Computing in C for Science

Lecture 1 of 5

Dr. Steven Capper
http://www2.imperial.ac.uk/~sdc99/ccourse/
steven.capper99@imperial.ac.uk

16th November 2011

Imperial College London

Introduction to the Course

This course is based on the C course written by my PhD supervisor Dan Moore: http://www.ma.ic.ac.uk/~drmii

Aims of the Course

- To introduce modern C programming from scratch and,
- provide insight into scientific computing (floating point arithmetic, optimisation, . . .).

Five lectures spread over five weeks.

- Each lecture will take ≈ 1 hour,
- and involve at least an hour of practical work.
- This is very intense.
- Please feel free to ask questions outside course hours.

A Rough History of C

Invented ≈ 1970

By Dennis Ritchie working in Bell Labs USA; to facilitate development of a portable UNIX.

C has been standardised

- 1989 ANSI standard ratified *ANS X3.159-1989*.
- 1990 ISO standard *ISO/IEC 9899:1990*. Aka *C90*.
- 2000 ISO standard *ISO/IEC 9899:1999*. Aka *C99*.

C has evolved into C++

Bjarne Stroustrup developed C++ (C with class). Unlike C, C++ is still under very active development (C++11 being the most recent standard at the time of writing).

What are C and C++?

- C is a cross-platform, compiled, general-purpose language.
- C++ can loosely be thought of as C's object oriented big brother.

The vast majority of the programs running on your computer (including the operating system kernel), are written in either C or C++.

Why Use C? (Over Maple, Matlab, S-Plus...)

Speed

C programs are compiled to machine code, the resulting routines *can* run several orders of magnitude quicker than their equivalents in interpreted environments.

Flexibility

The C language is intrinsically low level, one can manipulate complex data structures with surprisingly little code.

Portability

A well written C program can target many different environments (Windows PCs, Linux workstations, Apple Macs, DEC Alphas, Embedded devices, . . .).

Getting Started

You will need:

- A C compiler (many different ones to choose from, some are free).
- Some documentation (such as the lecture notes/exercises from this course, a good book, online guides).
- Lots, and lots of time.

Free C compilers

Free as in free for academic/commercial use

Linux/UNIX

- `gcc` - The GNU Compiler Collection, C compiler. `http://gcc.gnu.org`.

Windows

- `cygwin` - A set of GNU libraries ported to Windows (free usage restricted to GPL apps), `http://www.cygwin.com/`.
- `MinGW` - Minamilist GNU for Windows (no restrictions), `http://www.mingw.org/`.
- `Visual C++ 2008 Express Edition` - Microsoft's free compiler (no restrictions), `http://www.microsoft.com/express/vc/`

Some Free IDEs

`gcc` is a command line driven program. Thankfully, there exist many free Integrated Development Environments (IDEs) that simplify the development process. Popular IDEs include:

- `Eclipse` - For Windows/UNIX. Primarily used for Java, but is good for C development too. `http://www.eclipse.org/`
- `NetBeans` - For Windows/UNIX. Another Java IDE with C development functionality. `http://netbeans.org/`
- `Xcode Tools` - For Apple Macs. The development environment used by Apple. `http://developer.apple.com/technology`

Some Non-Free C Compilers

- Borland/Inprise/Borland/Code Gear `C++ Builder/Turbo C++` - Can be found at: `http://www.codegear.com`
- `Intel` - for Windows/Linux (will compile good code for AMD processors too!). Free for personal use, academic/commercial licenses obtainable from: `http://www.polyhedron.com`
- `SilverFrost`(Salford) - for Windows, includes a C compiler, personal evaluation version from: `http://www.silverfrost.com/32/ftn95/personal_edi`
- `Microsoft Visual Studio 2008 Professional` - Microsoft's flagship compiler. Ninety day free trial available: `http://msdn2.microsoft.com/en-us/vstudio/product`

Calling all Students!

