google, they could adapt the instructions given in this handout to allow them to program on an arduino instead.

The processors an arduinos are not typically powerful or fast, and so they are not useful for doing the kind of scientific computing that requires lots of number-crunching. They are, however, useful for the type of scientific computing that requires you to build a device that can control an experiment, or log some data, or react to sensors, or drive a motor, *etc.* It would therefore be entirely acceptable for a student to work with an arduino (or similar), if he or she wished to do so, since the skill set is valuable and relevant to experimental physics.

Since arduinos are not directly programmed in C++, one does not need to install a C++ compiler to use them. Instead you can either something called the ‘**Arduino IDE**’. This comes in two forms: a web-based one (which the author has not used, but is preferred by people who want all their data to live in the cloud) and stand-alone version of the IDE (called the “**Arduino desktop IDE**”) which is available for you to install on your laptop at the following url: <https://www.arduino.cc/en/software> . The author has prefers to use the latter.

The web contains a very large number of web-based introductions to programming on arduinos. The number and quality of them is such that it is more appropriate to direct people to search for them on the web than to try to reproduce less good versions here. One place you could start is here: <https://www.arduino.cc/en/Tutorial/HomePage> .

4 Editing, compiling and executing a simple program

4.1 A simple C++ program to add two numbers

The following is an example of a simple program (source code) written in the C++ programming language. The program is short but nevertheless complete. The program is designed to read two numbers typed by a user at the keyboard, compute their sum and print the result on the screen.

```
// Program to add two integers typed by user at keyboard
#include <iostream>
using namespace std;

int main()
{
    int a, b, total;

    cout << "Enter integers to be added:" << endl;
    cin >> a >> b;
    total = a + b;
    cout << "The sum is " << total << endl;

    return 0;
}
```

4.2 Overview of program structure and syntax

C++ uses notation that may appear strange to non-programmers. The notation is part of the **syntax** of a programming language, i.e., the formal rules that specify the structure of a legal program. These rules will be explained in more detail later, in sections 6–8. Here we give a quick overview.

Every C++ program consists of a header and a main body and has the following structure.¹⁵

```
// Comment statements, which are ignored by computer but inform reader
#include <header file name>

int main()
{
    declaration of variables;
    statements;

    return 0;
}
```

¹⁵Actually this is not true. Strictly it is only C (not C++) programs which need to have the structure indicated in which all variable declarations must come together in a single group which precedes the statements which use them. In contrast, C++ programs are allowed to mix declarations and statements. This can be a great advantage as the declaration can be much closer to the point of use, making (among other benefits) programs more readable. We will gloss over this distinction for the moment, however, and use C++ as it it were C for a little while.

We will consider each line of the program: for convenience the program is reproduced on the next page with line numbers to allow us to comment on details of the program. You must **not** put line numbers in an actual program.

```
1 // SimpleAdder.cc : Program to add two integers typed by user at keyboard
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7     int a, b, total;
8
9     cout << "Enter integers to be added:" << endl;
10    cin >> a >> b;
11    total = a + b;
12    cout << "The sum is " << total << endl;
13
14    return 0;
15 }
```

Line 1 At the top of the program are comments and instructions, which will not be executed. Lines beginning with `//` indicate that the rest of the line is a **comment**. Comments are inserted by programmers to help people read and understand the program. Here the comment states the purpose of the program. Comments are not executed by the computer. They can in general be placed anywhere in a program.

Line 2 Lines beginning with `#` are instructions to the compiler's *preprocessor*. The **include** instruction says "what follows is a file name, find that file and insert its contents right here". It is used to include the contents of a file of definitions that may be used in the program. Here the file `iostream` contains the definitions of some of the symbols used later in the program (e.g. `cin`, `cout`).

Line 3 This is used to specify that names used in the program (such as `cin` and `cout`) are defined in the standard C and C++ libraries. This is used to avoid problems with other libraries that may also use these names.

Line 5 When this program is executed the instructions will be executed in the order they appear in the main body of the program. The main body is delimited by `main()` and the opening and closing braces (curly brackets). This line also specifies that `main()` will return a value of type integer (`int`) on its completion (see line 14). Every C++ program, irrespective of what it is computing, begins in this way.

Line 6 The opening (left) brace marks the beginning of the main body of the program. The main body consists of instructions, which are **declarations** defining the data or **statements** on how the data should be processed. All C++ declarations and statements must end with a semicolon.

Line 7 This is a **declaration of variables** – like the cast of characters that precedes the script of a play. The words `a`, `b` and `total` are the names of **variables**. A variable is a location in the computer's memory where a value can be stored for use by a program. We can assign and refer to values stored at these locations by simply using the variable's name. The declaration also specifies the variables' **type**. Here the variables `a`, `b` and `total` are declared to be data of type **int**, which means these variables hold integer values. At this stage, the variables' values are undefined.

Line 9 This statement instructs the computer to output the *string* of characters contained between the quotation marks, followed by a new line (`endl`). The location of the output is denoted by `cout`, which in this case will be the terminal screen.

Line 10 This statement tells the computer to read data typed in at the keyboard (standard input), denoted by `cin`. These values are *assigned to* (stored in) variables `a` and `b`.

Line 11 This statement is an **arithmetic expression** assigning the value of the expression `a + b` (the sum of the integer values stored at `a` and `b`) to the variable `total`.

Line 12 Instructs the computer to display the value of the variable `total`.

Line 14 The last instruction of every program is the **return** statement. The return statement with the integer value 0 (zero) is used to indicate to the operating system that the program has terminated successfully.

Line 15 The closing (right) brace marks the end of the main body of the program.

Blank lines (Lines 4, 8 and 13) have been introduced to make the program more readable. They will be ignored by the compiler. **Whitespace** (spaces, tabs and newlines) are also ignored (unless they are part of a string of characters contained between quotation marks). They too can enhance the visual appearance of a program.

Indentation It does not matter where you place statements, either on the same line or on separate lines. A common and accepted *style* is that you indent after each opening brace and move back at each closing brace.

4.3 The development environment and the development cycle

C++ programs go through 3 main phases during development: editing (writing the program), compiling (i.e., translating the program to executable code and detecting syntax errors) and running the program and checking for logical errors (called *debugging*).

1. Edit

The first phase consists of editing a file by typing in the C++ program with a text editor and making corrections if necessary. The program is stored as a text file on the disk, usually with the file extension `.cc` to indicate that it is a C++ program (e.g. `SimpleAdder.cc`).

2. Compile

A compiler translates the C++ program into machine language code (**object code**), which it stores on the disk as a file with the extension `.o` (e.g. `SimpleAdder.o`). A linker then links the object code with standard library routines that the program may use and creates an **executable image**, which is also saved on disk, usually as a file with the file name without any extension (e.g. `SimpleAdder`).

3. **Execute**

The executable is loaded from the disk to memory and the computer's processing unit (Central Processing Unit) executes the program one instruction at a time.

Part IB Computing Course

ASSESSMENT 1

5 Instructions

5.1 Objectives

- Familiarisation with the teaching system and C++ development environment
- Edit, compile and execute a working program
- Become familiar with the compiler's messages

5.2 Getting started

We recommend what can at first appear to be an “old-fashioned” style of computer programming, involving a lot of “typing commands in a terminal window”. There are many other ways of learning to program, some of which are particularly handy if you want to design programs with glossy graphical user interface¹⁶ however we want to expose you the bare bones of lower level programming in this course. This style of working naturally leads into scripting and automation, and will provide you with skills you can use in many places. Though the terminal-based computing model may seem strange to someone only familiar with “point and click” computing, it is the model almost universally adopted for scientific programming requiring a high degree of automation. Most research computing done in The Cavendish or in similar institutions would be done in the manner described. Instructions for how to open/create a terminal were given on page 14 for mac users and on page 21 for windows users.¹⁷ We will need a terminal window so that we have somewhere to type out commands, so

open a terminal now!

[It is sometimes convenient to open more than one terminal at the same time, doing certain kinds of things in one, and other kinds of things in the other.] Once your terminal has opened you can run unix commands it. A command you will use a lot is

`ls`

[NOTE: some students have been mistyping this command, having a first letter as a capital ‘I’ rather than a lower case ‘l’ which is intended. ‘ls’ is short for ‘LIST’ not ‘IST’ !] which should show you the name of the directory you are currently “in”.¹⁸ You might not have any files yet, but you will have some soon. Also useful is

`pwd`

¹⁶You may like to google “integrated development environments”, ‘IDEs, Netbeans, eclipse, ...

¹⁷Linux users will know how to open a terminal.

¹⁸What windows calls a “folder” unix tends to call a “directory”. Two different words, but the concept is identical. Note that slashes separating directories and files are forward slashes in unix `/like/this` whereas they are `\backwards\in\windows`.

which should show you which directory you are currently in. Remember, if none of this is familiar to you, stop this part of the session and go straight to section [D.7](#) of the Appendix to get familiar with unix commands. You can try typing all sorts of things in the terminal with little fear of doing serious damage. **The worst commands you could type** are perhaps

```
rm -rf *
```

which would delete all the files in the current directory and any directories contained within it, or

```
rm *
```

which would delete the files in the current directory only. **Be very careful with the “rm” command. So long as you don’t accidentally type either of those commands or variants thereof**, you should be able to experiment to your heart’s content.

Now something more practical. Let’s try to start a plain text editor - the sort of thing we might use to type in and edit a C++ program. We don’t want to use a word processor like Microsoft Word, since we are not interested in formatting, fonts and pictures, etc. Programs don’t have those. We would also like to use an editor that knows a little bit about computer programs so that it can help us (e.g. by highlighting/colouring the different kinds of programmings in differently to allow us to spot mistakes).

Unfortunately there are a huge number of different code-editing programs, each with different features, new ones are being created all the time, and no one can agree which is the best. Here are some examples. If you don’t already have one of these you will need to download one. See which you like best:

geany I had not heard of this one until Oct 2019, but it looks pretty friendly for new users, is already present on the MCS computers in Cambridge. If you are working on your own computer it can easily be downloaded for free from <https://www.geany.org>. It can also be installed via homebrew or macports if you are on a mac. It was recommended by a number of the 2019 demonstrators. It is plenty of useful features for code editing. If you are new, maybe try this one first.

jedit This was a favourite in 2017, and was found to be pretty friendly for new users. It probably doesn’t have many high level features for experienced programmers, but it starts up quickly, and does have pretty code formatting.

gedit Easy to use, but seems very “heavy”, takes a while to start, not overly specific to programming, but found on many machines. Many of the 2019 demonstrators regard it as a good alternative to geany.

kate

sublime Other alternatives worth trying if you don’t like those above.

emacs This is one of the two big grand-daddy programs – beloved by some programmers and hated by others – it has been around for decades. Hugely configurable (but only in an obscure language called “lisp”), it does at least have menus and knows what a mouse is, but the menus are very quirky. Expect to use the control and alt keys a lot.

vi or vim Older even than **emacs**, but my personal favourite, this program is so old that it doesn’t have menus, doesn’t expect you to have a mouse, doesn’t expect you to even have arrow keys, expects you to remember to type things like “[ESC]:q![ENTER]” just to quit it, or

“`[ESC]%;shCathDoghg[ENTER]`” to do a search-and-replace from “Cat” to “Dog” everywhere in the document being edited. But `vim` is hugely powerful, fast, elegant, and does what you want (once you know how). Beginners should not try this program, except in a nuclear winter, when it is the only editor that will still work.

5.3 Computing exercise – editing, compiling and executing a simple program that adds numbers

The instructions below provide an example of how you might use an editor to type in the simple adder program (`SimpleAdder.cc`) listed on page 27 and then compile and run it.

Then execute it with different numbers at the input and check that it adds up the numbers correctly. What happens if you input two numbers with decimal places?

1. Consider where you want to keep your sourcecode.

You'll be creating many files. To keep things tidy we suggest that you consider using a separate folder or sub-folder for each topic you tackle, though of course you can do whatever you want. A user working on a unix-like system who wanted to work within a folder called `c++` contained within his/her home directory might do something like the following in a terminal:

```
cd $HOME
```

to go to their home directory.¹⁹ If they were using a system they were unfamiliar with, some users might then type

```
pwd
```

to check where their home directory actually was. You might want to try the same. The contents of the user's current directory (in this case the home directory) could be displayed by typing the

```
ls
```

to just 'list' list the file names, [NOTE: some students have been mistyping this command, having a first letter as a capital 'i' rather than a lower case 'L' which is intended. 'ls' is short for 'LIST' not 'IST' !] or

```
ls -l
```

to display a 'longer list' with more information (such as when each file was created, by whom, and what sort of file or directory it is). If the latter command showed that the `c++` folder (in which the user wanted to work) did not already exist then the `c++` folder could be created with

```
mkdir c++
```

After doing so, the new folder could be entered with

```
cd c++
```

¹⁹Just typing '`cd`' or '`cd ~`' would work just as well, but may seem more cryptic to newcomers.

Of course, the same user could also make such a folder without using the terminal at all. Judicious mouse clicks in the correct places could have created such a folder instead. But it is by getting familiar with the command-line based approach, shown above, that you can learn to use the power of automation ('scripting') when you want to be able force the computer to take boring repetitive tasks out of your hands. Persevere!

2. Create a file containing a program in that folder

We now need to create a new file called `SimpleAdder.cc`, in the folder we just entered, containing the listing from page 27. There are numerous ways one can do this, and you could choose whatever way you like best. If you installed a code editor outside of the terminal, you could start it in the same way you would start any other application on your computer, the type in the relevant code, and finally save the file with the appropriate name it in the appropriate folder/directory. If, instead, the editor you want to use is accessible from your terminal/shell, you might prefer to start create and edit the file from the terminal directly from there, e.g. by typing '`geany SimpleAdder.cc`' or '`vim SimpleAdder.cc`'. You should do whatever you find simplest.

It is a good idea to give the file the `.cc` extension because that (among other clues) can inform editors that the file contains a C++ program, and that may help the editor apply formats and colours to your code to make it easier to read.

Avoid putting spaces into the file name as spaces are often used to separate arguments to commands started at the terminal.

3. Compile

To compile your program, go to your terminal and ensure you are still in the right directory (that is, the same folder where the `.cc` file is located). [You could re-use the `pwd`, `cd` and `ls` commands from earlier if you are not sure.] Assuming that `ls` does indeed show you that you are in the same folder as `SimpleAdder.cc` you can then compile that file into an executable by typing

```
g++ -o SimpleAdder SimpleAdder.cc
```

If all is well, an executable called `SimpleAdder` will have been created in your folder. (Type `ls` to check.) If you have made a typing mistake, find the error and correct it using the editor. Save the corrected file and try the compilation again. Repeat until you get an error-message free compilation. If you get stuck, get help from friends or demonstrators.

4. Run

You can now execute the program by typing `./SimpleAdder` in a terminal, assuming you are still in the folder where the program was created. When the program prompt appears, input the two integer numbers to the program by typing in two numbers separated by spaces and followed by a return. The program should respond by printing out the sum of the two numbers.

You can (re)run the program as many times as you wish.

5. Modify

Modify the program in your editor to change its behaviour in some way. Recompile the program. It's a bit of a faff typing the whole command

```
g++ SimpleAdder.cc -o SimpleAdder
```

each time isn't it? You will probably find that you can avoid retyping it all, by simply pressing the **up-arrow key** on your keyboard when you are at your terminal's command prompt. In most terminals each press of the **up-arrow key** shows you a previous command you have typed, which you can then either re-use or alter.

Later on, if you start to create programs with many source files, you may want to investigate the use of 'makefiles' to automate the process of building complex projects. Google something like 'makefiles tutorial' if this interests you.

5.4 Exploring compiler error messages

As you write computer programs in C++, you will repeatedly edit, compile, and run programs. And sometimes the compiler will give you error messages. Often the messages can be quite cryptic. It's a good idea to make some deliberate errors, to get a feel for what your future errors might mean. (Just as trainee doctors learn best by giving known diseases to patients.)

Grab a copy of the program `HelloFor.cc` listed below

```
// HelloFor.cc

#include <iostream>
using namespace std;

int main()
{
    for (int i=1; i<=10; i++) {
        cout << i << " hello world" << endl ;
        if ( i == 7 ) {
            cout << "that was lucky!" << endl ;
        } else {
            cout << endl ;
        }
    }
}
```

You can either copy-paste it from this pdf into your editor, and save it from there, or you can download a copy from [\[this download link\]](#). Make sure to save it to the directory in which you are working, not some other directory.

First compile this program and run it. Remember, once you've got the `.cc` file, you can compile it with the command `g++ -o HelloFor HelloFor.cc`; run it with the command `./HelloFor`

This program contains several C++ features that you haven't learned yet, but don't worry; take a look at the program and the output, and see if you can figure out roughly what's going on in the lines following the word `main`. The program uses the special C++ commands `for` and `if`.

Once you have a feel for what the program is doing, modify the program and explore the error-messages generated by the compiler when you compile programs with deliberate syntax errors.

For example, try omitting a quote character (`"`), or inserting an extra quote character. Try omitting the semicolon (`;`) at the end of a line. Try omitting or adding a left-brace (`{`) or right-brace (`}`). How many different types of error can you get the compiler to produce? Make a list.

When the compiler gives error messages, it tells you the line-number where it finds an error. Is this line-number always the same as the line number where you introduced the typing error?

Assuming you make a single-character typing error, how big can you make the distance between the actual line-number of the error and the line-number where the compiler reports the error?

This knowledge of deliberately inflicted diseases and their symptoms should help you diagnose future bugs.

5.5 Compiler warnings

What happens if you replace the equality test (`i==7`) by (`i=7`)? Make the change and see. Then see if you can guess why the output is different. This is learning to program!

[By the way: if you want to stop a running program, hit ctrl-C.]

Now try compiling the program with the incorrect equals sign using this command:

```
g++ -Wall HelloFor.cc -o HelloFor
```

The flag `-Wall` means ‘Warnings (all)’. What difference does adding the flag `-Wall` make?

Suggested lesson: always compile with Warnings switched on.

5.6 Practice

The example `SimpleAdder.cc` showed you how to make a simple interactive program, which prompts the user for an integer using the special command `cin`.

Modify `HelloFor.cc` using code from `SimpleAdder.cc` to make a program that first asks the user to enter a lucky number (which plays the role of 7), then prints out the numbers 1 to 10, highlighting the lucky number in some way when it is printed. Modify the program so that the number of loops (currently 10) is also chosen interactively by the user.

5.7 Assessment

This course proceeds by self-assessment. Here is how you might self-assess in week one. It’s fine to work with others, as part of the learning process, but this bit, the assessed bit, and the task you set yourself should apply to you alone.

Set yourself an **additional programming task** along the lines of the preceding tasks, **and solve it!** Be imaginative! The [course web page](#) has links to many of the goals which students have set themselves in the past when taking this course. See the section called ‘ideas for assessed work’. Remember that your task should be one which should take you two hours, or thereabouts.

[If this instruction sounds too vague, go back and read the description of self assessment in Section [2.1](#) of this document to remind yourself how to enter into the spirit of this course.]

When you are done, create a file called `README` containing text something like the following:

```
I set myself the following additional goal (devised by me):
```

```
I decided to write a program to [BLAH BLAH BLAH] and extended
modified it to [ETC ETC ETC] until it could [BLAH BLAH BLAH].
```

```
My solution was in a file called _____.cc
```


If you are not confident that you have mastered the material, seek help right away. Seek help from fellow students, from demonstrators, or from the course website (see frontispiece). If you are still stuck, email the lecturer. Otherwise, pat yourself on the back, and submit your work as follows:

5.8 Instructions for submitting your self-assessed work

Submission of work will be by what is (hopefully) a simple web-based form. The link to the upload form may be found under “Submitting assessed work” near the top of the [course homepage](#), whose url is on the cover of this document. [N.B. : The form is slightly modified this year, so it is not impossible that teething problems will be found in the first few days. Report any problems with the submission process to the lecturer as soon as possible.] The page should require you to supply:

- Your name. Hopefully you know what this is!
- Your email address. This is only used for the purposes of mailing you the upload confirmation.
- The week for which you are submitting work, i.e. ASSESSMENT 1, ASSESSMENT 2, *etc.*
- A title for the task which you set yourself. This might be: “My toasted teacake timer” or “A program to calculate prime numbers” or “My first foray into C++” or whatever describes your program in a few words.
- A brief description of the task you set yourself. Into this box you should paste the contents of the README file you created earlier. About 100 words should be sufficient, here. 1000 words would be excessive! As an example you might paste in the text:

I set myself the following additional goal (devised by me):

I decided to write a program to print out the complete works of Charles Dickens, and then modified it by removing words repeatedly until it could only print Hello World.

My solution was in a file called Dickens.cc and in addition I created a file called M00.TXT containing the word "cow" of which I am moderately proud.

