Computing in C for Science

Lecture 2 of 5

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

23rd November 2011

Imperial College London

Preprocessor Directives

- One example is:

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

  Which tells the compiler to search `<stdio.h>` for functions.

- Another example is:

```
#define MAXSIZE 1024
```

  This replaces all occurrences of `MAXSIZE` with `1024`.

  - Define statements can be named in a similar way to variables, but
  - It is convention to use upper case for `#define` statements.

- Or even simpler:

```
#define NDEBUG
```

  Meaning `NDEBUG` is defined. This will be expanded later on.

Functions with Variable Number of Arguments

Sometimes we don't know in advance how many arguments (or what type) a function needs, so C allows functions to have an unknown number of arguments. Two examples we've seen so far are `printf` and `scanf`.

- The first parameter must be of a normal type (i.e. `int`).
- Three dots (`...`) are used for the last parameter.

```
int printf(char * formatString, ...)
```

Handling variable arguments

Variable arguments are manipulated using `va_start()`, `va_arg()`, and `va_end()`. These are found in `<stdarg.h>`.

Having just introduced these, I'm going to ask you **not** to use them! Arrays are almost always more appropriate to use.

The Stack

Let's consider this example function.

```
int hasRealRoots(double A,
        double B, double C)
{
    double d = B*B-4.0*A*C;
    if (d < 0) return 0;
    return 1;
}
```

- We need space to hold a copy of A, B and C.
- We need space to store our computed d.
- When we've finished, we need to get back to the calling function.

This is achieved by using a *stack*.

The Stack

Let's consider this example function.

```
int hasRealRoots(double A,
        double B, double C)
{
    double d = B*B-4.0*A*C;
    if (d < 0) return 0;
    return 1;
}
```

- We need space to hold a copy of A, B and C.
- We need space to store our computed d.
- When we've finished, we need to get back to the calling function.

This is achieved by using a *stack*.

The Stack

Let's consider this example function.

```
int hasRealRoots(double A,
        double B, double C)
{
    double d = B*B-4.0*A*C;
    if (d < 0) return 0;
    return 1;
}
```

- We need space to hold a copy of A, B and C.
- We need space to store our computed d.
- When we've finished, we need to get back to the calling function.

This is achieved by using a *stack*.

The Stack

Let's consider this example function.

```
int hasRealRoots(double A,
        double B, double C)
{
    double d = B*B-4.0*A*C;
    if (d < 0) return 0;
    return 1;
}
```

- We need space to hold a copy of A, B and C.
- We need space to store our computed d.
- When we've finished, we need to get back to the calling function.

This is achieved by using a *stack*.

The Stack

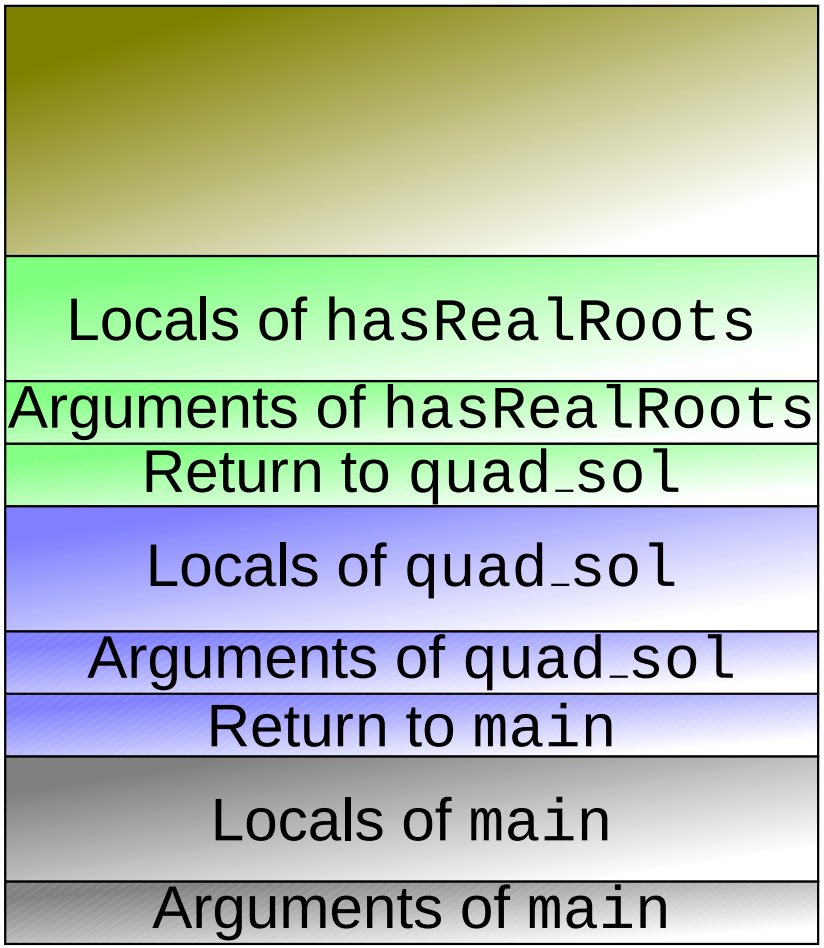
Let's consider this example function.

```
int hasRealRoots(double A,
        double B, double C)
{
    double d = B*B-4.0*A*C;
    if (d < 0) return 0;
    return 1;
}
```