Microsoft DreamSpark

Microsoft have released the full Visual Studio 2008 Professional Edition and Server 2008 (and 2008 R2) for student use. `https://downloads.channel8.msdn.com/`

Licensing

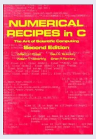
I have no idea how this is licensed, please do check before using it for research/commercial use!

Books for C

Kernighan and Ritchie (K&R2)

The C Programming Language, **Second Edition**, Prentice Hall. A very well-written, concise C reference.

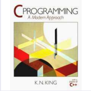
Numerical Recipes in C



By Press, Teukolsky, Vetterling & Flannery, **Second Edition**, CUP. Contains a lot of useful scientific computing information and provides high quality C example code. A free online edition can be found at: `http://www.nr.com`

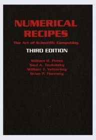
More Books for C/C++

King



C Programming A Modern Approach Norton & Company. A well paced introduction to the C programming language with some introductory material to C++ too.

Numerical Recipes in C++



By Press, Teukolsky, Vetterling & Flannery, **Third Edition**, CUP. A C++ only, heavily revised edition of the text. If you want to perform scientific computations in C++ rather than C, I would recommend this. More information at: `http://www.nr.com`

Building a C Program

- To *build* an executable from source, we carry out the following three steps:

Edit Source

Use a text editor to create a `.c` file.

Compile

With a C compiler, this creates *object file(s)*.

Link

Combine the object files together into an *executable*.

- These steps are can be automated by *Integrated Development Environments* (IDEs).

The Traditional Way to Start

```c
1  #include <stdio.h>
2
3  int main(void)
4  {
5     printf("Hello World!\n");
6     return 0;
7  }
```

The "Hello World" Program

A traditional first program started by Ritchie. This is one of the smallest possible C programs that demonstrates some functionality (printing to screen).

Line 1

A *pre-processor directive* (it begins with a #) advertising extra routines to the compiler.

Line 2

An empty line, or equivalently, a line consisting solely of *whitespace*. This is ignored by the compiler but makes the source code more readable.

Line 3

A *function declaration*, defining our `main` function. The `main` function is where our program starts and is known as an *entry point*. Our main function takes *no parameters* (`void`) and *returns* an integer (`int`).

Line 4

Opening brace, all statements enclosed between the braces {, } belong to the `main` function.

Line 5

A *statement*; the `printf` (print formatted) function is called with the argument `"Hello World!\n"`. This prints:
`Hello World!`
to *standard output* (usually a text console).

Line 6

A *return statement*, we exit `main` with a return code of 0. The system interprets 0 as "success".

Line 7

A *closing brace*, everything after this line does not belong to `main`.

Another C Program - What does this do?

```
 1 #include <stdio.h>
 2
 3 int main(void)
 4 {
 5    int low=-40, high=140, step=5, f, c;
 6    c = low;
 7    while (c <= high)
 8    {
 9       f = 32+9*c/5;
10       printf("%6d \t %6d\n", c, f);
11       c = c + step;
12    }
13    return 0;
14 }
```

Lines 1, 2, 3 & 4

Identical meaning as in the previous program.

Line 5

Local variable declarations; the integers `low`, `high`, `step`, `f` and `c` are declared. These are local to `main`. The variables `low`, `high` and `step` are *initialised* with the values; whilst `f` and `c` are *undefined*.

Line 6

The local variable `c` is *assigned* the value of `low`.

Lines 7, 8 & 12

A *while* loop is defined. For as long as the variable `c` is less than or equal to `high`, the code between the braces on lines 8 and 12 is executed.

Line 9

The local variable `f` is assigned a value from the integer arithmetic expression involving `c`.

Line 10

The variables `c` and `f` are printed to standard out, each six characters wide, separated by a tab and two spaces.

Line 11

The local variable `c` is incremented by `step`.

Lines 13 & 14

Have an identical meaning as in the last program.

`printf` - declared in `<stdio.h>`

We call `printf` as follows:

```
printf(formatString, var1, var2, ..., varN);
```

where,

`formatString`

The format string tells `printf` how many variables need printing. A format string can contain *format specifiers*, these tell `printf` exactly how to print out each variable, some examples:

`"%6d"`	print out an integer (6 characters wide).
`"%g"`	print out a floating point number.