- Finally you will need to paste in the sourcecode of your program(s) that you wrote to solve the problem(s) which you set for yourself. You might ask why this stage asks you to *paste* sourcecode into a text-box, rather than getting you to upload the sourcecode files themselves directly. The reason is that many people in previous years have managed to bungle their file uploads by sending the executables they compiled *from* their sourcecode rather than by sending the sourcecode itself!²⁰ Executables cannot be read by examiners and therefore cannot be marked. Sourcecode can be read by humans, and so can be marked. You get marks in this course for examinable submissions. If your program is split across a number

²⁰This despite numerous reminders not to do this.

of files, please paste them one after another into the sourcecode box on the web submission form, separating each with a comment on a line of its own of the form:

```
// Here begins file _____.cc
```

and so on. While it is expected that the majority of the sourcecode submitted will be C++, there may be circumstances later in term when you find it beneficial to submit one or two additional short shell scripts, or scripts containing “gnuplot” commands, etc. This is fine, so long as the content is “plain text”, and is human readable. Just paste them all into the box with the separators as indicated above. Do not attempt to submit pictures or any form of binary or executable! Here you might imagine pasting in the following:

```
// Here begins file Dickens.cc
#include <iostream>
int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

```
// Here begins file MOO.TXT
cow
```

If all is in order, after you click “Submit” **you should be presented with a message resembling that shown on the next page**. You should also receive the confirmation via email shortly after. If all that happens, your submission was a success and you can continue to the next task. If you suspect any problem with submission, contact the lecturer.

Thank you for submitting work for ASSESSMENT 1.

If you are on a shared computer, close this tab and restart your browser after reading the confirmation below, to ensure nobody else can submit work pretending to be you.

An email has been sent to you confirming your upload. If you don't receive an email within 24 hours it is likely that your email address has been entered incorrectly. In this case please send an email to `cgl20@cam.ac.uk`. The email should contain the following text:

To: `cgl20@cam.ac.uk,lester@hep.phy.cam.ac.uk`
Subject: Confirmation of IB NST Physics B upload
for Computation course, SESSION 1
Dear Christopher Lester,

This is an automatically generated mail.

Thank you for submitting work for SESSION 1.
Your work was submitted at 14:51 and 01 seconds on 18/01/2011.
Your submission was entitled 'Hello World', and was described as follows:

I set myself the following additional goal (devised by me):

I decided to write a program to print out the complete works of Charles Dickens, and then modified it by removing words repeatedly until it could only print Hello World.
My solution was in a file called `Dickens.cc` and in addition I created a file called `MOO.TXT` containing the word "cow" of which I am moderately proud.

You submitted the following code:

```
// Here begins file Dickens.cc
#include <iostream>
int main() {
    std::cout << "Hello World" << std::endl;
    return 0;
}
```

```
// Here begins file MOO.TXT
cow
```

If you finish the first assessment with time to spare, why not make a start on the second one?

END OF ASSESSMENT 1

6 Variables and constants

Programs need a way to store the data they use. Variables and constants offer various ways to represent and manipulate data. Constants, as the name suggests, have fixed values. Variables, on the other hand, hold values that can be assigned and changed as the program executes.

6.1 Variable types

Every variable and constant has an associated **type**, which defines the set of values that can be legally stored in it. Variables can be conveniently divided into integer, floating point, character and boolean types for representing integer (whole) numbers, floating point numbers (real numbers with a decimal point), the ASCII character set (for example ‘a’, ‘b’, ‘A’) and the boolean set (**true** or **false**) respectively.

More complicated types of variable can be defined by a programmer, but for the moment, we will deal with just the simple C++ types. These are listed below:

int	to store a positive or negative integer (whole) number
float	to store a real (floating point) number
bool	to store the logical values true or false
char	to store one of 256 character (text) values

6.2 Declaration of a variable

A variable is introduced into a program by a declaration, which states the variable’s **type** (i.e., **int**, **float**, **bool**, or **char**) and its name, which you are free to choose. A **declaration** must take the form: “*type variableName*;” as is seen in these four examples:

```
int      count;
float    length;
char     firstInitial;
bool     switched_on;
```

or the form: “*type variableName1, variableName2, ..., variableNameN*;” as is seen in these two examples:

```
float     base, height, areaCircle;
int       myAge, number_throws;
```

The variable name can be any sequence of characters consisting of letters, digits and underscores that do not begin with a digit. It must not be a special keyword of the C++ language and cannot contain spaces. C++ is case-sensitive: uppercase and lowercase letters are considered to be different. Good variable names tell you how the variable is used and help you understand the flow of the program. If you want to use a name made of two or more words, you may indicate the word boundaries by the underscore symbol (`_`) or by using an uppercase letter for the beginning of words.

6.3 Storage of variables in computer memory

When you run your program it is loaded into computer memory (RAM) from the disk file. A variable is in fact a location in the computer's memory in which a value can be stored and later retrieved. The variable's name is merely a label for that location – a *memory address*. It may help to think of variables as named boxes into which values can be stored and retrieved.

The amount of memory required for the variables depends on their type. This can vary between machines and systems but is usually one byte (8 bits) for a **char** variable, four bytes (32 bits) for an **int** and four bytes for a **float**. This imposes limits on the range of numbers assigned to each variable. Integer numbers must have values in the range -2147483648 to 2147483647 (i.e., $\pm 2^{31}$). Floats must be real numbers with magnitudes in the range 5.9×10^{-39} to 3.4×10^{38} (i.e., 2^{-127} to 2^{128}). They are usually stored using 1 bit for the sign (s), 8 bits for the exponent (E) and 23 bits for the mantissa (m) such that the number is equal to $s \times m \times 2^E$. The number of bits in the mantissa determines the accuracy of addition and subtraction. If two numbers are added and the smaller is smaller than $2^{-23} \approx 10^{-7}$ times the larger then the outcome of the addition will simply be the larger number. This inaccuracy of addition can cause problems for numerical methods such as dynamical simulations.

If an application requires very small or large numbers beyond these ranges, or greater precision, C++ offers an additional data types for floating point numbers: **double**. Variables of type **double** require double the amount of memory for storage but are useful when computing values to a high precision. Some programmers follow the rule ‘always use doubles, not floats’.

7 Assignment of variables

7.1 Assignment statements

It is essential that every variable in a program is given a value explicitly before any attempt is made to use it. It is also important that the value assigned is of the correct type.

The most common form of statement in a program uses the **assignment operator**, =, and either an **expression** or a **constant** to assign a value to a variable:

```
variable = expression;  
variable = constant;
```

The symbol of the assignment operator looks like the mathematical equality operator **but** in C++ its meaning is different. The assignment statement indicates that the value given by the expression on the right hand side of the **assignment operator** (symbol =) must be stored in the variable named on the left hand side. The assignment operator should be read as “becomes equal to” and means that the **variable** on the left hand side has its value changed to the value of the **expression** on the right hand side. (In other programming languages the assignment operator is represented by := or <-.) For the assignment to work successfully, the type of the variable on the left hand side should be the same as the type returned by the expression.

The statement in line 11 of the simple adder program is an example of an assignment statement involving an **arithmetic expression**.

```
total = a + b;
```

It takes the values of **a** and **b**, sums them together and assigns the result to the variable **total**. As discussed above, variables can be thought of as named boxes into which values can be stored.

Whenever the name of a box (i.e., a variable) appears in an expression, it represents the value currently stored in that box. When an assignment statement is executed, a new value is dropped into the box, replacing the old one. Thus, line 10 of the program means “get the value stored in the box named `a`, add it to the value stored in the box named `b` and store the result in the box named `total`.”

The assignment statement:

```
total = total + 5;
```

is thus a valid statement since the new value of `total` becomes the old value of `total` with 5 added to it. Remember the assignment operator (`=`) is **not** the same as the equality operator in mathematics (represented in C++ by the operator `==`).

7.2 Arithmetic expressions

Expressions can be constructed out of variables, constants, operators and brackets. The commonly used mathematical or **arithmetic operators** include:

operator	operation
+	addition
-	subtraction
*	multiplication
/	division
%	modulus (modulo division)

The definitions of the first four operators are as expected. The modulo division (modulus) operation with an integer is the remainder after division, e.g. $13 \bmod 4$ (`13%4`) gives the result 1.

Although addition, subtraction and multiplication are the same for both integers and reals (floating point numbers), division is different. If you write (see later for declaration and initialisation of variables on the same line):

```
float a=13.0, b=4.0, result;  
result = a/b;
```

then a real division is performed and 3.25 is assigned to `result`. A different result would have been obtained if the variables had been defined as integers:

```
int i=13,j=4, result;  
result = i/j;
```

here `result` is assigned the integer value 3.

The remainder after integer division can be determined by the modulo division (modulus) operator, `%`. For example, the value of `i%j` would be 1. Be careful when using division and modulo division with negative integers. What is $-13/4$? What is $-13\%4$? It's easy to get tripped up because there are two possible conventions, and each programming language makes its own choice. In most languages the division and remainder operators will be *consistent* in the sense that

for all `a` and `b`, $(a/b)*b + a\%b = a$.

but this still permits two conventions. In C++, $-13/4$ is -3 and $-13\%4$ is -1 .

7.3 Precedence and nesting parentheses

The use of parentheses (brackets) is advisable to ensure the correct evaluation of complex expressions. Here are some examples:

```
4 + 2 * 3   equals 10
(4+2) * 3   equals 18
-3 * 4      equals -12
4 * -3      equals -12 (but it's safer to use parentheses: 4 * (-3))
0.5(a+b)    illegal (missing multiplication operator)
(a+b) / 2    equals the average value of a and b if they are of type float
```

The order of execution of mathematical operations is governed by rules of precedence. These are similar to those of algebraic expressions. Parentheses are always evaluated first, followed by multiplication, division and modulus operations. Addition and subtraction are last. The best thing, however, is to use parentheses (brackets) instead of trying to remember the rules.

7.4 Initialisation of variables

Variables can be assigned values when they are first defined (called initialisation):

```
type    variable = literal constant;
float    ratio = 0.8660254;
int      myAge = 19;
char     answer = 'y';
bool     raining = false;
```

The terms on the right hand side are called constants. Note the ASCII character set is represented by type `char`. Each character constant is specified by enclosing it between single quotes (to distinguish it from a variable name). Each `char` variable can only be assigned a single character. These are stored as numeric codes. (The initialisation of words and character strings will be discussed later in the section on advanced topics.)

You may assign the value of an expression too –

```
type    variable = expression;
float    product = factor1*factor2;
```

– as long as the variables in the expression on the right hand side have already been declared and had values assigned to them. When declaring and initialising variables in the middle of a program, the variable exists (i.e., memory is assigned to store values of the variable) up to the first right brace that is encountered, excluding any intermediate nested braces, `{}`.

7.5 Expressions with mixed variable types

At a low level, a computer is not able to perform an arithmetic operation on two different data types of data. In general, only variables and constants of the *same* type should be combined in an expression. The compiler has strict *type checking* rules to check for this.

In cases where mixed numeric types appear in an expression, the compiler replaces all variables with copies of the highest precision type. It *promotes* them so that in an expression with integers and float variables, the integer is automatically converted to the equivalent floating point number for the purpose of the calculation only. The value of the integer is not changed in memory. Hence, the following is legal:

```
int i=13;
float x=1.5;
x = (x * i) + 23;
```

Here the values of `i` and `23` are automatically converted to floating point numbers and the result is assigned to the float variable `x`. However the expression `i/j` here:

```
int i=13,j=4;
float result;
result = i/j;
```

is evaluated by integer division and therefore produces the incorrect assignment of `3.0` for the value of `result`. You should try to avoid expressions of this type but occasionally you will need to compute a fraction from integer numbers. In these cases the compiler needs to be told specifically to convert all the variables on the right-hand side of the assignment operator to type `float` or `double`. This is done by **casting**.

In the C++ language this is done by using the construction:

```
static_cast<type> expression
```

For example:

```
int count=13, N=400;
float fraction;
fraction = static_cast<float>(count)/N;
```

converts (casts) the value stored in the integer variable `count` into a floating point number, `13.0`. The integer `N` is then promoted into a floating point number to give a floating point result.

7.6 Declaration and initialisation of symbolic constants

Like variables, **symbolic** constants have types and names. A constant is declared and initialised in a similar way to variables **but** with a specific instruction to the compiler that the value cannot be changed by the program. The values of constants must always be assigned when they are created.

```
const   type      constant-name = literal constant;
const   float     Pi = 3.14159265;
const   int       MAX = 10000;
```

The use of constants helps programmers avoid inadvertent alterations of information that should never be changed. The use of appropriate constant names instead of using the numbers also helps to make programs more readable.

b

8 Simple input and output

C++ does not, as part of the language, define how data is written to a screen, nor how data is read into a program. This is usually done by “special variables” (*objects*) called **input and output streams**, `cin` and `cout`, and the **insertion** and **extraction operators**. These are defined in the **header file** called `iostream`. To be able to use these *objects* and operators you must include the file `iostream` at the top of your program by including the following line of code before the main body in your program.


```
#include <iostream>
```

followed by

```
using namespace std;
```

[Note: In principle you can omit the “`using namespace std;`” line, but then you need to remember to always type “`std::cout`” and “`std::cin`” and “`std::endl`” further down this page, where you would otherwise been able to get away with typing “`cout`” and “`cin`” and “`endl`” in the manner the handout shows. For the moment you can think of the “`using namespace std;`” line as saying something like

“I’m lazy! Please put ‘`std::`’ in front of anything that needs it!”²¹

You will see some programs written with one style, and others with the other.]

8.1 Printing to the screen using output stream

A statement to print the value of a variable or a **string of characters** (set of characters enclosed by double quotes) to the screen begins with `cout`, followed by the **insertion operator**, (`<<`), which is created by typing the “less than” character (`<`) twice. The data to be printed follows the insertion operator.

```
cout << "text to be printed ";
cout << variable;
cout << endl;
```

The symbol `endl` is called a **stream manipulator** and moves the cursor to a new line. It is an abbreviation for *end of line*.

Strings of characters and the values of variables can be printed on the same line by the repeated use of the insertion operator. For example

```
int total = 12;
cout << "The sum is " << total << endl;
```

prints out

```
The sum is 12
```

and then moves the cursor to a new line.

8.2 Input of data from the keyboard using input stream

The input stream *object* `cin` and the **extraction operator**, (`>>`), are used for reading data from the keyboard and assigning it to variables.

```
cin >> variable ;
cin >> variable1 >> variable2 ;
```

Data typed at the keyboard followed by the *return* or *enter* key is assigned to the variable. The value of more than one variable can be entered by typing their values on the same line, separated by spaces and followed by a return, or on separate lines.

²¹More formally, programmers say “`std` is the namespace in which “`cout`” and “`cin`” and “`endl`” are declared. Google “c++ namespace” to find out more about namespaces.

9 Control Statements

In most of the programs presented above, the statements have been sequential, executed in the order they appear in the main program.

In many programs the values of variables need to be tested, and depending on the result, different statements need to be executed. This facility can be used to **select** among alternative courses of action. It can also be used to build **loops** for the **repetition** of basic actions.

9.1 Boolean expressions and relational operators

In C++ the testing of *conditions* is done using **boolean expressions**, which yield `bool` values that are either **true** or **false**. The simplest and most common way to construct such an expression is to use the so-called **relational operators**.

<code>x == y</code>	true if x is equal to y
<code>x != y</code>	true if x is not equal to y
<code>x > y</code>	true if x is greater than y
<code>x >= y</code>	true if x is greater than or equal to y
<code>x < y</code>	true if x is less than y
<code>x <= y</code>	true if x is less than or equal to y

Be careful to avoid mixed-type comparisons. If `x` is a floating point number and `y` is an integer the equality tests will **not** work as expected.

9.2 Compound boolean expressions using logical operators

If you need to test more than one relational expression at a time, it is possible to combine the relational expressions using the **logical operators**.

Operator	C++ symbol	Example
AND	<code>&&</code> , and	<code>expression1 && expression2</code>
OR	<code> </code> , or	<code>expression1 expression2</code>
NOT	<code>!</code>	<code>!expression</code>

The meaning of these will be illustrated in examples below.

9.3 The IF selection control statement

The simplest and most common selection structure is the **if** statement, which is written in a statement of the form:

```
if(boolean-expression) statement;
```

The **if** statement tests for a particular condition (expressed as a boolean expression) and only executes the following statement(s) if the condition is true. An example follows of a fragment of a program that tests if the denominator is not zero before attempting to calculate **fraction**.

```
if(total != 0)
    fraction = counter/total;
```

If the value of `total` is 0, the boolean expression above is false and the statement assigning the value of `fraction` is ignored.

If a sequence of statements is to be executed, this can be done by making a **compound statement** or **block** by enclosing the group of statements in braces.

```
if(boolean-expression)
{
    statements;
}
```

An example of this is:

```
if(total != 0)
{
    fraction = counter/total;
    cout << "Proportion = " << fraction << endl;
}
```

9.4 The IF/ELSE selection control statement

Often it is desirable for a program to take one branch if the condition is true and another if it is false. This can be done by using an **if/else** selection statement:

```
if(boolean-expression)
    statement-1;
else
    statement-2;
```

Again, if a sequence of statements is to be executed, this is done by making a compound statement by using braces to enclose the sequence:

```
if(boolean-expression)
{
    statements;
}
else
{
    statements;
}
```

An example occurs in the following fragment of a program to calculate the roots of a quadratic equation.

```

// testing for real solutions to a quadratic
d = b*b - 4*a*c;
if(d >= 0.0)
{
    // real solutions
    root1 = (-b + sqrt(d)) / (2.0*a);
    root2 = (-b - sqrt(d)) / (2.0*a);
    real_roots = true;
}
else
{
    // complex solutions
    real = -b / (2.0*a);
    imaginary = sqrt(-d) / (2.0*a);
    real_roots = false;
}

```

If the boolean condition is **true**, i.e. ($d \geq 0$), the program calculates the roots of the quadratic as two real numbers. If the boolean condition is **false**, then a different sequence of statements is executed to calculate the real and imaginary parts of the complex roots. Note that the variable **real_roots** is of type **bool**. It is assigned the value **true** in one of the branches and **false** in the other.

9.5 ELSE IF multiple selection statement

Occasionally a decision has to be made on the value of a variable that has more than two possibilities. This can be done by placing **if** statements within other **if-else** constructions. This is commonly known as *nesting* and a different style of indentation is used to make the multiple-selection functionality much clearer. This is given below:

```

if(boolean-expression-1)
    statement-1;
else if(boolean-expression-2)
    statement-2;
:
else
    statement-N;

```

For compound statement blocks, braces must be used. Here follows a dull example of ELSE IF multiple selection in use:

```

// Program to assess the user's for suitability as a major college donor:

// Find out how many houses are owned:
int howManyHousesSheOwns;
cout << "How many houses do you own?" << endl;
cin >> howManyHousesSheOwns;

// Give feedback
if (howManyHousesSheOwns<=0)
{
    cout << "And have you come far today?" << endl;
}
else if (howManyHousesSheOwns<10)
{
    cout << "Student hardship funds make a real difference to
            many students lives!" << endl;
}
else if (howManyHousesSheOwns<100)
{
    cout << "Have I told you about the business park the college
            would like to build in the Cotswolds?" << endl;
}
else
{
    // I'm speechless!
}

```

Note that since whitespace outside of strings is irrelevant in C++, you also see formatting like the following, which is entirely equivalent and preferred by many:

```

// Program to assess the user's for suitability as a major college donor:

// Find out how many houses are owned:
int howManyHousesSheOwns;
cout << "How many houses do you own?" << endl;
cin >> howManyHousesSheOwns;

// Give feedback
if (howManyHousesSheOwns<=0) {
    cout << "And have you come far today?" << endl;
} else if (howManyHousesSheOwns<10) {
    cout << "Student hardship funds make a real difference to
            many students lives!" << endl;
} else if (howManyHousesSheOwns<100) {
    cout << "Have I told you about the business park the college
            would like to build in the Cotswolds?" << endl;
} else {
    // I'm speechless!
}

```

9.6 SWITCH multiple selection statement

Instead of using multiple `if/else` statements C++ also provides a special control structure, `switch`.

For an integer variable `x`, see footnote²², the `switch(x)` statement tests whether `x` is equal to the constant values `x1`, `x2`, `x3`, etc. and takes appropriate action. The `default` option is the action to be taken if the variable does not have any of the values listed.