- We need space to hold a copy of A, B and C.
- We need space to store our computed d.
- When we've finished, we need to get back to the calling function.

This is achieved by using a *stack*.

The Stack - Rough Sketch (Stack Frames)



- Consider the case where we have `main`, which calls `quad_sol`, which in turn calls `hasRealRoots`.
- We add and remove items from the stack as the program executes.
- Adding/removing items from the stack takes very little time.
- The stack is fixed in size, if we go over the top (“smash the stack”) , our program crashes.

Recursive Functions

As C uses a stack by default when calling functions, we are able to write functions that call themselves. These are called *recursive functions*.

An Example: Computing the Factorial

$$n! = \prod_{i=1}^{n} i, \quad 0! = 1, \qquad n \in \mathbb{N}.$$

Lends itself to be coded up as a recursive function.

A Tougher Example: Fibonacci Numbers

$$F_n = F_{n-1} + F_{n-2}, \qquad F_0 = F_1 = 1.$$

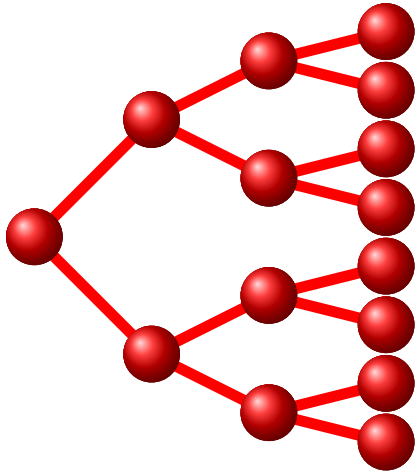
A naïve implementation of this will kill the stack, and take a very long time to execute.

Computing the Factorial

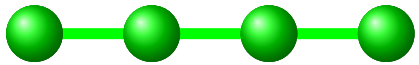
```c
#include <stdio.h>

int NFact(int N)
{
   if (N>1) return N*NFact(N-1);
   return 1;
}

int main()
{
   int n;
   printf("Enter n:");
   scanf("%d", &n);
   printf("%d! = %d\n", n, NFact(n));
   return 0;
}
```

Computing Fibonacci Numbers

```
int BadFib(int N)
{
   if (N < 2) return 1;
   return (BadFib(N-1) +
           BadFib(N-2));
}
```



```
int utilf(int a, int b, int n)
{
   if(n < 1) return b;
   return utilf(b, a+b, n-1);
}
int GoodFib(int n)
{
   return utilf(0, 1, n);
}
```



Functions - Summary

- Functions need to be declared before they are used. This is often done in *header files*.
- Up to one value can be returned from a function using the `return` statement.
- A variable `var` can be changed by a function if we pass the pointer `&var`.
- Pointers are declared using `type * variable;`

Pointers II

Reminder

- Declared using: `type * ptrVar;`
- Variable to pointer (pointer *referencing*): `ptrA = &A;`
- Pointer to variable (pointer *de-referencing*):
  `*ptrA = newVar;`

In addition

Pointers are memory addresses, and as such allow arithmetic!