`var1, ...`

`printf` accepts a variable list of arguments, which can be of different type. Care must be taken to match `formatString` with the variables.

Special Characters

- The backslash \ character in C has a special meaning, it is known as the *escape character*.
- We combine the escape character with other characters, to form an *escape sequence*, here are some examples:

`\n`	New line	`\f`	Form feed (new page)
`\t`	Tab	`\\`	\
`\b`	Backspace	`\"`	"
`\r`	Carriage return	`\'`	'
`\a`	Bell		

Numbers in C - 2 General Types

- *Integers* - `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, `unsigned long long`. Integer types in C can be thought of as rings of different sizes (i.e. hours on a clock face). Most importantly remember that division is not necessarily the inverse of multiplication.
- *Floats* - `float`, `double`, `long double`. These are *NOT* the same as $\mathbb{R}$ (associativity, and even commutativity not guaranteed, multiplicative inverses don't always exist). Programming floats well for numerical problems with large/small numbers is an art form.

Integer Types - For my 32 bit Windows Box

Type	Min	Max
short	-32768	32767
unsigned short	0	65535
int	-2147483648	2147483647
unsigned int	0	4294967295
long	-2147483648	2147483647
unsigned long	0	4294967295
long long	-9223372036854775808	9223372036854775807
unsigned long long	0	18446744073709551615

For example, here are two bit patterns for `short`:

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	= -1
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	= 7

(for more information see `<limits.h>`)

Integer Types(2)

- Two main subtypes *signed* and *unsigned*. Signed types use a sign bit.
- For signed types we, usually, have:
  - minimum value: $-2^{\text{size}-1}$
  - maximum value: $2^{\text{size}-1} - 1$
- For unsigned types we have:
  - minimum value: 0
  - maximum value: $2^{\text{size}} - 1$
- `short` is often used to conserve memory.
- `int` represents the *native* CPU integer type so is used for speed. (If in doubt use `int`).
- `long` and `long long` are used to maintain accuracy.

Integer Arithmetic

Base Operators

The four usual operators are defined $+$, $-$, `*` and $/$.

Ring arithmetic

Division is not always the reverse of multiplication:
`1/2=0, 0*2=0.`
Also, any result of a computation must lie within the ring, any number outside the range of the current data type will “wrap” around. (i.e. 11am + 3 hours gives 2pm).

Remainder Operator

The remainder operator % is unique to integer types, it acts as expected: `7%2 = 1`.

Floating Point Numbers (IEEE 754 Standard)

On my machine, a `float` (single precision) looks like:

b	e	e	e	e	e	e	e	e	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f	f

It consists of three parts, the *sign bit*(b), the *biased exponent*(e) and the *fraction*(f). We break down a number x:

$$x^{\text{float}} = (-1)^b \times 2^{e-127} \times \left(1 + f \times 2^{-23}\right), \quad \begin{array}{c} 0 < e < 255 \\ 0 \leq f \leq 2^{23} - 1 \end{array},$$

We have three special numbers, `-Inf` $(-\infty)$, `Inf` (∞) and `NaN` (Not a Number).
For `double` (double precision) we have:

$$x^{\text{double}} = (-1)^b \times 2^{e-1023} \times \left(1 + f \times 2^{-52}\right), \quad \begin{array}{c} 0 < e < 2047 \\ 0 \leq f \leq 2^{52} - 1 \end{array}.$$

Floating Point

Base Operators

As with integers, we have $+$, $-$, `*` and $/$.

Floating point code

- It looks like integer code but with a decimal point suffix.
- Scientific notation is achieved with e:
  `double speedofLight = 2.997e8;` (2.997×10^8)

Float Arithmetic

- Division is not always the reverse of multiplication.
- Operators may not be commutative!

$$A + B + C \neq A + C + B \qquad \text{(sometimes)}$$

The `pow(x, y)` function (declared in `<math.h>`)

Exponentiation

There is no exponentiation operator (e.g. $\wedge$, `**`) in C. Instead we have the following:

$$x^y = \texttt{pow}(\texttt{x}, \texttt{y})$$

This assumes x and y are of type `double`.