```
switch(x)
{
    case x1:
        statements1;
        break;

    case x2:
        statements2;
        break;

    case x3:
        statements3;
        break;

    default:
        statements4;
        break;
}
```

The `break` statement causes the program to proceed to the first statement after the `switch` structure. Note that the `switch` control structure is different to the others in that braces are not required around multiple statements. If you forget to put in one of the `break` statements, then execution can continue from one case statement into another. Sometimes you intend this to happen, but more often than not it is not intended and a source of many bugs. Be careful to include all the `break` statements unless you are very sure there are some you do not need.

The following example uses the `switch` statement to produce a simple calculator that branches depending on the value of the operator being typed in. The operator is read and stored as a character value (`char`). The values of `char` variables are specified by enclosing them between single quotes. The program is terminated (`return -1`) if two numbers are not input or the simple arithmetic operator is not legal. The return value of `-1` instead of `0` signals that an error took place.

²²The variable `x` **must** be either an integer type, or one that can evaluate to an integer. In practice this means you can successfully use `switch` on `ints` and `chars`, since single characters are represented by their so-called ASCII code and thus are on an equal footing as integers – indeed are the most basic of the integer types. Do not attempt to use `switch` on a variable which is a `float` or a `double` or a string. It almost certainly won't work, and even if it appears to work, it could fail at some later point.

```

// CalculatorSwitch.cc
// Simple arithmetic calculator using switch() selection.

#include <iostream>
using namespace std;

int main()
{
    float a, b, result;
    char operation;

    // Get numbers and mathematical operator from user input
    cin >> a >> operation >> b;

    // Character constants are enclosed in single quotes
    switch(operation)
    {
        case '+':
            result = a + b;
            break;

        case '-':
            result = a - b;
            break;

        case '*':
            result = a * b;
            break;

        case '/':
            result = a / b;
            break;

        default:
            cout << "Invalid operation. Program terminated." << endl;
            return -1;
    }

    // Output result
    cout << result << endl;
    return 0;
}

```

9.7 The WHILE repetition control statement

Repetition control statements allow the programmer to specify actions that are to be repeated while some condition is **true**. In the **while** repetition control structure:

```
while(boolean-expression)
{
    statements;
}
```

the boolean expression (or ‘condition’) is tested and the statements enclosed by the braces are executed repeatedly while the condition given by the boolean expression is true. The loop terminates as soon as the boolean expression is evaluated and tests false. Execution will then continue on the first line after the closing brace. You can have as many statements as you like inside the loop, including one or zero.

Note that if the boolean expression is initially false the statements are not executed. In the following example the boolean condition becomes false when the first negative number is input at the keyboard. The sum is then printed.

```
// AddWhilePositive.cc
// Computes the sum of numbers input at the keyboard.
// The input is terminated when input number is negative.

#include <iostream>
using namespace std;

int main()
{
    float number, total=0.0;

    cout << "Input numbers to be added: " << endl;
    cin >> number;

    // Stay in loop while input number is positive
    while(number >= 0.0)
    {
        total = total + number;
        cin >> number;
    }

    // Output sum of numbers
    cout << total << endl;
    return 0;
}
```


9.8 Increment and decrement operators

Increasing and decreasing the value of an integer variable is a commonly used method for counting the number of times a loop is executed. C++ provides a special operator `++` to increase the value of a variable by 1. The following are equivalent ways of *incrementing* a counter variable by 1.

```
count = count + 1;
count++;
```

The operator `--` decreases the value of a variable by 1. The following are both *decrementing* the counter variable by 1.

```
count = count - 1;
count--;
```

C++ also provides the operators `+=` and `-=`, which increase and decrease the value of a variable by an expression. For example:

```
count -= 5;
total += x + y;
```

the first line decreases `count` by 5; the second increases `total` by the value of the expression `x+y`.

9.9 The FOR repetition control statement

Often in programs we know how many times we will need to repeat a loop. A `while` loop could be used for this purpose by setting up a starting condition; checking that a condition is true; and then incrementing or decrementing a counter within the body of the loop. For example we can adapt the `while` loop in `AddWhilePositive.cc` so that it executes the loop `N` times and hence sums the `N` numbers typed in at the keyboard.

```
i=0;           // initialise counter
while(i<N)     // test whether counter is still less than N
{
    cin >> number;
    total = total + number;
    i++;       // increment counter
}
```

The initialisation, test and increment operations of the `while` loop are so common when executing loops a fixed number of times that C++ provides a concise representation – the `for` repetition control statement:

```
for(int i=0; i<N; i++)
{
    cin >> number;
    total = total + number;
}
```

This control structure has exactly the same effect as the `while` loop listed above.

The following is a simple example of a `for` loop with an increment statement using the increment operator to calculate the sample mean and variance of N numbers. The `for` loop is executed N times.

```
//ForStatistics.cc
//Computes the sample mean and variance of N numbers input at the keyboard.
//N is specified by the user but must be 10 or fewer in this example.

#include <iostream>
using namespace std;

int main()
{
    int    N=0;
    float  number, sum=0.0, sumSquares=0.0;
    float  mean, variance;

    // Wait until the user inputs a number in correct range (1-10)
    // Stay in loop if input is outside range
    while(N<1 || N>10)
    {
        cout << "Number of entries (1-10) = ";
        cin  >>  N;
    }

    // Execute loop N times. Start with i=1; increment i at end of each loop;
    //                               exit loop when i = N+1.
    for(int i=1; i<=N; i++ )
    {
        cout << "Input item number " << i << " = ";
        cin >> number;
        sum = sum + number;
        sumSquares = sumSquares + number*number;
    }

    mean = sum/float(N);
    variance = sumSquares/float(N) - mean*mean;

    cout << "The mean of the data is " << mean <<
          " and the variance is " << variance << endl;
    return 0;
}
```

A more general form of the `for` repetition control statement is given by:

```
for(statement1; boolean-expression; statement3)
{
    statement2;
}
```

The first statement (*statement1*) is executed and then the boolean expression (condition) is tested. If it is true *statement2* (which may be more than one statement) and *statement3* are executed. The boolean expression (condition) is then tested again and if true, *statement2* and *statement3* are repeatedly executed until the condition given by the boolean expression becomes false.

On page 53, we gave an example where we executed a loop *N* times using

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

In the `ForStatistics.cc` example, we used this form to loop *N* times.

```
for(int i=1; i<=N; i++) {    ...    }
```

Notice the two differences between these two commands.

The first example starts out with *i*=0 and continues looping as long as *i*<*N*, which means that the values that *i* takes on in the loop are 0, 1, ...*N*-1. This is a very common style in C++ – to start at zero, and go up to *N* – 1.

The second example starts out with *i*=1 and continues looping as long as *i*<=*N*, which means that the values that *i* takes on in the loop are 1, 2, ...*N*. After we exit the loop, *i* will be equal to *N*+1.

Beware! One of the most common programming errors is to mix up the ‘zero-offset’ convention and the ‘one-offset’ convention.

9.10 The DO...WHILE repetition control statement

Both the **while** and the **for** repetition methods check the condition *before* every iteration. This means that it’s possible that the body of the loop may never be executed – if the condition is already not true at the start. Sometimes this is exactly what you want. For example, if you want to find how many times a positive integer *n* is exactly divisible by 2, you could use:

```
i=0;    while((n%2)==0) { n /= 2; i ++ ; }
```

or

```
i=0;    for(;;(n%2)==0;) { n /= 2; i ++ ; }
```

On the other hand, sometimes you want the body of the loop always to be executed at least once, and you want the condition to be tested only before subsequent iterations. In such cases, you can use the construction

```
do
{
    body statements;
} while (boolean-expression) ;
```

Notice the final semicolon after the condition, in contrast to **for** and **while**.

A good example of this is a ‘please enter a number’ dialogue, where the question must be asked once, and perhaps repeated *until* the user supplies a valid answer.

```
do {
    cout << "Enter a number between 1 and 10:" ;
    cin >> N ;
} while( (N<1) || (N>10) ) ;
```

In English, this says ‘ask for a number *N*, repeatedly, as long as the number *N* is *outside* the range’. Remember, the two-pipes symbol ‘||’ is the logical operator OR.

Part IB Computing Course

ASSESSMENT 2

10 Instructions

10.1 Objectives

By now you should feel able to:

- Declare and define variables and constants
- Assign values to variables and manipulate them in arithmetic expressions
- Write the value of variables to the screen
- Read in the value of variables from the keyboard
- Write simple programs that use loops.

You might perhaps now decide to use those skills to write and execute the program described in the following computing exercise (section 10.2). But remember, the course is self-assessed, and so you should depart from the following suggestion if a harder (or easier) task would be more appropriate for you. Details and hints on the writing and *debugging* of the programs are given in section 10.3 and beyond.

10.2 Computing Exercises (suggestions)

Modify the `SimpleAdder.cc` program from earlier so that it converts a physical quantity in one set of units to another set of units and displays the result as a floating point number. For example, read in a time in years, and display that time in seconds; or read in a power in kWh per day and display it in W.

Write a program that uses a `for` loop to sum the first N odd numbers, where N is a number supplied by the user. When it has computed the sum, your program should check that

$$\sum_{i=1}^N (2i - 1) = N^2$$

Write a program that uses a loop to print out the sequence of floating-point numbers:

$$1, 1 + \frac{1}{1}, 1 + \frac{1}{1 + \frac{1}{1}}, 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}}, 1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1 + \frac{1}{1}}}}, \dots$$

(Print out the first 20 terms.) If you would like to see more digits of precision than the default (6), see the output formatting instructions on page 99.

Modify your program so that it reads in a number x from the user, then prints out the sequence of floating-point numbers:

$$x, x + \frac{1}{x}, x + \frac{1}{x + \frac{1}{x}}, x + \frac{1}{x + \frac{1}{x + \frac{1}{x}}}, x + \frac{1}{x + \frac{1}{x + \frac{1}{x + \frac{1}{x}}}}, \dots$$

What does this sequence

$$2, 2 + \frac{1}{2}, 2 + \frac{1}{2 + \frac{1}{2}}, 2 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2}}}, 2 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2 + \frac{1}{2}}}}, \dots$$

tend to?

Write a program that uses a loop to print out the sequence of floating-point numbers:

$$\sqrt{1}, \sqrt{1 + \sqrt{1}}, \sqrt{1 + \sqrt{1 + \sqrt{1}}}, \sqrt{1 + \sqrt{1 + \sqrt{1 + \sqrt{1}}}}, \dots$$

What does this sequence tend to? The square root function in C++ is called `sqrt(...)`. To tell the C++ compiler that you want to use maths functions such as `sqrt` you must put `#include <cmath>` at the top of your program, before or after the `#include <iostream>` statement.

Optional: Write a program that prints out all prime numbers less than or equal to N . Hint: use the modulo division operator (%).

10.3 Tips

1. Start with a working program

The easiest way to write programs is to modify the existing programs. Make small modifications, compiling whenever possible.

2. Making modifications

- Mentally check that the program is doing what it is required to do.
- Check that your program is laid out correctly with *indentation* at the beginning and *semi-colons* at the end of each statement.
- Include comment statements to help you and others understand what the program is doing.

3. Checking syntax

Compile, using `g++ -Wall` or `make`. If there are no syntax errors, the program will compile successfully. One or more errors will cause the compiler to print out error or warning messages, which will be listed along with program line numbers indicating where the error occurred. **Remember the tips on page 36** – always use *warnings* when compiling. And save typing by using `make`.

4. Compilation errors

Don't be worried by the number of error messages. The compiler is trying to give as much help as it can. You can get a lot of error messages from just one mistake so **always look at the first error message first**.

Find the line in the file to which it refers and try to correct it. Often, the compiler will report that the error is in a line just *after* the one in which there was a mistake. If you cannot spot the mistake straight away, look at neighbouring lines.

The most common errors will be due to undeclared variables or variables incorrectly declared, and missing or incorrect punctuation. If you are feeling stuck, a really good way to track down the cause of an error message is to explain to a friend, step by step, how you modified a compilable program to get the program with the compilation error.

5. Debugging tips

You don't need to close the editor each time you save the file and recompile; you should have only one copy of your editor running at any time.

Continue with this compile/debug/edit procedure until your program compiles successfully.

6. Finding logical errors

Run your program and check that it gives the correct answer.

10.4 Assessment

Here's the self-assessed bit.

Set yourself an **additional programming task** along the lines of the preceding tasks, and solve it by yourself. Then submit your task statement and the sourcecode of your solution in the same manner as described for assessment 1.

If you finish assessment 2 with time to spare, why not make a start on assessment 3 ?

END OF ASSESSMENT 2

Part IB Computing Course

ASSESSMENT 3

11 Instructions

11.1 Objectives

Here we will test our understanding of:

- Boolean expressions with relational operators
- Simple control structures for selection and repetition

You will now use these skills in the following computing exercises. You'll also learn to use the graphing program, `gnuplot`.

11.2 Computing Exercises

Reminder: programming in pairs is a very good idea, *as long as the weaker programmer does the typing*.

11.2.1 Simple Monte Carlo

Monte Carlo methods use random numbers to generate approximate answers to mathematical questions. A good Monte Carlo method has the property that the more random numbers are used, the more accurate the answer becomes.

The program `RandDemo6.cc` [\[download link\]](#) on the next page shows how to generate N pairs (x, y) uniformly distributed in the unit square, and test whether each pair is in the unit circle or not. The variable N is supplied by the user on the command line. (This non-interactive method of getting inputs from the user is an alternative to using `cin`.) This program has several other new features, some of which we'll explain later. For the moment, don't worry about the lines with `#include`, `#define`, `using`, or `sscanf`.

Make sure you understand everything inside the main `for` loop.

The program uses the 'ternary conditional operator' `"?"` in the assignment

```
outcome = ( x*x + y*y < 1.0 ) ? 1 : 0
```

In general,

```
outcome = ( expression1 ) ? expression2 : expression3
```

means that `outcome` is set equal to `expression2` if `expression1` is true, and to `expression3` if `expression1` is false.

Rewrite the program so as to use an `if-else` in place of this ternary conditional. Explain to a friend how the program works.

When `RandDemo6` runs, the last two columns of its output contain the values of N and `4.0 * fraction_in`. **What do you expect the value of `4.0*fraction_in` to do as N gets big?** (If necessary, recall one of the demos in the introductory lectures of the course.)

```

// RandDemo6.cc
//  usage:
//    ./RandDemo6 [<N> [<seed>]]
//  example:
//    ./RandDemo6                                (Run with default value of N and seed)
//    ./RandDemo6 30 9237832                      (Run with N=30 and seed=9237832)
//    * uses rand() to get a random integer
//    * gets parameters from command line

#include <iostream>
#include <cstdlib>
#include <cstdio>

using namespace std;

#define ranf() \
    ((double)rand()/(1.0+RAND_MAX)) // Uniform from interval [0,1)

int main(int argc, char* argv[])
{
    int    outcome, N=10, count_in=0 , seed=123 ;
    double fraction_in ;

    if(argc>1)  {
        sscanf( argv[1], "%d", &N ) ; // put the first command-line argument in N
    }
    if(argc>2) {
        sscanf( argv[2], "%d", &seed ) ; // put the 2nd argument in seed
    }
    // Write out a summary of parameters
    cout << "# " << argv[0] << " N=" << N << " seed=" << seed << endl ;

    // Initialise random number generator
    srand(seed);

    // Perform N experiments.
    for(int n=1; n<=N; n++) {
        double x = ranf();          // ranf() returns a real number in [0,1)
        double y = ranf();
        outcome = ( x*x + y*y < 1.0 ) ? 1 : 0 ;
        count_in += outcome ;
        //Integer variables must be converted (cast) for correct division
        fraction_in = static_cast<double>(count_in)/n;
        cout << "Location" << outcome << "\t" << x << "\t" << y << "\t"
            << count_in << "\t" << n << "\t" << 4.0 * fraction_in << endl;
    }
    return 0;
}

```


When you are running a program in a terminal (also known as a shell), the symbol `>` can be used to redirect the output of the program to a file. Send the output of `RandDemo6` to a file, using, say,

```
./RandDemo6 20 2389 > tmpfile
```

Start `gnuplot` by typing `gnuplot` in a terminal. When it has started, `gnuplot` gives you a prompt, which looks like this: `gnuplot>`. Plot the generated points (x, y) using, for example,

```
gnuplot> plot 'tmpfile' u 2:3 w points
```

What we mean here is “please type `plot 'tmpfile' u 2:3 w points`”. `gnuplot` has many options for making nice plots. Type `help` to get help. You can change the options interactively – for example,

```
gnuplot> set size square
gnuplot> replot
```

– but I recommend writing the `gnuplot` commands in a plain text file – say `splat.gnu` – and then loading the commands thus

```
gnuplot> load 'splat.gnu'
```

Putting the commands in a file has the advantage that you then have a permanent record of how a plot was generated, should you want to replot or modify it in the future.

Unix provides many utilities for dealing with textfiles. When you are doing scientific computation, we recommend that you write the results of the computation out in plain text files, then handle subsequent inspection or processing of the results with other programs – rather than writing a single monolithic program that does everything. Here’s an example. First run in your terminal:

```
./RandDemo6 200 1234321 > tmpfile
```

– which puts 200 outcomes in `tmpfile`. Now in your terminal run

```
grep Location1 tmpfile > file1
grep -v Location1 tmpfile > file0
```

The first `grep` command looks at every line in `tmpfile` and prints out the lines that contain the string `Location1`. The output went into `file1` because of the redirect (`>`). You can check what happened with the command:

```
more file1
```

The second command uses `grep` with a special flag `-v`, which means ‘invert the match’. Every line in `tmpfile` that does *not* contain the string `Location1` gets printed.

Use `file0`, `file1`, and `gnuplot` to make a pretty picture that you could use to explain to someone how this Monte Carlo method can be used to estimate π . By the way, `gnuplot` can easily plot functions as well as datafiles. For example,

```
gnuplot> plot sqrt(1-x*x) with line lt 3
```

plots the function $\sqrt{1-x^2}$ with a line of colour '3' (which might be blue). `gnuplot` always calls the variable on the horizontal axis `x`.

Having generated N points, what's the estimate of π ? How close do you expect this estimate to be to the true value? **Write down the expected error $\sigma(N)$ of the estimate as a function of N .** Then use `gnuplot` to plot $\pi \pm \sigma(N)$ as a function of N .

You can define and plot functions in `gnuplot` like this:

```
gnuplot> f(N) = 1.0/N
gnuplot> set xrange [1:200]
gnuplot> plot pi+f(x) with lines linetype 5, pi-f(x) with lines linetype 5
```

NB: the function `f(N)` is *not* the answer to the question above.

Now superpose your plot of the theoretical prediction about the estimate $[\pi \pm \sigma(N)]$ on the Monte Carlo estimate (column 6) as a function of N . The estimate can be plotted with

```
plot 'tmpfile' u 5:6 with linespoints
```

Does the Monte Carlo method work as expected? How many points are required to get an answer that's probably accurate to 1%? Do a couple of runs of that length, putting the outputs in separate files (say `tmpfile1` and `tmpfile2`), and see what happens. Zoom in on the interesting **yrange**. (You may be able to zoom in to a plot by right-clicking on the `gnuplot` display. Alternatively, use `set yrange [3.1:3.2]`.)

The output files may get rather large, and slow to plot. **Modify your C++ program so that rather than printing a line for every outcome, it prints a gradually decreasing fraction of the lines as n increases.**

How many points are required to get an answer that's probably accurate to 2 decimal places? To 0.1%? Make a prediction, then check it.

Exercise: Now write a program that estimates $\ln(2)$ to 2 decimal places by a Monte Carlo method.

[Hint: you may find the function $y = 1/x$ interesting to think about.]

11.3 Assessment

If you solved the previous $\ln(2)$ exercise **by yourself**, then please submit your solution to it as you piece of assessed work. It is important that you briefly describe your approach, and include a statement that you made the solution yourself. Submit this in the usual manner.

If on the other hand, you didn't solve the $\ln(2)$ exercise **by yourself**, please set yourself an **additional programming task** along the lines of the preceding tasks, and solve it by yourself. Then submit your task statement and your solution to that task instead, making sure to describe it well.

11.3.1 More efficient Monte Carlo (optional exercise)

How can you modify the Monte Carlo method to reduce the number of random numbers required to obtain $\ln(2)$ to a given accuracy?

11.3.2 Comment

A Monte Carlo approach seems a pretty goofy way to evaluate an integral, but in high-dimensional problems in statistical physics, quantum physics, and statistical inference, Monte Carlo methods are very widely used. See *Information Theory, Inference, and Learning Algorithms*, David J.C. MacKay, Cambridge University Press (2003) – available free online at www.inference.phy.cam.ac.uk/mackay/itila – for further reading about modern Monte Carlo methods.

11.3.3 Testing for randomness (optional exercise)

Returning to the points (x, y) that you plotted a couple of pages ago: do you think these plotted points (x, y) *look* like perfectly random points? If you think *not*, define a measure of non-randomness that captures what you think is wrong with the so-called random points, and write a program to measure this quantity for the points generated by the `random()` routine.

11.3.4 Estimating the value of π II (optional exercise)

Write a program to print out the first N terms in the series:

$$\sum_{i=1}^N \frac{1}{i^2} = 1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16} + \dots + \frac{1}{N^2}$$

The number of terms, N , is to be input by the user at the keyboard or on the command line.

Modify your program so that for each value of the index i , it evaluates the sum of the first i terms.

The sum of this series can be shown (Leonhard Euler (1707-1783)) to converge to $\pi^2/6$. Make another modification to your program so that at each *iteration* the estimate of π is printed alongside the sum. [To use the mathematical library function `sqrt()` you must include the header file `cmath` at the top of your program.] Plot the estimate as a function of N . How good is the estimate of π after $N=100$ iterations? How many iterations are needed to get an estimate of π that is accurate to 2 decimal places?

If you finish assessment 3 with time to spare, why not make a start on assessment 4 ?

END OF ASSESSMENT 3

12 Functions

Computer programs that solve real-world problems are usually much larger than the simple programs discussed so far. To design, implement and maintain larger programs it is necessary to break them down into smaller, more manageable pieces or *modules*. Dividing the problem into parts and building the solution from simpler parts is a key concept in problem solving and programming.

In C++ we can subdivide the functional features of a program into blocks of code known as **functions**. In effect these are subprograms that can be used to avoid the repetition of similar code and allow complicated tasks to be broken down into parts, making the program modular.

Until now you have encountered programs where all the code (statements) has been written inside a single function called `main()`. Every executable C++ program has at least this function. In the next sections we will learn how to write additional functions.

12.1 Function definition

Each function has its own name, and when that name is encountered in a program (the **function call**), execution of the program branches to the body of that function. After the last statement of the function has been processed (the **return** statement), execution resumes on the next line after the call to the function.

Functions consist of a header and a body. The **header** includes the name of the function and tells us (and the compiler) what type of data it expects to receive (the *parameters*) and the type of data it will return (*return value type*) to the calling function or program.

The **body** of the function contains the instructions to be executed. If the function returns a value, it will end with a **return** statement. The following is a formal description of the syntax for defining a function:

```
return-value-type function-name(parameter-list)
{
    declaration of local variables;
    statements;

    return return-value;
}
```

The syntax is very similar to that of the `main` program, which is also a function. `main()` has no parameters and returns the integer 0 if the program executes correctly. Hence the return value type of the `main()` function is `int`.