Variable sizes

- Different data types take up different amounts of memory.
- Example `float` is smaller than `double`.
- The `sizeof` keyword gives a type's size (in bytes).

```
#include <stdio.h>

int main()
{
   printf("sizeof(float) = %d\n", sizeof(float));
   printf("sizeof(double) = %d\n", sizeof(double));
   printf("sizeof(int) = %d\n", sizeof(int));
   printf("sizeof(short) = %d\n", sizeof(short));
   return 0;
}
```

Arrays

- These are blocks of data, all of the same type. Each element is indexed using the array index operator: e.g. `myArray[index]` or `primes[3]`.
- Arrays are declared with types and sizes: e.g. `double xVector[3];`
- Arrays can be initialised: e.g. `int primes[6] = {2, 3, 5, 7, 11, 13};`
- All the elements of an array can be initialised to the *same* value: e.g. `double lotsOfDoubles[100] = {0.0};`
- **Arrays in C are indexed from 0!**

Accessing Array Elements

Arrays in C are indexed from 0!

```c
#include <stdio.h>

int main()
{
    int primes[6] = {2, 3, 5, 7, 11, 13};

    printf("first prime = %d\n", primes[0]);
    printf("next prime = %d\n", primes[1]);

    return 0;
}
```

Pointer Arithmetic

- Different data types in C are different sizes.
- Pointers are usually declared with a type (i.e. `int *`, `float *`, `double *`).

Relation to Arrays

Given the array `myArray` and an integer `index`, the following is true:

```
myArray[index] = *(myArray + index)
```

- And this is the reason array indices start at 0...

Debugging

- Once you've designed, typed up and successfully compiled your program, the difficult part begins! Debugging!
- Problems in the code are usually either easy to locate or stubbornly elusive.

Easier Problems

- Program crash/fault at the same point every run.

Nastier Problems

- Numerical output differs to what is expected.
- Program crashes seemingly randomly.

Debugging Techniques

In increasing order of difficulty:

Create Verbose Output

- A few strategically placed `printf` statements can prove to be helpful, but they are human readable:
  - too few and you miss the problem,
  - too many and they rapidly become useless.
- Straightforward to implement (and to comment out).

Code Defensively

- Consider specialised test cases.
- Write code to test intermediate results.
- Use the `assert` macro.

Use a Debugger

For those non-trivial problems.

Assertion Checking

In the header `<assert.h>`, the macro `assert` is defined. It has the following syntax:

`assert(`*`logical_expression`*`);`

If *logical_expression* evaluates to false (zero) then:

- Program execution stops immediately.
- An error message is sent to *stderr* (the console) stating the line number where the assertion failed.

```
assert( a != 0); /* a should never be zero */
```

Switching it off

Placing `#define NDEBUG` before all the `#include <assert.h>` statements de-activates assertion checking.

Assertion Checking - An Example

```c
#include <stdio.h>
#include <assert.h>
/* the sqrt function is much better than this... */
double squareRoot(double N)
{
   double x = 1.0;
   int loop;
   /* negative numbers are not allowed! */
   assert(N >= 0.0);
   if (N == 0.0) return 0.0;
   for (loop = 0; loop < 10; loop++)
      x = (x*x+N)/(2.0*x);
   return x;
}

int main()
{
   double square;
   printf("Enter a non-negative number:");
   scanf("%lg", &square);
   /* SHOULD HAVE: if (x < 0.0) ...          */
   printf("Square root = %g\n", squareRoot(square));
   return 0;
}
```

Using a Debugger

- Two good debuggers are `gdb` (GNU debugger) and Microsoft's Visual Studio debugger.
- Programs need to be compiled with debug information.
- Running a program straight through a debugger will show you the line of code that crashed it (if it crashes).

Interactive Analysis of Running Code

- Program execution can be paused at *breakpoints*.
- Functions can be *stepped into*, *stepped over*, or *stepped out from*.
- Variables/arrays can be *watched*.

Debuggers are very tricky to use at first, but can help isolate some very subtle problems.