Beware

The pow function is often implemented as:

```
exp(y*ln(x))
```

For whole integer powers (i.e. x^2), one should perform the multiplication explicitly (`x*x`).

More Mathematical Functions in <math.h>

- Maths functions come with the *ANSI Standard C Library*, which contains many maths functions. To use them we need a: `#include <math.h>`
- Here some example functions:

```
sin(x)   asin(x)     sinh(x)   exp(x)
cos(x)   acos(x)     cosh(x)   log(x)
tan(x)   atan(x)     tanh(x)   log10(x)
sqrt(x)  atan2(x,y)  pow(x,y)  fabs(x)
```

(all the trigonometric functions use radians!)

Commenting C Programs

There are two ways of commenting files in C.

Traditional Way

Anything between `/*` and `*/` is a comment, i.e.

```
/* Hello World!  */
```

and,

```
/* This function is used to compute the
roots of a quadratic equation */
```

C++ Style

These are single line only, anything after `//` is a comment, i.e.

```
int c = 3; // set c to 3
```

Technically, C++ style comments aren't in the C standard. (But they are ubiquitous to C code anyway).

Variable Names

From K&R

"... Is a sequence of letters and digits. The first character must be a letter; the underscore _ counts as a letter. Upper and lower case letters are different."

- Punctuation or any other symbols are not allowed in variable names.
- The modern C standard discourages the use of an underscore as the first character of a variable name.

Simple Logical Expressions

- Are used to carry out branches (`if` statement) and loops (such as `for`, and `while`).
- Evaluate to either *true* (non-zero `int`) or *false* (zero).

Logical Operators

x	>	y	is x greater than y?
x	>=	y	is x greater than or equal to y?
x	<	y	is x less than y?
x	<=	y	is x less than or equal to y?
x	==	y	is x equal to y?
x	!=	y	is x different to y?

Compound Logical Expressions

We can create compound logical expressions using the following operators:

- `||` is a *logical or*. `le1 || le2` returns false if both `le1` and `le2` are false and true otherwise.
- `&&` is a *logical and*. `le1 && le2` returns true if and only if both `le1` and `le2` are true.
- `!` is a *logical not*. `!le1` returns the opposite of `le1`.

Here are two identical examples:

- `(x < 100) && (x%2 == 0)`
- `(x < 100) && !(x%2)`

Flow Control - `if`

Executes block(s) of code depending on the evaluation of a logical expression.

Simple `if`

```
        if (logical expression) {statements;}
```

`if`, `else if`, `else`

```
   if (logical expression)
      {statements;}
   else if (logical expression)
      {statements;}
   else if (logical expression)
      {statements;}
   else
      {statements;}
```

Flow Control - `while`

A `while` loop is used to repeatedly execute code as long as a logical expression is true.

Structure

```
   while (logical expression)
      { statements ;}
```

- If *logical expression* is false, then the *statements* are never executed.

Flow Control - `do {} while ()`

We place the *logical expression* after the *statements* giving us:

Structure

```
   do {statements;}
   while (logical expression)
```

- The *statements* are executed at least once.

`do while` or `while`?

Generally I prefer `while` over `do while`, as it forces me to initialise variables properly.

Flow Control - for loop

```
for ( start expression ;
      logical expression ;
      step expression)
      { statements ;}
```

- Print out ten numbers:

```
for (x=0; x < 10; x = x + 1)
   printf("x = %d\n", x);
```

- Keep looping indefinitely (printing out dots)

```
for (;;) printf(".");
```

Flow Control - switch - case

We can selectively execute code based on a value, using the following:

```
switch (integer_statement) {
case integer_value1: statements1; break;
case integer_value2: statements2; break;
case integer_value3:
case integer_value4: statements3; break;
default: statements4; break;}
```

- Execution starts at either one of the `case`'s or at `default`.
- Execution stops at the end } or at `break`.
- `case`, `default` and `break` are optional.