```
int main()
{
    declarations;
    statements;

    return 0;
}
```

12.2 Example of function definition, declaration and call

Let us first look at an example of program written entirely in the function `main()` and then we will modify it to use an additional **function call**.

We illustrate this with a program to calculate the factorial ($n!$) of an integer number (n) using a `for` loop to compute:

$$n! = 1 \cdot 2 \cdot 3 \dots (n-2) \cdot (n-1) \cdot n$$

```
// MainFactorial.cc
// Program to calculate factorial of a number

#include <iostream>
using namespace std;

int main()
{
    int number=0, factorial=1;

    // User input must be an integer number between 1 and 10
    while(number<1 || number>10)
    {
        cout << "Enter integer number (1-10) = ";
        cin >> number;
    }

    // Calculate the factorial with a FOR loop
    for(int i=1; i<=number; i++)
    {
        factorial = factorial*i;
    }

    // Output result
    cout << "Factorial = " << factorial << endl;
    return 0;
}
```

Even though the program is very short, the code to calculate the factorial is best placed inside a function since it is likely to be executed many times in the same program or in different programs (e.g., calculating the factorials of many different numbers, computing binomial coefficients and permutations and combinations).

```

// FunctionFactorial.cc
// Program to calculate factorial of a number with function call

#include <iostream>
using namespace std;

// Function declaration (prototype)
int Factorial(int M);

int main()
{
    int number=0, result;

    // User input must be an integer number between 1 and 10
    while(number<1 || number>10)
    {
        cout << "Integer number = ";
        cin >> number;
    }

    // Function call and assignment of return value to result
    result = Factorial(number);

    //Output result
    cout << "Factorial = " << result << endl;
    return 0;
}

// Function definition (header and body)
// An integer, M, is passed from caller function.
int Factorial(int M)
{
    int factorial=1;

    // Calculate the factorial with a FOR loop
    for(int i=1; i<=M; i++)
    {
        factorial = factorial*i;
    }

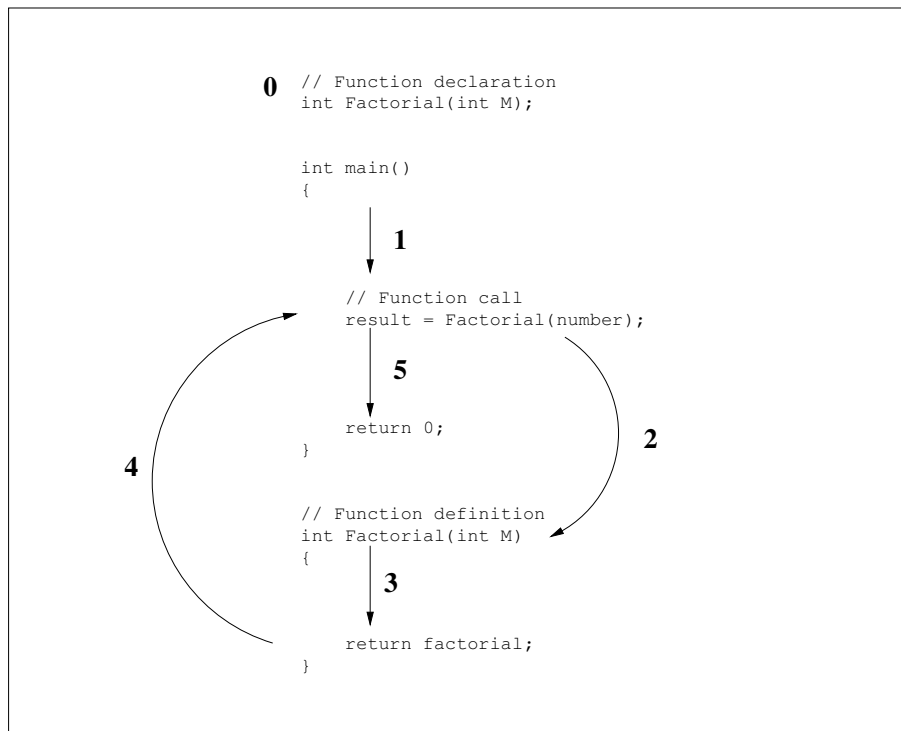
    return factorial; // This value is returned to caller
}

```

Three modifications to the program have been made to incorporate a function. If we look at the modified sample program, `FunctionFactorial.cc`, we find:

1. The **declaration** of the function above the main program. The declaration (also known as the *prototype*) tells the compiler about the function and the *type* of data it requires and will *return* on completion.
2. The **function call** in the main body of the program determines when to branch to the function and how to return the value of the data computed back to the main program.
3. The **definition** of the function `Factorial()` below the main program. The definition consists of a **header**, which specifies how the function will interface with the main program, and a **body**, which lists the statements to be executed when the function is called.

Before a function can be used it must be declared (0), usually at the top of the program file.²³ When the function name is encountered in the main body of the program (1), execution branches to the body of the function definition (2). Copies of the values of function arguments are stored in the memory locations allocated to the parameters. The statements of the function are then executed (3) up to the first **return** statement when control returns to the main body (4). Any value returned by the function is stored by an assignment statement. Execution in the main body is resumed immediately after the function call (5).



12.3 Function header and body

The function is defined below the body of `main()`. The **header** in this example:

```
int Factorial(int M)
```

²³Strictly speaking, it's not always essential to **declare** a function before it is used, as long as the compiler encounters the function's **definition** before the function is used.

indicates that the `Factorial()` function expects to be **passed** an integer value (the *parameter type*) from the `main` body of the program and that the value passed will be stored locally in a variable named `M` (the *formal parameter name*). The *return value type* of the function is also `int` in this example, indicating that at the end of executing the body of the function, an integer value will be returned to the statement in which the function was called. Functions that do not return a value have *return value type* `void`.

The **body** of the function computes the factorial of a number in exactly the same way as in the example with only a `main()` function. The execution of the function terminates with a *return statement*:

```
return factorial;
```

which specifies that the value stored in the function variable `factorial` should be passed back to the calling function.

12.4 Function declaration

Every function should be **declared** before it is used. The declaration tells the compiler the name, return value type and parameter types of the function. In this example the declaration:

```
int Factorial(int M);
```

tells the compiler that the program passes the value of an integer to the function and that the *return value* must be assigned to an integer variable. The declaration of a function is called its **prototype**, which means the “first” time the function is identified to your program.

The function prototype and the function definition must agree exactly about the return value type, function name and the parameter types. The function prototype is usually a copy of the function header followed by a semicolon to make it a declaration and placed before the main program in the program file.

12.5 Function call and execution

The function definition is entirely passive. By itself it does nothing unless instructed to execute. This is done by a statement in the main program called the **function call**.

For example the statement:

```
result = Factorial(number);
```

calls the function `Factorial()` and passes a copy of the value stored in the variable, `number`. When the function is called, computer memory is allocated for the parameter, `M`, and the value passed is copied to this memory location. Memory is also allocated to the (local) variables `factorial` and `i`. The statements of the function are then executed and assign a value to the variable `factorial`. The **return** statement passes this value back to the calling function. The memory allocated to the parameters and local variables is then destroyed. The value returned is assigned to the variable on the left-hand side, `result`, in the expression used to call the function. The net effect of executing the function in our example is that the variable `result` has been assigned the value of the factorial of `number`.

A function can be called any number of times from the same or different parts of the program. It can be called with different parameter values (though they must be of the correct type). For example the following fragment of code can be used to print out the factorials of the first 10 integers:


```

for(int i=1; i<=10; i++)
{
    result = Factorial(i);
    cout << i << "! = " << result << endl;
}

```

and:

```

binomialCoefficient = Factorial(n)/(Factorial(k) * Factorial(n-k));

```

can be used to compute the binomial coefficient:

$$\frac{n!}{k!(n-k)!}$$

(though this is *not* the best way to compute a binomial coefficient!).

12.6 Function arguments

The names of variables in the statement calling the function will not in general be the same as the names in the function definition, although they must be of the same type. We often distinguish between the **formal parameters** in the function definition (e.g. `M`) and the **actual parameters** for the values of the variables passed to the function (e.g. `number` in the example above) when it is called.

Function arguments (actual parameters) can include constants and mathematical expressions. For example the following statement assigns the value 24 to the variable `result`.

```

result = Factorial(4);

```

The function arguments can also be functions that return an appropriate value (for example `Factorial(Factorial(4))`), although this concise style makes the code harder to read and debug.

12.7 Another example

Here is another example of the declaration, definition and call of a function, `AreaTriangle()`, in a program to calculate the area of a regular hexagon inscribed in a circle of radius input by the user.

```
// HexagonValue.cc
// Program to calculate the area of a regular hexagon inscribed in a
// circle as sum of areas of 6 triangles by calling AreaTriangle()

#include <iostream>
using namespace std;

// AreaTriangle function prototype
float AreaTriangle(float base, float height);

int main()
{
    float side, segmentHeight, hexagonArea;
    float cosTheta = 0.866025;

    cout << "Program to calculate the area of a hexagon" << endl;
    cout << "Enter side of hexagon: ";
    cin >> side;

    // Base of triangle is equal to side, height is side*cos(30)
    segmentHeight = side*cosTheta;

    // Function returns area of segment. 6 segments for total area.
    hexagonArea = 6.0f * AreaTriangle(side,segmentHeight);

    cout << "Area of hexagon = " << hexagonArea << endl;
    return 0;
}

// AreaTriangle function definition
float AreaTriangle(float base, float height)
{
    float area;

    area = (base*height)/2.0f;
    return area;
}
```

The statement:

```
hexagonArea = 6.0f * AreaTriangle(side,segmentHeight);
```

calls the function to calculate the area of a triangle with base and height given by the values stored in `side` and `segmentHeight` and then assigns the value of 6 times the area (the return value of the function) to the variable `hexagonArea`. It is therefore equivalent to the following:

```
segmentArea = AreaTriangle(side,segmentHeight);  
hexagonArea = 6.0f*segmentArea;
```

Note that `6.0f` has been used instead of plain `6.0` to suppress a compiler warning about the possibility that we might care whether `6.0` was represented as a single-precision or double-precision number. The ‘f’ says we want it to be a single-precision float. But we don’t really care here.

12.8 Passing by value or reference

There are two ways to pass values to functions. Up to now we have looked only at examples of **passing by value**. In the *passing by value* way of passing parameters, a *copy* of the variable is made and passed to the function. Changes to that copy do not affect the original variable’s value in the calling function. This prevents the accidental corrupting of variables by functions and so is often the preferred method for developing correct and reliable software systems. One disadvantage of *passing by value* however, is that only a single value can be returned to the caller. If the function has to modify the value of an argument or has to return more than one value, then another method is required.

An alternative uses **passing by reference** in which the function is told where in memory the data is stored (i.e., the function is passed the memory address of the variables). In passing the address of the variable we allow the function to not only read the value stored but also to change it.

Here’s an analogy. When you tell a friend about a webpage on the course wiki, you can either print out a *copy* of the webpage for them; or you can send them the URL of the webpage. Giving them the copy is ‘pass by value’; giving them the URL is ‘pass by reference’. When you ‘pass by value’, they can manipulate their copy of the page, and their manipulations have no effect on the original page. When you ‘pass by reference’, they will be able to edit the contents of the webpage.

There are two ways to pass a parameter by reference. The simple (and rather subtle) way is described here in the example `SortReference.cc`. An alternative approach to passing by reference is demonstrated on the web in `SortReference2.cc`. [\[Download Link\]](#)

To indicate that a function parameter is *passed by reference* the symbol `&` is placed next to the variable name in the parameter list of the function definition and prototype (but **not** the function call). Inside the function the variable is treated like any other variable. Any changes made to it, however, will automatically result in changes in the value of the variable in the calling function.

A simple example of passing by reference is given below for a function that swaps the values of two variables in the calling function’s data by reading and writing to the memory locations of these variables. Note that the parameters are mentioned only by name in the function call. This appears to be the same as *calling by value*. The function header and prototype, however, must use the `&` symbol by the variable name to indicate that the call is by reference and that the function can change the variable values.

```

// SortReference.cc
// Program to sort two numbers using call by reference.
// Smallest number is output first.

#include <iostream>
using namespace std;

// Function prototype for call by reference
void swap(float &x, float &y);

int main()
{
    float a, b;

    cout << "Enter 2 numbers: " << endl;
    cin >> a >> b;
    if(a>b)
        swap(a,b);

    // Variable a contains value of smallest number
    cout << "Sorted numbers: ";
    cout << a << " " << b << endl;
    return 0;
}

// A function definition for call by reference
// The variables x and y will have their values changed.

void swap(float &x, float &y)
// Swaps x and y data of calling function
{
    float temp;

    temp = x;
    x = y;
    y = temp;
}

```

13 Math library and system library built-in functions

Functions come in two varieties. They can be defined by the user or built in as part of the compiler package. As we have seen, user-defined functions have to be declared at the top of the file. Built-in functions, however, are declared in **header files** using the `#include` directive at the top of the program file, e.g. for common mathematical calculations we include the file `cmath` with the `#include <cmath>` directive, which contains the *function prototypes* for the mathematical functions in the `cmath` library.

13.1 Mathematical functions

Math library functions allow the programmer to perform a number of common mathematical calculations:

Function	Description
<code>sqrt(x)</code>	square root
<code>sin(x)</code>	trigonometric sine of x (in radians)
<code>cos(x)</code>	trigonometric cosine of x (in radians)
<code>tan(x)</code>	trigonometric tangent of x (in radians)
<code>exp(x)</code>	exponential function
<code>log(x)</code>	natural logarithm of x (base e)
<code>log10(x)</code>	logarithm of x to base 10
<code>fabs(x)</code>	absolute value (unsigned)
<code>ceil(x)</code>	rounds x up to nearest integer
<code>floor(x)</code>	rounds x down to nearest integer
<code>pow(x,y)</code>	x raised to power y

13.2 Random numbers

Other **header files** that contain the function prototypes of commonly used functions include `cstdlib` and `ctime`. These contain functions for generating random numbers and for manipulating time and dates respectively. If you want to use these functions then you would need to have lines like `#include <cstdlib>` or `#include <ctime>` at the top of your program.

The function `rand()` *randomly* generates an integer between 0 and the maximum value that can be stored as an integer. Every time the function is called:

```
randomNumber = rand();
```

a different number will be assigned to the variable `randomNumber`. Each number is supposed to have an equal chance of being chosen each time the function is called. The details of how the function achieves this will not be discussed here.

Before a random number generator is used for the first time it should be initialised by giving it a number called the *seed*. Each seed will result in a different sequence of numbers. The function `srand()` initialises the random number generator, `rand()`. It must be called with an arbitrary integer parameter (the *seed*). If you want fresh random numbers every time, you can conveniently generate a fresh seed by using the value returned by the system clock function `time()` with the parameter `NULL`. This returns the calendar time in seconds, converted to an integer value. The following call, which is usually used only once, can be used to initialise the random number generator:

```
srand(time(NULL));
```

Alternatively, if you want to make random experiments *that can be exactly reproduced*, you might prefer to give the user explicit control of the seed.

The `RandDemo6.cc` program used these system functions. Here's another example.

```

// Function to simulate rolling a single 6-sided die.
// Function takes no arguments but returns an integer.
// Each call will randomly return a different integer between 1 and 6.

int RollDie()
{
    int randomNumber, die;

    randomNumber = rand();
    die = 1 + (randomNumber % 6);
    return die;
}

```

13.3 How can I find out what library a function is in?

Sometimes you know the name of a function you want to use (e.g. `cosh`) but don't know the name of the library it might be in. There is a simple unix terminal command that will tell you straight away. For example, to find out which library `cosh(x)` is in, and how to use it, type `man 3 cosh` into a terminal window, and you should see a response looking something like this:

```

COSH(3)                                Linux Programmer's Manual                                COSH(3)

NAME
    cosh, coshf, coshl - hyperbolic cosine function

SYNOPSIS
    #include <math.h>

    double cosh(double x);
    float coshf(float x);
    long double coshl(long double x);

    Link with -lm.

DESCRIPTION
    The cosh() function returns the hyperbolic cosine of x,
    which is defined mathematically as (exp(x) + exp(-x)) / 2.

CONFORMING TO
    SVr4, POSIX, 4.3BSD, C99. The float and the long double
    variants are C99 requirements.