Floating Point Numbers (IEEE 754 Standard)
(from the previous lecture)

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

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

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

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

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

Floating Point Number Analysis

In `<float.h>`, there are some useful quantities:

Quantity	Float	Double
Maximum Value	`FLT_MAX`	`DBL_MAX`
Minimum Value	`FLT_MIN`	`DBL_MIN`
Max Decimal Exponent	`FLT_MAX_10_EXP`	`DBL_MAX_10_EXP`
Min Decimal Exponent	`FLT_MIN_10_EXP`	`DBL_MIN_10_EXP`
ϵ	`FLT_EPSILON`	`DBL_EPSILON`

Floating point ϵ

ϵ is the smallest (in magnitude) number such that:

$$\texttt{1.0+}\epsilon\texttt{ != 1.0}$$

Floating Point Accuracy

- Some numbers can be represented in floating point exactly: e.g. 2^i, any integers that fit in the significand (mantissa).
- Most numbers need to be approximated, e.g. $\sqrt{2}$, π.
- One overlooked example is `0.1`!
- It is possible (though rare) to get exact answers from floating point arithmetic
- Relative errors of $\approx 10^{-15}$ for `double` and $\approx 10^{-6}$ for `float` are considered to be very good.
- Multiplication and division generally preserve relative error (but can take us outside the floating point range).

The Largest Source of Floating Point Errors

Addition and subtraction are the largest contributors to floating point error.

The Golden Rule

Do not subtract two very similar floating point numbers!

(This leads to "*catastrophic cancellation*".)

Casting

Casting (the conversion of data from one type to another) can either be *implicit* or *explicit*.

Implicit Casting

Conversion where there is no ambiguity (i.e. to a “bigger” data type) can be done automatically:

```
double x = 5; /* conversion from int to double */
double fEps = FLT_EPSILON; /* float to double */
```

Explicit Casting

If we wish to force a type conversion we place the destination type in brackets before the source variable:

```
oldtype oldData = ...
newtype newData = (newtype) oldData;
```

Explicit casting should be avoided if possible.

Scope: The Accessibility of Variables

Every variable in C has, associated with it, a *scope*. This defines how the variable can be accessed by functions in C. Some of the scoping rules are:

- All variables declared in the normal way inside a function are *local* to that function.
- Local variables can only be changed within the function they are defined, *unless*:
  - A pointer to a local variable may be passed to a function, extending the scope of that variable.
  - The are declared to be `extern` (more on this later).

Scope: Example 1

```c
#include <stdio.h>

void F1()
{
    int i = 4;
    printf("In F1(): I = %d\n", i);
}

int main()
{
    int i = 2;
    printf("In main(): I = %d\n", i);
    F1();
    printf("In main() again: I = %d\n", i);
    return 0;
}
```

Scope: Example 2

```c
#include <stdio.h>

void F1(int i)
{
   printf("In F1(): I = %d\n", i);
   i = 3;  /* what does this do? */
}

int main()
{
   int i = 2;
   printf("In main(): I = %d\n", i);
   F1(i);
   printf("In main() again: I = %d\n", i);
   return 0;
}
```

Scope: Example 3

```c
#include <stdio.h>

void F1(int * i)
{
   printf("In F1(): I = %d\n", *i);
   *i = 3;  /* what does this do? */
}

int main()
{
   int i = 2;
   printf("In main(): I = %d\n", i);
   F1(&i);
   printf("In main() again: I = %d\n", i);
   return 0;
}
```

Scope Blocks

Variable scope is not restricted to functions, indeed anything contained in braces (*scope blocks*) has it's own scope:

```
#include <stdio.h>

int main()
{
   int i = 1;

   if (i==1)
   {
      int j = 10; /* local to if block */
      printf("i+j=%d\n", i+j);
   }

   /* printf("j = %d\n", j); - error */

   return 0;
}
```

Local Variable Lifetime

The lifetime of a local variable is limited:

```
#include <stdio.h>

void F1()
{
   int i = 1;
   printf("In F1(): i = %d\n", i);
   i = i + 1; /* won't do much */
}

int main()
{
   F1();
   F1();
   return 0;
}
```

Local Variables

- Every time `F1()` is invoked, the value of `i` is reset to `1`.
- The variable `i` is a local variable which lives on the stack:
  - it will be destroyed every time we leave `F1()`,
  - and recreated every time we invoke `F1()`.
- We can ask the C compiler to retain the value of `i`:

`static` variables

A variable can be declared to be `static`, this tells the C compiler to set aside memory other than the stack to store the variable. The variable will not be destroyed until the program ends.