Some Loop Control Features

Execution of code inside a loop (`do`, `while`, `for`) can be manipulated by the following statements.

`break;`

Break out of the current loop. Any statements in the loop following the `break` are ignored and the loop condition automatically evaluates to false, ending the loop.

`continue;`

Jump to the end of the current loop (effectively ignoring everything below the `continue` statement. Whether or not the loop continues executing depends on the loop condition.

`scanf()` - Reading Data from Standard Input

For two variables A and B, both of type `double`, we use:

```
scanf("%lf %lf", &A, &B);
```

- where the % represent *format specifiers*

Format Specifiers

Consist of a %, a numerical width specification and a field code:

d	`int`	g	`float` (general form)
u	`unsigned int`	lf	`double` (fixed form)
f	`float` (fixed form)	le	`double` (exponential form)
e	`float` (exponential form)	lg	`double` (general form)

- and the & represents the *address* of the variable in memory. This is known as a *pointer reference operator*.

Why the &A in `scanf()`?

- Functions in C can return only one value.
- Sometimes we want more than one value to change.
- If we tell `scanf` *where* the variables are in memory, `scanf` can change them itself.

The ability to manipulate memory directly is what makes C so powerful. (and potentially dangerous).

Pointers

A *pointer* is a variable that stores a memory location, they are declared as follows:

```
double * ptrA;
```

& - Pointer reference operator

Returns the memory address (pointer to) of a variable.

```
double * ptrA = &A;    // ptrA points to A
```

*** - Pointer de-reference operator**

Converts a memory address to a variable:

```
* ptrA  = 1.234;       // A is now 1.234
```

Defining Functions

The C language only provides essential functionality, meaning a lot of functions need to be written yourself. Here are a few general rules for functions:

- Functions cannot define other functions within them.
- An optional single value can be returned.
- All arguments to a functions are passed by value and remain unaffected by the function.
- Passing pointers to functions allows them to “return” multiple variables.

An Example: Quadratic Equation Solver

As a worked example we write a function to solve the quadratic equation:

$$Ax^2 + Bx + C = 0 \qquad A, B, C \in \mathbb{R}$$

Our quadratic solver will:

- Take the three doubles `A`, `B` and `C` as arguments.
- Solve the quadratic and return an `int` signifying to the caller the type of answer available:
  - -1 A = 0, we have a linear equation.
  - 0 There are two distinct real roots.
  - 1 We have a pair of complex conjugate roots.
  - 2 Both roots are real and identical.

The Code

One possible function prototype is:

```
int quad_roots (double A, double B, double C,
                double * r1, double * r2);
```

- The variables `A`, `B` and `C` are unchanged by `quad_roots`.
- We need to return two doubles (the roots of the equation), thus we take in pointers `double *r1` and `double *r2`.
- C90 does not allow for complex number types (C99 does support them), so we have to think a little bit about the complex number case.

Code Snippet for Calling `quad_roots`

```
...
int main()
{
   double A, B, C, root1, root2;
   int quad_case;
   ...
   quad_case = quad_roots(A, B, C, &root1,
                          &root2);

   switch(quad_case)
   {
   case -1: linear equation
```

Code Snippet for `quad_roots`

```
int quad_roots(double A, double B, double C,
               double * r1, double *r2)
{
   double d;

   /* linear case */
   if (A == 0.0)
   {
      *r1 = -C/B;
      return -1;
   }

   /* compute the discriminant */
   d = B*B-4.0*A*C;
```

Declarations vs Definitions

Function Declarations

These tell the compiler about the *existence* of a function, which then allows us to call it. A declaration ends with a `;`.

```
int quad_roots (double A, double B, double C,
                double * r1, double * r2);
```

Function Definitions

The code making up the function is supplied to the compiler. A function can only be defined once. A definition contains braces { and }:

```
int quad_roots (double A, double B, double C,
                double * r1, double * r2)
{...}
```