SEE ALSO
    acosh(3), asinh(3), atanh(3), ccos(3), sinh(3), tanh(3)

```

2002-07-27

COSH(3)

which indicates not only the name of the header file you will have to include (in this case “**math.h**”) but also supplied info on related functions that are in the same library. Press **Q** to quit viewing this manual page. Note that in C++ programs, you often see **cmath** instead of **math.h**. You can use them interchangeably, but the former is to be preferred in C++ programs, and the latter in C programs.

Part IB Computing Course

ASSESSMENT 4

14 Instructions

14.1 Objectives

After reading through sections 7 and 8 of the tutorial guide and studying and executing the C++ programs in the examples (boxed) you should now be familiar with

- Definition, declaration and calling of functions
- Passing values to and returning values from functions
- Math and system library functions

You will now use these skills in the following computing exercises. The information in sections C and D will help you to think about what is involved in producing a working solution. Suggestions for the *algorithms* (mathematical recipes) and their implementation in C++ are provided.

14.2 Computing Exercises

14.2.1 Finding a solution to $f(x) = 0$ by iteration

Write a function that computes the square root of a number in the range $1 < x \leq 100$ with an accuracy of 10^{-4} using the bisection method. The Math library function *must not* be used.

Test your function by calling it from a program that prompts the user for a single number and displays the result.

Modify this program so that it computes the square roots of numbers from 1 to 10. Compare your results with the answers given by the `sqrt()` mathematical library function.

14.3 The bisection method

The problem of finding the square root of a number, c , is a special case of finding the root of a non-linear equation of the form $f(x) = 0$ where $f(x) = c - x^2$. We would like to find values of x such that $f(x) = 0$.

A simple method consists of trying to find values of x where the function changes sign. We would then know that one solution lies somewhere between these values. For example: If $f(a) \times f(b) < 0$ and $a < b$ then the solution x must lie between these values: $a < x < b$. We could then try to narrow the range and hopefully converge on the true value of the root. This is the basis of the so-called bisection method.

The bisection method is an iterative scheme (repetition of a simple pattern) in which the interval is halved after each iteration to give the approximate location of the root. After i iterations the root (let's call it x_i , i.e., x after i iterations) must lie between

$$a_i < x_i \leq b_i$$

and an approximation for the root is given by:

$$p_i = \frac{(a_i + b_i)}{2}$$

where the error between the approximation and the true root, ϵ_i , is bounded by:

$$\epsilon_i = \pm \frac{(b_i - a_i)}{2} = \pm \frac{(b_1 - a_1)}{2^i}.$$

At each iteration the sign of the functions $f(a_i)$ and $f(p_i)$ are tested and **if** $f(a_i) \times f(p_i) < 0$ the root must lie in the half-range $a_i < x < p_i$. Alternatively the root lies in the other half (see figure). We can thus update the new lower and upper bound for the root:

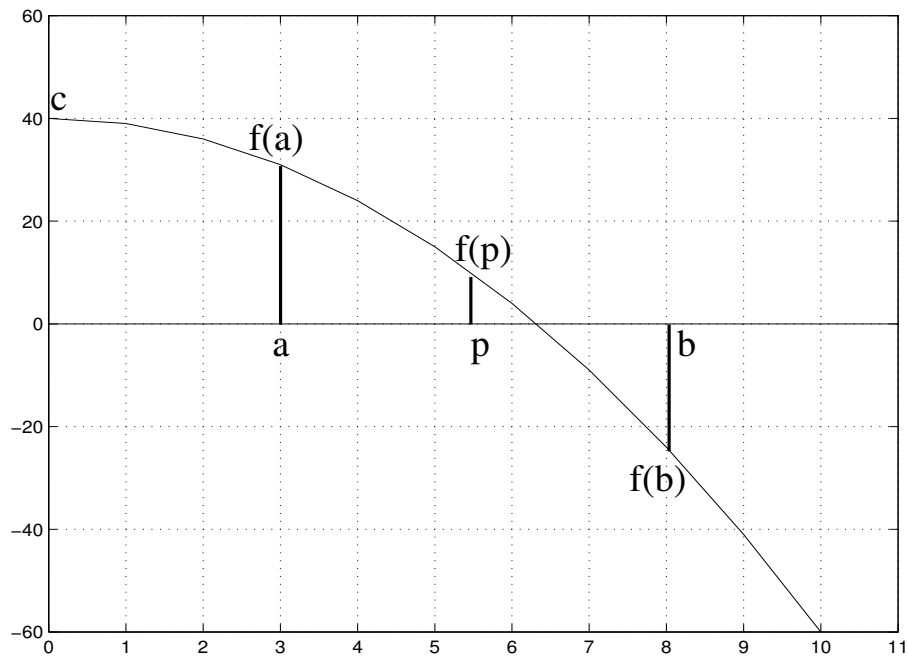
if $f(a_i) \times f(p_i) < 0$ then $a_{i+1} = a_i$ and $b_{i+1} = p_i$

else $a_{i+1} = p_i$ and $b_{i+1} = b_i$

The bisection method is guaranteed to converge to the true solution but is slow to converge since it uses only the sign of the function. An alternative is the Newton–Raphson method, which takes into account the gradient of the function, $f'(x)$, and only needs one starting guess for the root. In the Newton–Raphson method the next approximation to the root is given by:

$$p_{i+1} = p_i - \frac{f(p_i)}{f'(p_i)}.$$

14.4 Notes on algorithm and implementation



The square root of a number will be found by calling a user-defined function to implement one of the iterative algorithms (i.e., repetition of a pattern of actions) described above. Review tutorial section 12, which describes how to define a function and how to pass parameters to the function and return values to the main program.

You are required to define a function to find the square root of a number c , i.e., find the zeroes of $f(x) = c - x^2$. Your solution is to be accurate to 5 decimal places. Since the number input by the user is between 1 and 100 the root will satisfy $0 < x \leq 10$ and a valid initial guess for the lower and upper bound for the solution will always be $a_1 = 0.1$ and $b_1 = 10.1$. The error after i iterations will be $\pm \frac{10}{2^i}$. To produce a solution that is accurate to 5 decimal places we will need more than 20 iterations.

1. Getting Started:

Start with a very simple program that prompts the user for the value of a real number in the range $1 < c \leq 100$. Alternatively, steal `RandDemo6.cc`'s method of getting the float c from the command-line (something like `sscanf(..., "%f", &c)`, and not forgetting to `#include` the relevant headers).

2. Function definition

You are required to **define** a *function*, `MySquareRoot()`, which is *passed* the number (i.e., a single parameter of type `float`) and *returns* the approximate value of its square root (i.e., return value type is `float`).

The C++ code for the function should be placed below the body of `main()`. Begin the implementation of the function by typing in the *function header* and the opening and closing braces. For example:

```
float MySquareRoot(float square)
{
    // Body of function definition
}
```

3. Function body and implementation of algorithm

Inside the body of the function (i.e., after the opening brace):

- (a) You will need to declare *local variables* to store the values of a_i , b_i , p_i and $f(a_i) \times f(p_i)$. For example: **lower**, **upper**, **root** and **sign**. Initialize the values of **lower** and **upper** to 0.1 and 10.1 respectively. (A possible additional programming goal would be to make a smarter initialization method such that square roots can be found for any c in a larger range.)
- (b) Set up a loop using the **while** or **for** repetition control statements to repeat the following algorithm (bisection method) at least 20 times.
- (c) In each execution of the loop:
 - Estimate the value of the root as the average of the lower and upper bounds. Store this value in variable **root**.
 - Evaluate the function at the current value of lower (i.e., a_i) and at the current estimate of the root, (p_i).
 - Evaluate $f(a_i) \times f(p_i)$ and store this value in variable **sign**.
 - Depending on the value of **sign** update the lower and upper bounds by the bisection method described above.
- (d) The function must end with a **return** statement to pass back the approximate value of the square root.

4. Function declaration:

Declare the function by including the function prototype before `main()`. Compile your program to make sure you have not made any typing or syntax errors.

5. Testing of function:

Call your function from your program with the (actual) parameter, e.g. `number`. The *return value* of the function is to be assigned to a variable (e.g. `squareRoot`) which should also be declared in `main()`:

```
squareRoot = MySquareRoot(number);
```

Test your function creatively.

6. Loop

Set up a loop in the main routine to *call* the function 10 times to calculate the square roots of the integers 1, 2, ... 10.

14.5 Assessment

Set yourself an **additional programming task** along the lines of the preceding tasks, and solve it by yourself. Then submit your task statement and your solution in the usual manner.

If you finish assessment 4 with time to spare, why not start to prepare for the remaining two assessments? They are harder than those we have done so far, and gain more credit.

END OF ASSESSMENT 4

14.6 ‘Have I done enough?’

If you find yourself asking “Have I done enough for this week’s session?”, the answer is “please self-assess!” Do you feel you’ve got a firm grasp of all the highlighted concepts? If so, then that’s enough. One thing to check is “*have I developed my understanding of this week’s physics topic?*” That’s one of the aims this term – to get fresh insights into orbits, dynamics, waves, statistical physics, pressure, temperature, and so forth.

More programming concepts

The new programming tools we'll use now are arrays (great for representing vectors, or lists of similar things); structures (great for organizing things that belong together, such as a particle's position, mass, and velocity, or a vector and its range of indices); and how to modularize your code.

We now cover the following topics.

arrays	how to allocate memory on the fly for things like vectors
modularizing	chopping up your code so you can reuse it elegantly
structures and packages	how to use structures in your elegant modules
formatted output	getting the number of decimal places that you want

15 Arrays

More realistic programs need to store and process a substantial number of items of data. The types of variable considered so far have been simple data types capable of holding a single value.

Arrays are a consecutive group of variables (memory locations) that all have the same type and share a common name. In many applications, arrays are used to represent vectors and matrices. We now describe a simple approach to arrays. A more elegant approach will be described in due course.

15.1 Declaration

An array is declared by writing the type, followed by the array name and **size** (number of elements in the array) surrounded by square brackets.

<i>type</i>	<i>array-name</i> [<i>number-elements</i>];
float	position[3];
int	count[100];
char	YourSurname[50];

The above examples declare `position` to be an array that has 3 elements which are real numbers (e.g., 3D vector). `YourSurname` contains 50 characters (e.g., storing a surname) and `count` is an array of 100 integer values.

An array can have any legal variable name but it cannot have the same name as an existing variable. The array's size is fixed when the array is declared and remains the same throughout the execution of the program.

15.2 Array elements and indexing

To refer to a particular element of the array we specify the name of the array and the *position number* or **index** of the particular element. Array elements are **counted from 0** and not 1. The

first elements will always have position and index 0 and the last element will have index number (position) N-1 if the array has N elements.

The first elements in the arrays declared above are referred to by `position[0]`, `count[0]` and `YourSurname[0]` respectively. The last element of an array declared as `array[N]` with N elements is referred to by `array[N-1]`. The last elements of the example arrays above are therefore referred to by `position[2]`, `count[99]` and `YourSurname[49]` respectively.

The index of an array can also be an expression yielding an integer value.

15.3 Assigning values to array elements

Elements of an array can be treated like other variables but are identified by writing the name of the array followed by its index in square brackets. So for instance the statements:

```
marks[1] = 90.0;
scaled[1] = (marks[1] - mean)/deviation;
```

show how the second element of an array called `marks` is assigned a value and how the second element of the array `scaled` is assigned the result of a calculation using this element value.

What is useful about arrays is that they fit well with the use of loops. For example:

```
int count[100];
for(int i=0; i< 100; i++)
{
    count[i] = 0;
}
```

can be used to **initialise** the 100 elements of an integer array called `count` by setting them all to 0. The following program uses arrays and loops to calculate the scalar product of two vectors input by the user.

```

// ScalarProduct.cc
// Calculating the scalar product between vectors input by user

#include <iostream>
using namespace std;

int main()
{
    float vectorA[3], vectorB[3], scalar=0.0;

    // Get input vectors from user.
    cout << "Enter elements of first vector: " << endl;
    for(int i=0;i<3;i++)
    {
        cin >> vectorA[i];
    }
    cout << "Enter elements of second vector: " << endl;
    for(int i=0;i<3;i++)
    {
        cin >> vectorB[i];
    }

    // Calculate scalar product.
    for(int i=0;i<3;i++)
    {
        scalar = scalar + (vectorA[i] * vectorB[i]);
    }

    // Output result.
    cout << "The scalar product is " << scalar << endl;
    return 0;
}

```

Note: care must be taken never to try to assign array elements that are not defined. For example, there's nothing to stop you from including the statement `cin >> vectorA[723]` in your program. The program will compile and will execute, but the consequences of such assignments on the execution of the program are unpredictable and very difficult to detect and debug. It is left to the programmer to check that the array is big enough to hold all the values and that undefined elements are not read or assigned. Assigning array values outside the permitted range is probably one of the most common programming errors. An object-oriented approach to programming, which we'll discuss in due course, can reduce the risk of such programming errors. There are also debugging utilities, which can help spot memory mismanagement. We'll discuss these later in the course.

15.4 Passing arrays to functions

15.4.1 Function definition and prototype

To pass an array to a function, the array type and name is included in the formal parameter list (i.e., in the function definition and prototype parameter list). The following is a function header for a function that receives an array of real numbers.

```
void NormaliseData(float arrayName[], int arraySize)
```

15.4.2 Function call

In C++ the entire array is passed using the array name (and **no** square brackets). This looks similar to *pass by value* but is actually a pass by reference. The actual parameter passed to the function, the name of the array, is in fact the memory *address* of the first element of the array. The important point to note here is that (as with passing by reference) when you pass an array to a function you will be able to access and modify the values of any of its elements. Arrays cannot be passed by value.

In the following call to the `NormaliseData()` function, a 300 element array, `marks`, is passed to the function:

```
NormaliseData(marks, 300);
```

The function can read and change the value of any of the elements of the array.

`RotationMatrix.cc` on page 86 is a working example.

If you need to pass only a single element of the array and you want to take advantage of the simplicity and safety of passing by value, this can be done, for example, by having `marks[2]` as an actual parameter in the call to the function. For example in the following function:

```
float ProcessIndividual(float mark);           //Function prototype
scaledMark = ProcessIndividual(marks[2]);      //Function call
```

the call passes the value of `marks[2]`, which it stores as a local variable (`mark`). Changes to the value of the local variable will not affect the value in the calling function.

Another way to enforce safe passing of an array that should *not* be modified by a function is to declare the array `const` from the point of view of the function, as illustrated by this function:

```
void print1( const float array[] , int N ) {
    for ( int n = 0 ; n < N ; n ++ ) {
        cout << array[n] << " " ;
    }
    cout << endl;
}
```

15.5 Character arrays

A common use of the one-dimensional array structure is to create character strings. A character string is an array of type `char` that is terminated by the `null` terminator character, `'\0'`. The symbol `\0` looks like two characters but represents one. It is called an *escape sequence*.

A character array can be *initialised* on declaration by enclosing the characters in double quotes. The null character will be automatically appended.

```

// CharacterArray.cc
// Initialisation of a character array and passing to functions

#include <iostream>
using namespace std;

void PrintExtensionNumber(char phoneNumber[]);

int main()
{
    char phone[11];
    char prompt[] = "Enter telephone number: ";

    // Get phone number
    cout << prompt ;
    cin >> phone ;

    // If first two digits are 3 then it is a university number.
    if( phone[0]=='3' && phone[1]=='3')
    {
        cout << "University extension ";
        PrintExtensionNumber(phone);
    }
    else
    {
        cout << "Dial 9-" << phone[0];
        PrintExtensionNumber(phone);
    }

    return 0;
}

// Function to print out a phone number, ignoring the first digit.
void PrintExtensionNumber(char phoneNumber[])
{
    for(int i=1; phoneNumber[i] != '\0'; i++)
    {
        cout << phoneNumber[i];
    }
    cout << endl;
}

```

15.6 Multi-dimensional arrays

The arrays that we have looked at so far are all one-dimensional arrays, however, C++ allows multidimensional arrays. For example to declare a two-dimensional array to represent the data of a 3 by 4 matrix called `myMatrix` we could use the following statement and syntax:


```
float  myMatrix[3][4];
```

You may think of the element with indices `i` and `j`, `myMatrix[i][j]`, as the matrix element in the `i` row and `j` column, but remember that array indices always start from 0.

The following program illustrates the passing of arrays to and from functions. The function `ComputeMatrix()` assigns values to the elements of a 2 by 2 rotation matrix. The actual parameters passed are the name of the rotation matrix (i.e., memory address of the first element) and an expression to determine the rotation angle in radians. The element `matrix[0][1]` is the element of the first row and second column of the matrix.

The function `RotateCoordinates()` computes the coordinates of a point after transformation by the following equation:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

It is passed the name of the matrix and the name of the initial position vector and transformed position vector.

```

// RotationMatrix.cc
// Program to calculate coordinates after rotation

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

void ComputeMatrix(float matrix[2][2], float angle);
void RotateCoordinates(float rot[2][2], float old[2], float transformed[2]);

int main()
{
    float angle, point[2], transformedPoint[2];
    float rotation[2][2];

    // Get angle of rotation and initial position from input.
    cout << "Enter magnitude of rotation in xy plane in degrees: " ;
    cin >> angle;
    cout << "Input x and y coordinates: " << endl;
    cin >> point[0] >> point[1];

    // Calculate coefficients of rotation matrix and transform point.
    // The value of pi is declared in math as M_PI.
    ComputeMatrix(rotation, (M_PI*angle/180.0));
    RotateCoordinates(rotation, point, transformedPoint);

    // Output result.
    cout << "The (x,y) coordinates in the rotated system are ";
    cout << transformedPoint[0] << " and " << transformedPoint[1] << endl;
    return 0;
}

void ComputeMatrix(float matrix[2][2], float angle)
{
    matrix[0][0] = cos(angle);
    matrix[0][1] = sin(angle);
    matrix[1][0] = -sin(angle);
    matrix[1][1] = cos(angle);
}

void RotateCoordinates(float rot[2][2], float old[2], float transformed[2])
{
    transformed[0] = (rot[0][0] * old[0]) + (rot[0][1] * old[1]);
    transformed[1] = (rot[1][0] * old[0]) + (rot[1][1] * old[1]);
}

```

15.7 Structures

Arrays are examples of data structures in which all the elements must be of the same type. C++ allows the user to define more general data structures, using the keyword **struct** to define a collection of related variables of any type, called a **structure**. For example:

```
struct StudentType{
    char   name[100];
    int    age;
    int    entryYear;
    float  marks[5];
    char   college[20];
};
```

defines a new data type called **StudentType**, which is made up of five *fields* or *data members*: two integers, an array of floating point numbers and two character arrays. The body of the structure definition is delineated by braces and must end with a semicolon. The definition is placed at the top of a program file, between include directives and the function prototypes.

Once a structure has been defined, the structure name can be used to declare objects of that type. This is analogous to declaring simple variables.

```
StudentType person;
StudentType firstYear[400];
```

declares the variable **person** and the one-dimensional array, **firstYear[400]** to be of type **StudentType**.

Data members of a structure are accessed and assigned values by specifying the field (data member) name using the *dot operator*, for example:

```
person.age = 19;
firstYear[205].entryYear = 1999;
```

Structures have the advantage that, by collecting related items together, they can be manipulated as single items. For example whole structures can be copied using an assignment statement (unlike arrays):

```
firstYear[24] = person;
```

and manipulated efficiently with repetition control statements:

```
for(int i=0; i< 400; i++)
{
    firstYear[i].entryYear = 1999;
}
```

15.8 Enumerated constants

There are many examples of data that are not inherently numeric. For example, the days of the week, months of the year, colours. We can refer to such data types by defining symbolic constants and using these symbolic constants in expressions in the program. For example:

```
const int Mon=0, Tue=1, Wed=2, Thu=3, Fri=4, Sat=5, Sun=6;
```

C++ provides a more convenient way for the user to define a new data type. The C++ **enumeration** statement:

```
enum Days {Thu, Fri, Sat, Sun, Mon, Tue, Wed};
```

creates a new variable type with legal values `Thu`, `Fri`, ..., `Wed`, which are in fact **symbolic constants** for 0, 1, ..., 6. The enumeration is simply assigning an integer to a symbolic constant. The definition makes it convenient to work with days of the week.

Variables can be declared as having the user-defined type and this will ensure that they are only assigned one of the legal values. The following statement declares the variable `day` to have the user-defined type, `Days`.

```
Days    day;
```

16 Reading and writing to files

Storage of data in variables and arrays is temporary. Files are used for permanent retention of large amounts of data. We will now show how C++ programs can process data from files. To perform file processing in C++, the header file `<fstream>` must be included. The latter includes the definitions of `ifstream` and `ofstream` *classes* (special structure definitions). These are used for input from a file and output to a file.

The following command opens the file called, `name.dat`, and reads in data, which it stores sequentially in variables `a` and `b`:

```
ifstream fin;
fin.open("name.dat");
fin >> a >> b;
```

The *object* called `fin` is declared to be of type (*class*) `ifstream`. The *member function* called `open()` is used to associate the file name, `name.dat`, with the object name, `fin`.

In a similar way we can write data (in this example, the values of an array) to a file called `output.dat` with:

```
ofstream fout;
fout.open("output.dat");
for(int i=0; i<N; i++)
{
    fout << array[i] << endl;
}
```

`fin` and `fout` are `ifstream` and `ofstream` *objects* respectively. Note that `fin` and `fout` are arbitrary names assigned by the programmer. Using `fin` and `fout` highlights the similarity with the simple input and output statements of section 8, which use the input and output stream objects `cin` and `cout` respectively. The syntax of the input and output statements is identical. The only difference is that care must be taken to ensure the files are opened and closed correctly, i.e., that the file exists and is readable or writable. This can be checked by testing the value of `fin.good()` or `fout.good()`. These will have the values `true` if the files are opened correctly.

The program on the next page reports an error if the file name specified was not found or could not be opened and prompts the user for another file name. After finishing reading from and writing to files, the files must be **closed** using:

```
fin.close();
fout.close();
```

and the `ifstream` and `ofstream` *objects* `fin` and `fout` can be re-assigned.

```
// OpenFile.cc
// Program to read data from a file. File name is input by user.