Why Use Static Variables?

- To count the number of times a function has been called.
- To have the function remember something between calls so on subsequent calls it can calculate what has changed (time, storage, the date, etc...), or resume from where it left off (reading from a list...)

Worked Example: Elapsed Time

- We write a function `timer` which returns the time since it was last called.
- Our program will contain two `.c` files to demonstrate linking.

Timer Example: timer()

```c
#include <time.h> /* for clock function */

double timer()
{
   static double oldTime = 0.0;
   double newTime, diff;

   newTime = clock();
   diff = newTime - oldTime;
   oldTime = newTime;
   return diff/CLOCKS_PER_SEC;
}
```

Timer Example: main()

```c
#include <stdio.h>

double timer(); /* declare timer */

int main()
{
   timer(); /* reset clock */
   printf("Wait... and hit return\n");
   getchar(); /* wait for return */
   printf("Elapsed time: %.2f seconds\n", timer());
   printf("Resetting clock, hit enter again\n");
   getchar();
   printf("Elapsed time: %.2f seconds\n", timer());
   return 0;
}
```

Global Variables - Shared between Functions

We can define a *global* variable:

```
#include <stdio.h>

double global = 42.0;

void F1()
{
   printf("Global = %g\n", global);
}

int main()
{
   F1();
   global = global + 1.0;
   F1();
   return 0;
}
```

Global Variables - Between Multiple Files

Global to All Files

If we want to share a global variable between files, then we define it *once* as:

```
type myglobalvariable = value;
```

Then in every other file we declare it as:

```
extern type myglobalvariable;
```

Global Only to Current File

If we do not wish to share a global variable, we define it as follows:

```
static type myPrivateGlobal = value;
```

Shared/Private Global Variables - File 1 of 2

```c
#include <stdio.h>

int globalEverywhere = 42;
static int globalHereOnly = 1;

void F2(); /* defined in file 2 */

int main()
{
   printf("globalEverywhere = %d\n", globalEverywhere);
   printf("globalHereOnly = %d\n", globalHereOnly);

   F2();

   printf("globalEverywhere = %d\n", globalEverywhere);
   printf("globalHereOnly = %d\n", globalHereOnly);

   return 0;
}
```

Shared/Private Global Variables - File 2 of 2

```c
#include <stdio.h>

extern int globalEverywhere;
/*if we forget the extern we get a linker error */

static int globalHereOnly = 100;

void F2()
{
   printf("F2(): globalEverywhere = %d\n",
                  globalEverywhere);

   printf("F2(): globalHereOnly = %d\n",
                  globalHereOnly);
}
```

Imperative versus Functional Programming

Two programming techniques are popular in C:

Imperative

- Very long functions.
- Lots of global variables.
- Very few function calls.

Functional

- Lots of small functions.
- Each function has a clearly defined rôle.
- Global variables avoided as much as possible.

I would encourage leaning towards the latter, a good program will contain traits from both styles.

The C Preprocessor - Conditional Compilation

We have already seen the `#include` statement. Conditional statements are also possible:

```
#include <stdio.h>

int main()
{
#ifdef NDEBUG
    printf("Assertions DISABLED\n");
#else
    printf("Assertions ENABLED\n");
#endif
    return 0;
}
```

This logic is performed at *compile time*.

The C Preprocessor - How to `#define` Externally

We are not restricted to `#define` statements in source code.

Visual Studio

In the Visual Studio "project properties" $\longrightarrow$ "C/C++" $\longrightarrow$ "Preprocessor" option we can specify preprocessor definitions.

gcc

In gcc we can specify define statements in the command line as follows:

```
gcc myfile.c -DNDEBUG -o myprogram
```