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

int main()
{
    char fileName[80];

    // Get filename from keyboard input
    cout << "Enter input file name: ";
    cin >> fileName;

    // Declare fin to be of type (class) ifstream
    // Associate file name with fin
    ifstream fin;
    fin.open(fileName);

    // Prompt for new file name if not able to open
    while(fin.good() == false)
    {
        cout << "Unable to open file. Enter a new name: ";
        cin >> fileName;
        // once bad, the file stream will stay bad unless cleared
        fin.clear();
        fin.open(fileName);
    }
    return 0;
}
```

Health Warning. The following section is included because it's good to see examples of how blocks of memory can be allocated. However, if you find yourself needing to use arrays (or array-like things) with sizes determined at run time in C++ you almost certainly would be better-off using one of the “Standard Template Library Container Classes” such as the the “`std::vector`”. Sadly, these are not covered in this course, though they feature in the Part II course.

17 Direct allocation of arrays with “new”

Above, we declared arrays of fixed size like this

```
float      position[3];
int        count[100];
for(int i=0; i< 100; i++)
{
    count[i] = 0;
}
```

This fixed-size approach is an inflexible way of doing things, and it hard-wires numbers like ‘3’ and ‘100’ into your code, whereas such numbers should almost always be parameters. It would be more elegant to replace the ‘100’ above by a name (such as *N*), and use that name everywhere, and fix the value of *N* just once at the start of the program. It can sometimes be useful to allow *N* to be set interactively or on the command-line of the program – in the way that `RandDemo6.cc`, for example, could optionally set the number of points *N* (page 34 of tutorial 1). The ugly way of doing arrays does not allow the creation of arrays of size controlled by a variable.

In C++ we can create variable-sized arrays on the fly while a program is running. We use a special command `new` to allocate the memory for an array, and a complementary command `delete` to free up the memory again when we don’t need it any more.

The example program `NewDelete.cc` illustrates how to use `new` and `delete`, and how to pass an array to a function.

```
// NewDelete.cc
#include <iostream>
using namespace std;

void show( double *a , int N ){
    for( int n=0; n<N; n++ )
        cout << "a["<<n<<"]="<<endl ;
}

int main()
{
    double *a ;
    // This creates a pointer but doesn't allocate any memory for the array

    int    N = 20 ;

    a = new double[N] ; // allocates memory for a[0]..a[N-1]
    for( int n=0; n<N; n++ )
        a[n] = n*n ;
    show( a , N ) ;
    delete [] a ;      // frees the memory
}
```

In C++, array indices by default run from zero. But if you want an array that runs from, say, 1 to *N*, then you can do this by creating an array `a[0]...a[N-1]` of the right size, then offsetting the

pointer by 1 (using `b=a+1`), so that `b[1]...b[N]` points to the same locations as `a[0]...a[N-1]`. This convenient offsetting is demonstrated in the next example.

We can also allocate memory on the fly for more complex objects such as matrices. A good way to think of an $M \times N$ matrix with typical entry `q[m][n]` is that it consists of M vectors, each of length N . We can allocate memory for *a pointer (q) to an array of pointers* (`q[1]...q[M]`). Each of the pointers `q[1]...q[M]` is just like the pointer `a` in the previous example, pointing to its own personal chunk of memory of size N . This way of handling matrices is demonstrated in several examples on the website.

You can pass arrays to functions only by reference, not by value.

18 Modularizing

In section 12, we introduced the idea of chopping a program into small chunks called functions. It's good style to put each conceptual bit of the program in its own function. Try to avoid writing the same piece of code more than once.

In accordance with this principle of writing everything once only, it's also a good idea to put any function that you might want to use again into a file that other programs can make use of. There's a couple of ways to split programs over multiple files.

1. The simple way with `#include`. If you put the directive `#include "blah.cc"` on one line of a C++ program then the compiler will behave exactly as if the contents of the file `blah.cc` were there, in your file, at that point. In this way you need to compile only one file – the other files get read in appropriately.
2. The professional way with *linking*. Alternatively, you can split the program into multiple files *each of which is compiled separately*. In this approach, the compiler needs to be run several times, once for each separate file, and then a final time to *link* the compiled files together. *Compiling* each individual `.cc` file creates a corresponding `.o` file. *Linking* takes all the `.o` files and combines them into a single executable. Just one of the `.cc` files should contain a definition of the `main` function, which is where the executable starts.

When compiling an individual file, the compiler doesn't need to know anything about the functions in the other files, except for the syntax of any functions that get used in the current file. That syntax is conveyed to the compiler by function declarations. The recommended technique to handle these function declarations is to ensure that all functions in the file `blah.cc` are *declared* in another file called `blah.h`, which is `#included` by `blah.cc` and by any other files that wish to use `blah.cc`'s functions.

This 'linking' technique is illustrated by the four files that follow. The main program is in `package1.cc`. This program uses utility functions for memory allocation and array printing, located in the file `tinyutils.cc`. Both `package1.cc` and `tinyutils.cc` include the header file `tinyutils.h`. And finally, to keep track of what needs compiling when, it is essential to use a `Makefile`. The `Makefile` must contain an explicit statement that "`package1` depends on `package1.o` and `tinyutils.o`", and (on the following line, *which must start with a single tab character*) an explicit instruction of how to link them together.

```
// tinyutils.h
//   declares the functions defined in tinyutils.cc

using namespace std;

// memory management
double *dvector ( int low, int high ) ;
void   freedvector ( double* a , int low ) ;

// printing
void printVector( double *b , int lo , int hi , int style=1 ) ;
// Note that any default parameter values (such as 'style=1')
// must be specified in the declaration.

// maths
double square( double a ) ;
```

Note that in the makefile below, the indentation **must** be done with a tab character,(not with spaces and not ignored) or the makefile will not work.

```
# Makefile for package1 and package2

CFLAGS = -g -Wall

LIBS = -l stdc++ -lm

CXX = g++

# These lines define what package1 depends on, and how to make it
package1: package1.o tinyutils.o
    $(CXX) $(CFLAGS) $(LIBS) package1.o tinyutils.o -o package1
package2: package2.o tinyutils2.o
    $(CXX) $(CFLAGS) $(LIBS) package2.o tinyutils2.o -o package2

%.o: %.cc
    $(CXX) $(CFLAGS)      $< -c -o $@

%: %.cc Makefile
    $(CXX) $(CFLAGS) $(LIBS) $< -o $@
```



```

// package1.cc
// demonstrates how to use functions
// defined in a separately-compiled file (tinyutils.cc)

#include <iostream>
using namespace std;

// Both this file and tinyutils.cc include the
// function declarations from a single header file:
#include "tinyutils.h"

// In this example, we use functions 'dvector',
// 'square', 'printVector', and 'freedvector', all defined in
// tinyutils.cc

int main()
{
    double *b , *a ;
    int     N = 20 ;

    // allocate the space for b[1]..b[N]
    b = dvector( 1 , N ) ;
    a = dvector( 1 , N ) ;

    for ( int n = 1 ; n <= N ; n ++ )
        a[n] = static_cast<double>(n) ;
    for ( int m = 1 ; m <= N ; m ++ )
        b[m] = square( a[m] ) ;
    printVector( b , 1 , N ) ;

    // free the memory
    freedvector( b , 1 ) ;
    freedvector( a , 1 ) ;
    return 0;
}

```

```

// tinyutils.cc
// provides functions for double vectors allocation and clean-up
#include <iostream>
using namespace std;
#include "tinyutils.h"

// allocates memory for an array of doubles, say b[low]..b[high]
// example usage:  b = dvector( 1 , N ) ;
double *dvector ( int low, int high ) {
    int N = high-low+1 ;
    double *a ;
    if ( N <= 0 ) {
        cerr << "Illegal range in dvector: "
              << low << ", " << high << endl ;
        return 0 ; // returns zero on failure.
    }
    a = new double[N] ; // allocates memory for a[0]..a[N-1]
    if(!a) {
        cerr << "Memory allocation failed\n" ;
        return 0 ; // returns zero on failure.
    }
    return (a-low) ; // offset the pointer by low.
} // the user uses b[low]..b[high]

void freedvector ( double *b , int low ) {
    delete [] &(b[low]) ; // The '[]' indicate that what's
} // being freed is an array

// Note that default parameter values (such as 'style=0') have already
// been specified in the function declaration in tinyutils.h.
void printVector( double * b , int lo , int hi , int style ) {
    // style 1 means "all on one line"; style 0 means "in one column"
    for ( int n = lo ; n <= hi ; n ++ ) {
        cout << b[n] ;
        if(style) {
            if ( n == hi )    cout << endl;
            else              cout << "\t" ;
        } else {
            cout << endl;
        }
    }
}

double square( double x ) {
    return x*x ;
}

```

When we type `make package1`, the following things happen.

1. `make` looks at the Makefile and learns that `package1` depends on `package1.o` and `tinyutils.o`.
2. If `package1.o` needs to be made, `make` invokes

```
g++ -pedantic -g -Wall package1.cc -c -o package1.o
```

At this stage, the compiler compiles just the functions that are defined in `package1.cc`.

3. Similarly for `tinyutils.o`, `make` invokes

```
g++ -pedantic -g -Wall tinyutils.cc -c -o tinyutils.o
```

4. For the final linking step, `make` invokes

```
g++ -pedantic -g -Wall -l stdc++ -lm package1.o tinyutils.o -o package1
```

yielding the executable `package1`. It's at this stage that the compiler will complain if any functions have been declared but not defined.

19 Structures and packages

We described, last term, how to use structures. Structures are a great way to organize things that belong together, and that should never really be separated from each other, such as a vector and its index range. By putting such things together into a single object, we can make code briefer (because we just refer to the one object, rather than its parts), and less buggy. The next example shows how to rewrite the previous example's vector-creation and vector-printing using a structure that contains the vector's pointer and its index range. The structure is defined in the header file. Why is it a good idea to use this structure? For this toy example, it doesn't seem like a big deal, but what you can notice is that function-calls that do things with the vector, once it's been created (such as `printDvector(b)`), are simpler and briefer, because we don't need to send along the baggage (`low`, `high`) that is required in the un-structured approach. The structure contains this baggage, so when we pass the pointer to the structure to other functions, those functions get access to exactly the baggage they need. The only down-side of this structured approach is that when we want to access the contents of the vector, we have to get the pointer to the vector out of the structure, so what used to be `a[n]` in the old approach (where `a` was the pointer to the array) becomes `a.v[n]` in the new approach (where `a` is the structure).

```

// package2.cc
// demonstrates how to use structures and functions
// defined in a separately-compiled file (tinyutils2.cc).
// The structure Dvector is defined in tinyutils2.h

#include <iostream>
using namespace std;

// Both this file and tinyutils2.cc include the
// function declarations from a single header file:
#include "tinyutils2.h"

int main()
{
    Dvector a , b ;
    int     N = 20 ;

    // allocate the space for b.v[1]..b.v[N]
    allocate( b , 1 , N ) ;
    allocate( a , 1 , N ) ;

    for ( int n = 1 ; n <= N ; n ++ )
        a.v[n] = static_cast<double>(n) ;
    for ( int m = 1 ; m <= N ; m ++ )
        b.v[m] = square( a.v[m] ) ;
    printDvector( b ) ;

    // free the memory
    freeDvector( b ) ;
    freeDvector( a ) ;
    return 0;
}

```

```

// tinyutils2.h
//   declares the structures and functions defined in tinyutils2.cc

using namespace std;

struct Dvector {
    double *v ; // the vector itself
    int     low;
    int     high;
} ; // don't forget the semicolon in the structure definition

// memory management
int     allocate( Dvector &a , int low, int high ) ;
void    freeDvector ( Dvector &a );

// printing
void printDvector( Dvector &b , int style=0 ) ;
// Default parameter values (such as 'style=0')
// must be specified in the declaration.

// maths
double square( double a ) ;

```

```

// tinyutils2.cc
// provides functions for double vectors allocation and clean-up
#include <iostream>
using namespace std;
#include "tinyutils2.h"

// allocates memory for an array of doubles. Example: allocate( b, 1 , N ) ;
int allocate ( Dvector &a , int low, int high ) {
    a.low = low ; a.high = high ;
    int N = high-low+1 ;
    if ( N <= 0 ) {
        cerr << "Illegal range in dvector: "
              << low << ", " << high << endl ;
        return 0 ; // returns zero if failure.
    }
    a.v = new double[N] ; // allocates memory for a[0]..a[N-1]
    if(!a.v) {
        cerr << "Memory allocation failed\n" ;
        return 0 ;
    } else {
        a.v -= low ; // offset the pointer by low.
        return 1 ;
    }
}

void freeDvector ( Dvector &b ) {
    delete [] &(b.v[b.low]) ;
    b.high = b.low - 1 ;
}

// Note that default parameter values (such as 'style=0') have already
// been specified in the function declaration in tinyutils2.h.
void printDvector( Dvector &b , int style ) {
    // style 1 means "all on one line"; style 0 means "in one column"
    for ( int n = b.low ; n <= b.high ; n ++ ) {
        cout << b.v[n] ;
        if(style) {
            if ( n == b.high )      cout << endl;
            else                    cout << "\t" ;
        } else {
            cout << endl;
        }
    }
}

double square( double x ) {
    return x*x ;
}

```

20 Formatted output

We've mainly printed out numbers using `cout` with commands like

```
cout << myint << " " << mydouble << endl ;
```

What if you don't like the way `cout` makes your numbers look, however? Too many decimal places? Too few? Well, `cout` can be bossed around and can be told to serve up numbers with different numbers of decimal places. You can find out more about `cout` by looking in a manual online or on paper.

Here we'll describe another way to control output formatting, using the old-fashioned C function, `printf` (which means `print`, formatted). It's probably worth learning a bit about `printf` because its syntax is used in quite a few languages.

The commands

```
int age = 84; printf( "his age is %d years" , age ) ;
```

will print the string 'his age is 84 years'. The command

```
printf( "%d %d\n" , i , j ) ;
```

causes the integers `i` and `j` to be printed, separated by a single space, and followed by a newline (just like `cout << i << " " << j << endl ;`). The command

```
printf( "%3d %-3d\n" , i , j ) ;
```

causes the same numbers to be printed out but encourages `i` to take up 3 columns, and to be right-aligned within those 3 columns; and encourages `j` to take up 3 columns and to be left-aligned. Text and numbers can be mixed up. For example,

```
printf( "from %3d to %-3d\n" , i , j ) ;
```

might be rendered as

```
from 123 to 789
```

Notice how you specify the format of the whole string first, then the missing bits, each of which was indicated in the format string by a specifier, *%something*. The format specifier for a single character is `%c`; for a string of characters, `%s`. The special character `\n` is a newline; `\t` is a tab; `\"` gives a double quote; `\\` gives a single backslash. The command

```
printf( "%f %e %g \n" , x , y , z ) ;
```

prints the floating point numbers `x`, `y`, and `z` as follows: `x` is printed as a floating point number with a default number of digits (six) shown after the decimal point; `y` is printed in scientific (exponential) notation; `z` is printed using whichever makes more sense, fixed floating point or scientific notation. The program `Printf.cc` on the website illustrates more examples. I usually use the format

```
printf( "%12.6g %12.6g %12.6g" , x , y , z ) ;
```

to print my real numbers – this format gives each of them 12 columns, and shows 6 digits of precision.

21 Notes concerning the remaining Assessments 5 and 6.

The aim of the remaining Assessments will be not only to learn more about C++ but also to see how programming can help you understand physics better.

This final two assessed tasks are harder than last term's. It's essential to prepare before each week's programming exercises. Read this manual and sketch out a plan of what you are going to do *before* sitting down at the computer. The exercises involve physics, so you will need to prepare both physics thoughts and computing thoughts. **Please allow two hours preparation time per week.**

Part IB Computing Course

ASSESSMENT 5 : “PLANET”

22 Instructions

22.1 Objectives

Physics objectives: to better understand Newton’s laws, circular motion, angular momentum, planets’ orbits, Rutherford scattering, and questions about spacemen throwing tools while floating near space shuttles.

Computing objectives: structures, arrays, using gnuplot; simulation methods for differential equations (Euler, Leapfrog).

Remember, you are welcome to work with others, with the weaker programmer doing the majority of the typing, *etc.*

22.2 Task

Simulate Newton’s laws for a small planet moving near a massive sun. For ease of plotting, assume the planet and its velocity both lie in a two-dimensional plane. Put the sun at the origin $(0, 0)$ and let the planet have mass m and initial location $\mathbf{x}^{(0)} = (x_1, x_2)$ and initial velocity $\mathbf{v}^{(0)} = (v_1, v_2)$. The equations of motion are:

$$\begin{aligned}\frac{d\mathbf{x}}{dt} &= \mathbf{v}(t) \\ m\frac{d\mathbf{v}}{dt} &= \mathbf{f}(\mathbf{x}, t),\end{aligned}\tag{1}$$

where, for a standard inverse-square law (gravity or electrodynamics) the force \mathbf{f} is:

$$\mathbf{f}(\mathbf{x}, t) = A \frac{\mathbf{x}}{\left(\sqrt{\sum_i x_i^2}\right)^3},\tag{2}$$

with $A = -GMm$ for gravitation, and $A = Qq/(4\pi\epsilon_0)$ for electrodynamics with two charges Q and q .

Try to write your programs in such a way that it’ll be easy to switch the force law from inverse-square to other force laws. For example, Hooke’s law says:

$$\mathbf{f}(\mathbf{x}, t) = k\mathbf{x}.\tag{3}$$

How should we simulate Newton’s laws (1) on a computer? One simple method called Euler’s method makes *small, simultaneous* steps in \mathbf{x} and \mathbf{v} . We repeat lots of times the following block:

EULER’S METHOD

- 1: Find $\mathbf{f} = \mathbf{f}(\mathbf{x}, t)$
- 2: Increment \mathbf{x} by $\delta t \times \mathbf{v}$
- 3: Increment \mathbf{v} by $\delta t \times \frac{1}{m}\mathbf{f}$
- 4: Increment t by δt

We might hope that, for sufficiently small δt , the resulting state sequence (\mathbf{x}, \mathbf{v}) in the computer would be close to the true solution of the differential equations.

Euler's method is not the only way to approximate the equations of motion.

An equally simple algorithm can be obtained by reordering the updates. In the following block, the updates of \mathbf{f} and \mathbf{x} have been exchanged, so the force is evaluated at the new location, rather than the old.

```
A: Increment  $\mathbf{x}$  by  $\delta t \times \mathbf{v}$ 
B: Find  $\mathbf{f} = \mathbf{f}(\mathbf{x}, t)$ 
3: Increment  $\mathbf{v}$  by  $\delta t \times \frac{1}{m}\mathbf{f}$ 
4: Increment  $t$  by  $\delta t$ 
```

It's not at all obvious that this minor change should make much difference, but often it does. This second method, in which position and velocity are updated alternately, is called the Leapfrog method. Here is a precise statement of the Leapfrog method.

```
LEAPFROG METHOD
Repeat
{
    Increment  $\mathbf{x}$  by  $\frac{1}{2}\delta t \times \mathbf{v}$ 
    Increment  $t$  by  $\frac{1}{2}\delta t$ 
    Find  $\mathbf{f} = \mathbf{f}(\mathbf{x}, t)$ 
    Increment  $\mathbf{v}$  by  $\delta t \times \frac{1}{m}\mathbf{f}$ 
    Increment  $\mathbf{x}$  by  $\frac{1}{2}\delta t \times \mathbf{v}$ 
    Increment  $t$  by  $\frac{1}{2}\delta t$ 
}
```

In this version, we make a half-step of \mathbf{x} , a full-step of \mathbf{v} , and a half-step of \mathbf{x} . Since the end of every iteration except the last is followed by the beginning of the next iteration, this algorithm with its half-steps is identical to the previous version, except at the first and last iterations.

When simulating a system like this, there are some obvious quantities you should look at: the angular momentum, the kinetic energy, the potential energy, and the total energy.

Techniques to use:

1. Represent the vectors \mathbf{x} , \mathbf{v} , and \mathbf{f} using **arrays**. Make sure you look at some simple examples of arrays before using them for planets.
2. Put all the variables and arrays associated with a single object (such as a planet) in a **structure**. Make sure you look at some simple examples of structures.

22.3 Ideas for what to do

This is a self-directed and self-assessed course, and I'd like you to choose a planet-simulating activity that interests you. Here are some suggestions. **You don't have to do all of these. You can also invent your own.** The more you do, the more educational it'll be. But do take your pick, and feel free to steal working code (e.g. from the course website) if you'd prefer to focus your energy on experimenting with working code, rather than writing and debugging your own.

1. Write code that implements Euler's method and the Leapfrog method. Get it going for one initial condition, such as $\mathbf{x}^{(0)} = (1.5, 2)$, $\mathbf{v}^{(0)} = (0.4, 0)$. Before running your program, predict the correct motion, roughly. Compare the two methods, using **gnuplot** to look at the

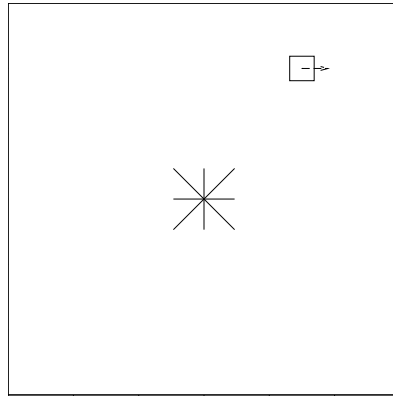


Figure 10: The initial condition $\mathbf{x}^{(0)} = (1.5, 2)$, $\mathbf{v}^{(0)} = (0.4, 0)$.

resulting trajectories, the energies, and the angular momenta. (If you'd like more detailed guidance through a possible approach to this task, **see the appendix on p. 113**. A worked solution for this first part is also available.)

2. Once you have a trustworthy simulation: take a collection of similar initial conditions, all having the same initial position and initial energy but differing in the direction of the initial velocity. What happens? Show all the evolving trajectories in a single movie.
3. Or take initial conditions that differ slightly in their initial position or velocity, such as “the spaceshuttle and the spaceman who is 100 yards from the spaceshuttle, both orbitting the earth” (in what direction does he need to nudge himself in order to get home?). Or “the spaceman and the hammer” – if he throws his hammer ‘forwards’, where does it end up going? If he throws it ‘back’ or ‘sideways’, where does it end up going? (Here the idea is to simulate two planets orbitting a single sun; to get the spaceman analogy, think of the spaceman and his shuttle being like the two planets, and the earth playing the role of the sun.)
4. Get an initial condition that leads to a perfectly circular orbit. Now perturb that initial condition by giving a little kick of momentum in 8 different directions. Show all nine resulting trajectories in a single movie. Which kicks lead to the period of the orbit changing? Which kicks lead to the period of the orbit staying the same? Which kicks lead to orbits that come back to the place where the kick occurred? If an object is in an elliptical orbit, what kicks do you give it in order to make the orbit larger and circular? What kicks do you give it to make the orbit smaller and circular? What's the best time, in an elliptical orbit, to give the particle a kick so as to get the particle onto an orbit that never comes back, assuming we want the momentum in the kick to be as small as possible?
5. Take initial conditions coming in from a long way away, particles travelling along equally-spaced parallel lines. What happens? (The distance of the initial line from the sun is called the impact parameter of the initial condition.)
6. People who criticise Euler's method often recommend the *Runge–Kutta method*. Find out what Runge–Kutta is (Google knows), implement it, and compare with the Leapfrog method. Choose a big enough step-size δt so that you can see a difference between the methods. Given equal numbers of computational operations, does Runge–Kutta or Leapfrog do a

better job of making an accurate trajectory? For very long runs, and again assuming equal numbers of computational operations, does Runge–Kutta or Leapfrog do a better job of energy conservation? Of angular momentum conservation?

22.4 What to hand in

This SESSION has a slightly different than SESSIONS 1 to 4 as the emphasis is now on using computing to help you learn about physics, rather than just learning to program. **This means that there is greater importance attached to the “Brief Description” part of the submission than in the earlier sessions, and consequently it will be longer than in the earlier sessions, and will contain a greater discussion of physics directed goals than of the computing ones.**

You will be expected to use the “brief description” part of the web-based submission form to explain not only the technical/computational challenges which you faced, but also to devote a greater than normal time to explaining the *physics goals* which you set yourself (perhaps based on the suggestions of section 22.3) and what use of your program helped you learn about them. This additional part of the “Brief Description” is the part you must ‘engage with’ if you are to gain the extra credit for this section.

As usual, you will gain credit for uploading the programs you used to solve the problems, but here it is possible (though not required) that you may have used gnuplot, or other tools, and may decide to hand in evidence of them too. The choice is yours. In all cases, explain what your submissions allowed you to do or to test.

As the description will be longer than normal, and to avoid having to curse your browser if you hit the “back button” inadvertently, you are reminded to create your brief description in a separate README file (saving as you compose) before pasting it into the submission form as the final step. The form itself is crude and will not remember text if you happen to hit the back button or visit another page seconds before pressing “submit” !

END OF ASSESSMENT 5 : “PLANET”

Part IB Computing Course

ASSESSMENT 6 : “BONKERS”

23 Instructions

23.1 Objectives

Physics objectives: to better understand collisions, conservation laws, and statistical physics, especially equipartition, properties of ideal gases, the concept of temperature, and the Boltzmann distribution; also the way in which microscopic physics leads to macroscopic phenomena; what happens when a piston compresses an ideal gas; adiabatic expansion; fluctuations and dissipation, equilibration of systems with different temperatures.

Computing objectives: structures, arrays, using gnuplot; memory allocation.

You are encouraged to work in pairs, with the weaker programmer doing all the typing.

23.2 Task

We’re going to simulate hard spheres colliding with each other elastically in a box. An incredible range of interesting physics phenomena can be studied in this way.

The heart of this computing exercise is going to be a single function – let’s call it `collide` – which receives two particles with masses m_1 and m_2 and velocities u_1 and u_2 , and returns the new velocities v_1 and v_2 of the two particles after an elastic collision between the two.

You could start by writing a function that implements this elastic collision. Check your function with special cases that you’ve solved by hand, and by putting in a range of initial conditions and evaluating whether total momentum and total energy are indeed conserved as they should be.

An elegant approach to this programming task uses a structure to represent each particle – something like this, for a particle moving in one dimension:

```
struct particle {
    double x      ; // position
    double p      ; // momentum
    double im     ; // inverse mass
    double v      ; // velocity
    double T      ; // kinetic energy
    double a      ; // radius of particle
} ; // Note the definition of a structure ends with a semicolon
```

At this stage it’s not crucial, but at some point I recommend making sure all references to the particle’s mass use the *inverse mass*, rather than the mass – this allows you to treat the walls of the box as standard particles that just happen to have infinite mass (that is, inverse-mass zero).

Once you have defined a `struct` like `particle`, you may define an array of `particles` in just the same way that you define arrays of other objects like `ints` or `doubles`. For example

```
a = new particle[N] ;
```

23.3 Ideas for what to do

There's a lot of choice. This is a self-directed and self-assessed course, and I'd like you to choose a bonking-simulating activity that interests you.

Here are some suggestions. **You don't have to do all of these. You can also invent your own.** The more you do, the more educational it'll be. But do take your pick, and feel free to steal working code (e.g. from the course website) if you'd prefer to focus your energy on experimenting with working code, rather than writing and debugging your own.

1. Write code that uses a one-dimensional `collide` function to simulate the motion of N particles in a one-dimensional box. Each particle can collide only with its immediate neighbours; each of the two end particles can collide with one wall too. To simulate the dynamics, you must identify the times at which collisions occur, and (assuming you want to make a realistic movie for showing in gnuplot) spit out the state of the simulation *at equally spaced times*. A suggested strategy is to take the current state and figure out which pair of adjacent particles will collide *next*. Then advance the simulation exactly to the moment of that next collision (stopping if appropriate at intermediate times along the way, so as to spit out the required states for the movie, equally spaced in time). Collisions should be handled by passing the pair of particles to the `collide` function. Motion in between collisions is simple (since there are no forces) and it should be handled by another function, `leapForward`, say. Printing out of the state at certain times of interest should be handled by another function, `showState`, say. (A worked solution for this first part is available on the course website.)

[A possible difficulty with this approach of computing all collisions that occur is that it is conceivable that the true number of collisions in a finite time might be very large, a phenomenon known as chattering. You can get the idea of chattering by imagining quickly squashing a moving ping-pong ball between a table-tennis bat and a table.]

2. Testing: Put just two particles in a box, with equal masses. Check that the dynamics are right. Make the two masses unequal. Make a scatter-plot of the positions of the two particles. Make a scatter-plot of the velocities of the two particles. Use more masses. Check that kinetic energy is conserved.
3. Put quite a few *unequal* masses in the box (say, 10 masses, with a variety of masses spanning a range of a factor of 4 in magnitude), run the simulation for a long time, and make histograms of the velocities of two of the particles whose masses are in the ratio 4:1. What do you find? Make histograms of the *positions* of the particles. If you make some of the particles *really heavy* compared to their neighbours, what happens to the histogram of the positions of the neighbours? For example, make all the particles except for the two end particles be much larger; or make half the particles (those on the left hand side) heavy, and the other half light. (In all simulations make sure no two adjacent particles have identical masses.)
4. What happens if the right-hand wall (with infinite mass) is moved at constant speed towards or away from the other wall?

Ye have heard it said that, under some circumstances, $pV^\gamma = \text{constant}$. What is γ for a *one-dimensional* ideal gas? How should the total energy vary with V under these conditions?

5. Set up N_1 light masses to the left of a single big heavy mass, and N_2 more light masses to the right of the heavy mass. Call the heavy mass a piston, if you like, and think of it as separating two ideal gases from each other. The light masses don't need to be identical to each other. An example set of masses for $N_1 = N_2 = 5$ could be (1.1, 1.2, 1.1, 1.3, 1.1, 100.0, 4.1, 4.2, 4.1, 4.8, 4.4) where the 100-mass is the piston. Give randomly chosen velocities to the particles. What should happen? How long does it take for 'equilibrium' to be reached? Give an enormous velocity to the piston and small velocities to the other particles. What should happen? How long does it take for 'equilibrium' to be reached? Can you get the piston to oscillate roughly sinusoidally (before 'equilibrium' is reached)? What is the frequency of such oscillations? Predict the frequency using the theory of adiabatic expansion/compression of gases.

The first six collisions
between two particles of
masses 3 and 1 and two
walls.

23.4 What to hand in

This SESSION has a slightly different than SESSIONS 1 to 4 as the emphasis is now on using computing to help you learn about physics, rather than just learning to program. **This means that there is greater importance attached to the “Brief Description” part of the submission than in the earlier sessions, and consequently it will be longer than in the earlier sessions, and will contain a greater discussion of physics directed goals than of the computing ones.**

You will be expected to use the “brief description” part of the web-based submission form to explain not only the technical/computational challenges which you faced, but also to devote a greater than normal time to explaining the *physics goals* which you set yourself (perhaps based on the suggestions of section 23.3) and what use of your program helped you learn about them.

This additional part of the “Brief Description” is the part you must ‘engage with’ if you are to gain the extra credit for this section.

As usual, you will gain credit for uploading the programs you used to solve the problems, but here it is possible (though not required) that you may have used gnuplot, or other tools, and may decide to hand in evidence of them too. The choice is yours. In all cases, explain what your submissions allowed you to do or to test.

As the description will be longer than normal, and to avoid having to curse your browser if you hit the “back button” inadvertently, you are reminded to create your brief description in a separate README file (saving as you compose) before pasting it into the submission form as the final step. The form itself is crude and will not remember text if you happen to hit the back button or visit another page seconds before pressing “submit” !

END OF ASSESSMENT 6 : “BONKERS”

Part IB Computing Course

ASSESSMENT 7 : “RECURSION” (optional!)

24 Instructions

24.1 Objectives

This is an optional extra task idea. Only six tasks can be submitted for assessment, but there are seven weeks of demonstrators you can make use of if you want to ...

Physics objectives: to better understand statistical physics by counting some interesting things.

Computing objectives: recursion.

Recursion means defining a function f in terms of the function f . For example we could define the factorial function $f(x)$ by:

1. if $x > 1$ then return $x \times f(x - 1)$
2. otherwise return 1.

A recursive function in a computer program is one that calls itself. Here's an example, following the above definition closely.

```
// factorial calculator - recursive
#include <iostream>
using namespace std;

int factorial (int a)
{
    if (a > 1)
        return (a * factorial (a-1));
    else
        return (1);
}

int main ()
{
    int number;
    cout << "Please type a number: ";
    cin >> number;
    cout << number << "! = " << factorial(number) << endl;
    return 0;
}
```

Errors in the definition of recursive functions often lead to disaster, since it's all too easy to write a function that keeps calling itself and never stops.

Some programmers find recursion an elegant way to express many programming tasks. Here is an example. The task is ‘print all ternary strings of length L ’. For $L = 1$ the output should be

0, 1, 2.

For $L = 2$ the output should be

00, 01, 02, 10, 11, 12, 20, 21, 22

Here is a solution.

```
// allStrings.cc
// Enumerate all ternary strings by recursion

#include <iostream>
using namespace std;

void appendAllStrings( char *prefix , int remainingLength )
{
    if (remainingLength == 0)
        cout << prefix << endl ;
    else {
        int lp = strlen(prefix) ;
        for ( int i = 0 ; i <= 2 ; i ++ ) { // Extend prefix by one character.
            prefix[lp] = i+'0' ;           // By adding i to the character '0'
                                           // we get the characters '0', '1', '2'
            appendAllStrings( prefix , remainingLength-1 );
        }
        prefix[lp] = '\0' ;                // Remove what was added
                                           // [ '\0' is the null character ]
    }
    return ;
}

int main ()
{
    int length;
    cout << "Please type a length: ";
    cin >> length;
    char *prefix ;
    prefix = new char[length] ;           // Assume this memory is all-null
    appendAllStrings( prefix , length ) ;
    delete [] prefix ;                    // Free the memory
    return 0;
}
```

The function `strlen` returns the length of the string generated so far; the line

`prefix[lp] = ...`

extends the string by one character. The algorithm proceeds by starting from the null string, and extending it by all possible single characters, extending each in turn by all possible characters, and so forth.

Such ‘branching processes’ are the type of problem for which recursion is especially recommended.

24.2 Recursion exercises

Solve each of these tasks using a recursive function.

1. Write a function called `twos` that takes a single integer as its argument and returns the number of factors of 2 in the number. (Hint: odd numbers have no factors of 2, numbers that are twice an odd number have one, numbers that are four times an odd have two, and so on.) For example: `twos(-12);` should return 2.
2. Write a function called `printWithCommas` that takes a single nonnegative long integer argument and displays it with commas inserted in the conventional way. For example:
 - `printWithCommas(12045670);` displays 12,045,670.
 - `printWithCommas(1);` displays 1.
3. INTERESTING PROPERTIES OF A HARD SPHERE GAS.

A legal state of four particles in a 10×10 box.

A square box of size $H \times H$ lattice-points contains one big particle and $T = 3$ little particles. The big particle occupies 4×4 lattice points. Each little particle occupies 2×2 lattice points. Particles may not overlap. All legal states of this $T + 1$ -particle system are equiprobable. How probable are the alternative locations of the big particle? Find the answer for $H = 10$ (by recursively enumerating all legal states, and keeping count). It is a good idea to spit out the answer for smaller values of T along the way towards the answer for the biggest value of T .

If not all locations of the big particle are equiprobable then we can describe the effect of the little particles in terms of an effective ‘force’ acting on the big particle. Such forces are called ‘entropic forces’. Entropic forces in hard-sphere mixtures with a variety of sizes of spheres are an area of industrial research interest.²⁴

END OF ASSESSMENT 7 : “RECURSION” (optional!)

²⁴Y. Mao, P. Bladon, H. N. W. Lekkerkerker, and M. E. Cates. *Mol. Phys.* **92**, 151 (1997).

R. Dickman, P. Attard, and V. Simonian. ‘Entropic Forces in Binary Hard-Sphere Mixtures: Theory and Simulation’, *J. Chem. Phys.* **107**, 205–213 (1997).

‘Entropic Attraction and Repulsion in Binary Colloids Probed with a Line Optical Tweezer’ J. C. Crocker, J. A. Matteo, A. D. Dinsmore, and A. G. Yodh. *Physical Review Letters*, **82**, 4352–4355 (1999).

Appendices

A PLANET: step by step guidance

1. Chop the problem into small pieces.
2. Define a structure that contains the state of the planet. (See page 95 of this tutorial and page 60 of the first term's tutorial.) Your structure could be as simple as:

```
struct particle {  
    double x[2] ; // (x,y) coordinates  
    double v[2] ; // velocity  
} ;
```

but as you continue, you will probably think of other sensible things to add to the structure. You might find it elegant to give a name to the dimensionality of the position space, say D.

```
#define D 2 // number of dimensions  
struct particle {  
    double x[D] ; // (x,y) coordinates  
    double v[D] ; // velocity  
} ; // Note the definition of a structure ends with a semicolon
```

If you then use D everywhere, it makes the meaning of your code clearer (compared with using '2'), and it makes it easier to update your simulation's dimension (say from 2 to 3 dimensions).

3. Write a simple `main` that defines a particle, and check that it compiles and runs.

```
int main()  
{  
    particle a ;  
  
    a.v[0] = 0.4;  
    a.v[1] = 0.0;  
    a.x[0] = 1.5;  
    a.x[1] = 2;  
  
    return 0;  
}
```

4. Write a function `showState` that prints out the particle's position and velocity. Call it from `main` and check that your program compiles and runs correctly. Here is an example of a function that does the right sort of thing:

```

void showState ( particle &a )
{
    int i=0;
    cout << "some of the state: " << a.x[i] << endl ;
}

```

This could be called by `main` using a command such as

```
showState( a ) ;
```

Notice the use of the ampersand in the function definition

```
void showState ( particle &a ).
```

This ampersand means that we are passing the particle by reference, rather than by value. (See page 45 of last term's tutorial.)

5. Remember the tips from last term about debugging your code. (1) Use a **makefile**. (2) Switch on all the compiler warnings. (3) Whenever you think that your code ought to compile, check that it does so. (4) If you have code that compiles ok, but that doesn't behave as expected when you run it, *run it in a debugger* (for example, **kdbg**). Even if it seems to be running as expected, it might be a good idea to run it in a debugger. When I use **kdbg**, the main things I click on are 'set breakpoint' (right click), 'run', and the 'step over' icon.
6. Write a function that computes the squared-distance from the origin.
7. Write a function **Force** that computes the force acting on the particle, or the acceleration of the particle. (Perhaps the force or acceleration should be in your structure?)
8. Write a function **PositionStep** that makes a change of a particle's position in the direction of its velocity. One of the arguments of this function should be the required time step size.
9. Write a function **VelocityStep** that makes a change of a particle's velocity in the direction of its acceleration.
10. Write a function that computes the energy (kinetic and potential) of the particle.
11. Write a function that computes the angular momentum about the origin.
12. Write a Leapfrog simulator with a loop that calls the above functions appropriately.
13. Write the simulated trajectory to a file using a function like **showState**. To plot the trajectory from a file whose columns are

time, x_1 , x_2 , v_1 , v_2 ,

the following gnuplot commands may be useful.

```
set size ratio -1 ; ## this makes units on the x and y axes have equal size
plot 'tmp' u 2:3 w l, 'tmp' every 10 u 2:3:4:5 w vector
## this plots (x1,x2) with lines, and plots every 10th state using a vector
## of length (v1,v2) based at the point (x1,x2)
```

gnuplot's `plot` command is very versatile. The website shows how you can make a gnuplot animation using a single data file containing a trajectory.

14. Be aware of the size of the files you are writing. If you give gnuplot a file of more than 1 megabyte, you should expect it to be sluggish. Maybe it would be a good idea to reduce the amount of information written to file. You may be able to get better performance by using the local filesystem of the machine at which you are sitting. You can write anything you want into the folder `/tmp`. For example,

```
ls > /tmp/mylist
```

The `/tmp` folder is a good place to put anything large and anything that you want to access frequently or quickly. But don't leave anything important in `/tmp` – after you log out, the `/tmp` folder may be cleaned up.

B An introduction to object-oriented programming and classes

One of the most important differences between the C++ programming language and some other programming languages is the emphasis on the representation of data using programmer or user-defined data types. In C++ an extension to the definition of structures allows the user to include both *data members* (as above) and *member functions* which are allowed to process the data. The encapsulation of data and functions into packages called *objects* of user-defined types called *classes* is a key part of object-oriented programming.

In its simplest form, a **class** is like a structure that includes the definition of functions (member functions or class methods) that are allowed to process the data. See the simple example below.

1. Classes and Objects

The class (e.g. **Date** in the example) can be considered as a specification while the actual item (e.g. **today**) is an instance of a class and is called an *object*. Declaring an object is very similar to declaring a variable:

```
class-name object-name ;
```

2. Accessing data members and functions

The data members and member functions of the class can be accessed by simply naming them in conjunction with the name of the object they belong to using the *dot operator*.

```
object-name.item-name
```

3. Defining a class member function

The declaration and definition of class member functions is similar to those used for standard functions: we use function prototypes to declare the functions and place the statements to be executed in a function definition. The only difference is that with a class member function we must tell the compiler which class the function is a member of. This is done by including the name of the class in the function header using a *scope resolution* operator represented by `::`.

```
return-type class-name::function-name(parameter-list )
```

4. Restricting access to class members

One of three levels of access can be specified for each class member (both data and functions) using the keywords **public**, **private** or **protected** followed by a colon. These refer to how the class members may be accessed in a program.

The **public** members of a class may be accessed directly from anywhere in the program that has access to an object of the class. The **private** members can be read or modified only by class member functions.

B.0.1 A simple example

This example shows how to create a simple class and define the class's member functions.

```
// SimpleClass.cc
// A program to demonstrate creation and use of a simple class for dates
#include <iostream>
using namespace std;

// Declaration of Date class
class Date {
public:
    void set(int, int, int);
    void print();

private:
    int year;
    int month;
    int day;
};

int main()
{
    // Create a Date object called today
    Date today;

    // Call Date member function set()
    today.set(1,9,1999);
    cout << "This program was written on ";
    today.print();
    cout << endl;
    return 0;
}

// Date member function definitions
void Date::set(int d, int m, int y)
{
    if(d>0 && d<31) day = d;
    if(m>0 && m<13) month = m;
    if(y>0) year =y;
}

void Date::print()
{
    cout << day << "-" << month << "-" << year << endl;
}
```

C Further reading

This tutorial has introduced the basic elements of the C++ programming language. The following references provide a comprehensive treatment and useful tips on good programming practice.

1. *C++ How to Program (5th edition)*, Deitel, H.M and Deitel, P.J.
Prentice Hall, Englewood (NJ), 2005.
2. *Code Complete: A Practical Handbook of Software Construction*, McConnell, S.
Microsoft Press, Redmond (WA), 1993.
3. *C++ in plain English*, Brian Overland.
M&T Books, 1999.

D Objectives suggested for each week of assessment

D.1 Week 1

- Familiarisation with the teaching system and C++ development environment
- Edit, compile and execute a working program
- Become familiar with the compiler's messages

D.2 Week 2

- Declare and define variables and constants
- Assign values to variables and manipulate them in arithmetic expressions
- Write the value of variables to the screen
- Read in the value of variables from the keyboard
- Write simple programs that use loops.

D.3 Week 3

- Boolean expressions with relational operators
- Simple control structures for selection and repetition

D.4 Week 4

- Definition, declaration and calling of functions
- Passing values to and returning values from functions
- Math and system library functions

D.5 Week 5 : “PLANET”

Physics objectives: to better understand Newton's laws, circular motion, angular momentum, planets' orbits, Rutherford scattering, and questions about spacemen throwing tools while floating near space shuttles.

Computing objectives: structures, arrays, using gnuplot; simulation methods for differential equations (Euler, Leapfrog).

D.6 Week 6 : “BONKERS”

Physics objectives: to better understand collisions, conservation laws, and statistical physics, especially equipartition, properties of ideal gases, the concept of temperature, and the Boltzmann distribution; also the way in which microscopic physics leads to macroscopic phenomena; what happens when a piston compresses an ideal gas; adiabatic expansion; fluctuations and dissipation, equilibration of systems with different temperatures.

Computing objectives: structures, arrays, using gnuplot; memory allocation.

D.7 Optional week 7 : “RECURSION”

This is an optional extra task idea. Only six task can be submitted for assessment, but there are seven weeks of demonstrators you can make use of if you want to ...

Physics objectives: to better understand statistical physics by counting some interesting things.

Computing objectives: recursion.

E Thirty useful unix commands

Thirty useful unix commands

This guide, based on a University computing service leaflet, is intended to be as generally valid as possible, but there are many different versions of Unix available within the University, so if you find a command option behaving differently on your local machine you should consult the on-line manual page for that command. Some commands have numerous options and there is not enough space to detail them all here, so for fuller information on these commands use the relevant on-line manual page.

The names of commands are printed in **bold**, and the names of objects operated on by these commands (e.g. files, directories) are printed in *italics*.

Thirty useful unix commands – index

1. **cat** - display or concatenate files
2. **cd** - change directory
3. **chmod** - change the permissions on a file or directory
4. **cp** - copy a file
5. **date** - display the current date and time
6. **diff** - display differences between text files
7. **file** - determine the type of a file
8. **find** - find files of a specified name or type
9. **ftp** - file transfer program
10. **grep** - searches files for a specified string or expression
11. **gzip** - compress a file
12. **help** - display information about bash builtin commands
13. **info** - read online documentation
14. **kill** - kill a process
15. **lpr** - print out a file
16. **ls** - list names of files in a directory
17. **man** - display an on-line manual page
18. **mkdir** - make a directory
19. **more** - scan through a text file page by page

- 20. **mv** - move or rename files or directories
- 21. **nice** - change the priority at which a job is being run
- 22. **passwd** - change your password
- 23. **ps** - list processes
- 24. **pwd** - display the name of your current directory
- 25. **quota** - disk quota and usage
- 26. **rm** - remove files or directories
- 27. **rmdir** - remove a directory
- 28. **sort** - sort and collate lines
- 29. **ssh** - secure remote login program
- 30. **scp** - securely copy files between computers

1. **cat** – display or concatenate files

cat takes a copy of a file and sends it to the standard output (i.e., to be displayed on your terminal, unless redirected elsewhere), so it is generally used either to read files, or to string together copies of several files, writing the output to a new file.

cat *ex*

displays the contents of the file *ex*.

cat *ex1 ex2 > newex*

creates a new file *newex* containing copies of *ex1* and *ex2*, with the contents of *ex2* following the contents of *ex1*.

2. **cd** – change directory

cd is used to change from one directory to another.

cd *dir1*

changes directory so that *dir1* is your new current directory. *dir1* may be either the full pathname of the directory, or its pathname relative to the current directory.

cd

changes directory to your home directory.

cd ..

moves to the parent directory of your current directory.

3. **chmod** – change the permissions on a file or directory

chmod alters the permissions on files and directories using either symbolic or octal numeric codes. The symbolic codes are given here:-

u	user	+	to add a permission	r	read
g	group	-	to remove a permission	w	write
o	other	=	to assign a permission explicitly	x	execute (for files), access (for directories)

The following examples illustrate how these codes are used.

chmod u=rw file1

sets the permissions on the file *file1* to give the user read and write permission on *file1*. No other permissions are altered.

chmod u+x,g+w,o-r file2

alters the permissions on the file *file2* to give the user execute permission on *file2*, to give members of the user's group write permission on the file, and prevent any users not in this group from reading it.

chmod u+w,go-x dir1

gives the user write permission in the directory *dir1*, and prevents all other users having access to that directory (by using **cd**. They can still list its contents using **ls**.)

chmod g+s dir2

means that files and subdirectories in *dir2* are created with the group-ID of the parent directory, not that of the current process.

4. **cp** – copy a file

The command **cp** is used to make copies of files and directories.

cp *file1 file2*

copies the contents of the file *file1* into a new file called *file2*. **cp** cannot copy a file onto itself.

cp *file3 file4 dir1*

creates copies of *file3* and *file4* (with the same names), within the directory *dir1*. *dir1* must already exist for the copying to succeed.

cp -r *dir2 dir3*

recursively copies the directory *dir2*, together with its contents and subdirectories, to the directory *dir3*. If *dir3* does not already exist, it is created by **cp**, and the contents and subdirectories of *dir2* are recreated within it. If *dir3* does exist, a subdirectory called *dir2* is created within it, containing a copy of all the contents of the original *dir2*.

5. **date** – display the current date and time

date returns information on the current date and time in the format shown below:-

```
Wed Jan 9 14:35:45 GMT 2002
```

6. **diff** – display differences between text files

diff *file1 file2* reports line-by-line differences between the text files *file1* and *file2*. The default output will contain lines such as ‘*n1 a n2,n3*’ and ‘*n4,n5 c n6,n7*’, (where ‘*n1 a n2,n3*’ means that *file2* has the extra lines *n2* to *n3* following the line that has the number *n1* in *file1*, and ‘*n4,n5 c n6,n7*’ means that lines *n4* to *n5* in *file1* differ from lines *n6* to *n7* in *file2*). After each such line, **diff** prints the relevant lines from the text files, with *<* in front of each line from *file1* and *>* in front of each line from *file2*.

There are several options to **diff**, including **diff -i**, which ignores the case of letters when comparing lines, and **diff -b**, which ignores all trailing blanks.

diff -cn produces a listing of differences within *n* lines of context, where the default is three lines. The form of the output is different from that given by **diff**, with *+* indicating lines that have been added, *-* indicating lines that have been removed, and *!* indicating lines that have been changed.

diff *dir1 dir2* will sort the contents of directories *dir1* and *dir2* by name, and then run **diff** on the text files that differ.

7. **file** – determine the type of a file

file tests named files to determine the categories their contents belong to.

file *file1*

can tell if *file1* is, for example, a source program, an executable program or shell script, an empty file, a directory, or a library, but (a warning!) it does sometimes make mistakes.

8. **find** – find files of a specified name or type

find searches for files in a named directory and all its subdirectories.

find . -name '*.cc' -print

searches the current directory and all its subdirectories for files ending in **.cc**, and writes their names to the standard output. In some versions of Unix the names of the files will only be written out if the **-print** option is used.

find /local -name core -user user1 -print

searches the directory */local* and its subdirectories for files called *core* belonging to the user *user1* and writes their full file names to the standard output.

9. **ftp** – file transfer program – no longer recommended. See the section on **scp** and **ssh** instead.

ftp is an interactive file transfer program. While logged on to one machine (described as the local machine), **ftp** is used to logon to another machine (described as the remote machine) that files are to be transferred to or from. As well as file transfers, it allows the inspection of directory contents on the remote machine. There are numerous options and commands associated with **ftp**, and **man ftp** will give details of those.

A simple example **ftp** session, in which the remote machine is hermes, is shown below:-

ftp hermes.cam.ac.uk

If the connection to hermes is made, it will respond with the prompt:-

Name (hermes.cam.ac.uk:user1) :

(supposing *user1* is your username on your local machine). If you have the same username on hermes, then just press *Return*; if it is different, enter your username on hermes before pressing *Return*. You will then be prompted for your hermes password, which will not be echoed.

After logging in using **ftp** you will be in your home directory on hermes. Some Unix commands, such as **cd**, **mkdir**, and **ls**, will be available. Other useful commands are:

help

lists the commands available to you while using **ftp**.

get remote1 local1

creates a copy on your local machine of the file *remote1* from hermes. On your local machine this new file will be called *local1*. If no name is specified for the file on the local machine, it will be given the same name as the file on hermes.

send local2 remote2

copies the file *local2* to the file *remote2* on hermes, i.e., it is the reverse of **get**.

quit

finishes the **ftp** session. **bye** and **close** can also be used to do this.

Some machines offer a service called “anonymous ftp”, usually to allow general access to certain archives. To use such a service, enter *anonymous* instead of your username when you ftp to the machine. It is fairly standard practice for the remote machine to ask you to give your email address as a password. Once you have logged on you will have read access in a limited set of directories, usually within the /pub directory tree. It is good etiquette to follow the guidelines laid down by the administrators of the remote machine, as they are being generous in allowing such access. See leaflet G72 for more detailed examples of using ftp.

WARNING! When you use **ftp** the communications between the machines are not encrypted. This means that your password could be snooped when you use it make an **ftp** connection. If available, the commands **sftp** (secure file transfer program) or **scp** (secure remote file copy program) are preferable, as they provide encrypted file transfer. See later discussion of this **scp**.

10. **grep** – searches files for a specified string or expression

grep searches for lines containing a specified pattern and, by default, writes them to the standard output.

grep *motif1 file1*

searches the file *file1* for lines containing the pattern *motif1*. If no file name is given, **grep** acts on the standard input. **grep** can also be used to search a string of files, so

grep *motif1 file1 file2 ... filen*

will search the files *file1*, *file2*, ... , *filen*, for the pattern *motif1*.

grep -c *motif1 file1*

will give the number of lines containing *motif1* instead of the lines themselves.

grep -v *motif1 file1*

will write out the lines of *file1* that do NOT contain *motif1*.

11. **gzip** – compress a file

gzip reduces the size of named files, replacing them with files of the same name extended by .gz. The amount of space saved by compression varies.

gzip *file1*

results in a compressed file called *file1.gz*, and deletes *file1*.

gzip -v *file2*

compresses *file2* and gives information, in the format shown below, on the percentage of the file's size that has been saved by compression:-

```
file2 : Compression 50.26% -- replaced with file2.gz
```

To restore files to their original state use the command **gunzip**. If you have a compressed file *file2.gz* , then

gunzip *file2*

will replace *file2.gz* with the uncompressed file *file2*.

12. **help** – display info about bash builtin commands

help gives access to information about builtin commands in the **bash** shell. Using **help** on its own will give a list of the commands it has information about. **help** followed by the name of one of these commands will give information about that command. **help history**, for example, will give details about the **bash** shell history listings.

13. **info** – read online documentation

info is a hypertext information system. Using the command **info** on its own will enter the **info** system, and give a list of the major subjects it has information about. Use the command **q** to exit **info**. For example, **info bash** will give details about the **bash** shell.

14. **kill** – kill a process

To kill a process using **kill** requires the process id (PID). This can be found by using **ps**. Suppose the PID is 3429, then

```
kill 3429
```

should kill the process. If it doesn't then sometimes adding **-9** helps...

```
kill -9 3429
```

15. **lpr** – print out a file

lpr is used to send the contents of a file to a printer. If the printer is a laserwriter, and the file contains PostScript, then the PostScript will be interpreted and the results of that printed out.

```
lpr -Pprinter1 file1
```

will send the file *file1* to be printed out on the printer *printer1*. To see the status of the job on the printer queue use

```
lpq -Pprinter1
```

for a list of the jobs queued for printing on *printer1*. (This may not work for remote printers.)

16. **ls** – list names of files in a directory

ls lists the contents of a directory, and can be used to obtain information on the files and directories within it.

ls *dir1*

lists the names of the files and directories in the directory *dir1*, (excluding files whose names begin with **.**). If no directory is named, **ls** lists the contents of the current directory.

ls -a *dir1*

will list the contents of *dir1*, (including files whose names begin with **.**).

ls -l *file1*

gives details of the access permissions for the file *file1*, its size in kbytes, and the time it was last altered.

ls -l *dir1*

gives such information on the contents of the directory *dir1*. To obtain the information on *dir1* itself, rather than its contents, use

ls -ld *dir1*

17. **man** – display an on-line manual page

man displays on-line reference manual pages.

man *command1*

will display the manual page for *command1*, e.g **man** *cp*, **man** *man*.

man -k *keyword*

lists the manual page subjects that have *keyword* in their headings. This is useful if you do not yet know the name of a command you are seeking information about, but can produce a lot of output. To refine the output you could, for example, use **man -k** *keyword* | **grep** '**1**' to get a list of user commands with *keyword* in their headings (user commands are in section 1 of the man pages). The | means that the output of **man -k** is piped to (i.e., is used as the input for) **grep**.

man -Mpath *command1*

is used to change the set of directories that **man** searches for manual pages on *command1*

18. **mkdir** – make a directory

mkdir is used to create new directories. In order to do this you must have write permission in the parent directory of the new directory.

mkdir *newdir*

will make a new directory called *newdir*.

mkdir -p can be used to create a new directory, together with any parent directories required.

mkdir -p *dir1/dir2/newdir*

will create *newdir* and its parent directories *dir1* and *dir2*, if these do not already exist.

19. **more** – scan through a text file page by page

more displays the contents of a file on a terminal one screenful at a time.

more *file1*

starts by displaying the beginning of *file1*. It will scroll up one line every time the *return* key is pressed, and one screenful every time the *space* bar is pressed. Type **?** for details of the commands available within **more**. Type **q** if you wish to quit **more** before the end of *file1* is reached.

more -n *file1*

will cause *n* lines of *file1* to be displayed in each screenful instead of the default (which is two lines less than the number of lines that will fit into the terminal's screen).

20. **mv** – move or rename files or directories

mv is used to change the name of files or directories, or to move them into other directories. **mv** cannot move directories from one file-system to another, so, if it is necessary to do that, use **cp** instead – copy the whole directory using **cp -r oldplace newplace** then remove the old one using **rm -r oldplace**.

mv *name1 name2*

changes the name of a file called *name1* to *name2*.

mv *dir1 dir2*

changes the name of a directory called *dir1* to *dir2*, unless *dir2* already exists, in which case *dir1* will be moved into *dir2*.

mv *file1 file2 dir3*

moves the files *file1* and *file2* into the directory *dir3*.

21. **nice** – change the priority at which a job is being run

nice causes a command to be run at a lower than usual priority. **nice** can be particularly useful when running a long program that could cause annoyance if it slowed down the execution of other users' commands. An example of the use of **nice** is

nice gzip *file1*

which will execute the compression of *file1* at a lower priority.

If the job you are running is likely to take a significant time, you may wish to run it in the background, i.e., in a subshell. To do this, put an ampersand **&**, after the name of your command or script. For instance,

rm -r *mydir* **&**

is a background job that will remove the directory *mydir* and all its contents.

The command **jobs** gives details of the status of background processes, and the command **fg** can be used to bring such a process into the foreground.

22. **passwd** – change your password

Use **passwd** when you wish to change your password. You will be prompted once for your current password, and twice for your new password. Neither password will be displayed on the screen.

23. **ps** – list processes

ps displays information on processes currently running on your machine. This information includes the process id, the controlling terminal (if there is one), the cpu time used so far, and the name of the command being run.

ps

gives brief details of your own processes in your current session.

To obtain full details of all your processes, including those from previous sessions use:-

ps -fu *user1*

using your own user name in place of *user1*.

ps is a command whose options vary considerably in different versions of Unix. Use **man ps** for details of all the options available on the machine you are using.

24. **pwd** – display the name of your current directory

The command **pwd** gives the full pathname of your current directory.

25. **quota** – display disk quota and usage

quota gives information on a user's disk space quota and usage.

On some systems using **quota** without options will only give details of where you have exceeded your disk quota on local disks, in which case, use the -v option

quota -v

to get details of your quota and usage on all mounted filesystems.

26. **rm** – remove files or directories

rm is used to remove files. In order to remove a file you must have write permission in its directory, but it is not necessary to have read or write permission on the file itself.

rm *file1*

will delete the file *file1*. If you use

rm -i *file1*

instead, you will be asked if you wish to delete *file1*, and the file will not be deleted unless you answer **y**. This is a useful safety check when deleting lots of files.

rm -r *dir1*

recursively deletes the contents of *dir1*, its subdirectories, and *dir1* itself, and should be used with suitable caution.

rm -rf *dir1*

is like **rm -r**, except that any write-protected files in the directory are deleted without query. This should be used with even more caution.

27. **rmdir** – remove a directory

rmdir removes named empty directories. If you need to delete a non-empty directory **rm -r** can be used instead.

rmdir *exdir*

will remove the empty directory *exdir*.

28. **sort** – sort and collate lines

The command **sort** sorts and collates lines in files, sending the results to the standard output. If no file names are given, **sort** acts on the standard input. By default, **sort** sorts lines using a character by character comparison, working from left to right, and using the order of the standard character set.

sort -d

uses “dictionary order”, in which only letters, digits, and white-space characters are considered in the comparisons.

sort -r

reverses the order of the collating sequence.

sort -n

sorts lines according to the arithmetic value of leading numeric strings. Leading blanks are ignored when this option is used, (except in some System V versions of **sort**, which treat leading blanks as significant. To be certain of ignoring leading blanks use **sort -bn** instead.).

29. **ssh** – secure remote login program

ssh is used for logging onto a remote machine, and provides secure encrypted communications between the local and remote machines using the SSH protocol. The remote machine must be running an SSH server for such connections to be possible. For example,

ssh -X *YourCRSID@linux.phy.pwf.cam.ac.uk*

will commence a login connection to the Physics PWF server.

You can connect using your password for the remote machine, or you can set up a system of passphrases to avoid typing login passwords directly (see the man page for **ssh-keygen** for information on how to create these).

The optional **-X** flag makes it possible for the remote machine to open X windows on your local machine.

30. **scp** – securely copy files between computers

scp is used for copying files between any two computers that you can log on to with **ssh**.

Suppose you could log in via **ssh** to computer **computer1.co.uk** with user-id **britBoy** on which there was a file **/var/allmypasswords.txt**. And suppose that you wished to copy this file to one called **/home/ici.txt** on a different computer **ordinateur2.fr**, which you are customarily able to log into via **ssh** with user-id **pierre**. Then to effect that file transfer you would type the following from a terminal on any computer:

```
scp britboy@computer1.co.uk:/var/allmypasswords.txt pierre@ordinateur2.fr:/home/ici.txt
```

If it so happened that the terminal from which you wanted to execute the **scp** command was already on **computer1.co.uk** then you could use the simpler form:

```
scp /var/allmypasswords.txt pierre@ordinateur2.fr:/home/ici.txt
```

Likewise, if it so happened that the terminal from which you wanted to execute the **scp** command was already on **ordinateur2.fr** then you could use the simpler form:

```
scp britboy@computer1.co.uk:/var/allmypasswords.txt /home/ici.txt
```

scp can also copy directories and all the contents recursively. To do the above on directories one would need something like:

```
scp -r britboy@computer1.co.uk:/var pierre@ordinateur2.fr:/home
```

where the **-r** means “Do this recursively on directories”.