C for Science

Week 1

Getting Started, C Program structure, Data Types and I/O

Dr Maria Valera(m.valera-espina@doc.ic.ac.uk)

Imperial College London

February 2016

Table of Contents week 1

Table of Contents

Introduction to the Course

This course is based on its previous encarnations run by Sam Bolt, Dr Steve Capper and Dan Moore and Hanly and Koffman's book.

Aims of the course

- To introduce C programming from scratch.
- To provide an insight into scientific computing.

Learning Outcomes

- Design and implement efficient programs for a range of common scientific problems.

Course Content

- Program structure.
- C Types: conversions and casts.
- Control structures: sequence,selection and repetition.
- Conditions: operands, operators and its precendence.
- Arithmetic and Logical expressions.
- Functions.
- string Types.
- Pointers and Arrays Types.
- Memory management.
- Input/Output.
- Structures: Type Data Structure and Dynamic Data Stucture.
- C Standard Library and Scientific C-Libraries.
- Optimisation & Debugging.

Recommended Sources

- The C Programming Language: Brian W. Kernighan, Dennis M. Ritchie. Second Edition, Prentice Hall.
- Numerical Recipes in C: Teukolsky, Vetterling & Flannery. Second Edition,CUP. A free online edition: `http://www.nr.com`
- C: A reference Manual: Samuel P. Harbison, Guy L. Steele. Prentice Hall.
- Problem solving and Program Design in C. Jeri R. Hankly, Elliot B. Koffman. Pearson.
- Lots of C programming tutorials and resources: `http://www.cprogramming.com`

Table of Contents

C History

- An imperative, procedural, statically typed language developed at AT&T Bell Labs by Dennis Ritchie between 1969 and 1973.
- Development allowed Unix to be written in a high-level language. Previously it was written in PDP-7 assembler.
- In 1978, the “K&R” book by Brian Kernighan and Dennis Ritchie was released and served as an pre-standard.
- Use of C spread wildly, compilers becoming available on most architectures.
- Transformed into a standard C89/C90 and revised since (C99, C11).
- Many of the C constructs influenced later languages and exist in programming languages such as C++, Java, Perl, C#.

Why learn C?

- Still used heavily today: the Linux Kernel, GNU Core Utilities, Git etc.
- Provides high-level constructs, but also supports low-level operations such as directly accessing memory.
- Language constructs map into low-level code in a predictable manner, making it possible to reason about performance.
- Portable (with some significant caveats).

We will be developing under Linux. However, this course will not cover C programming using non-native C features (e.g. networking, shared memory).

Table of Contents

Getting Started

Make sure that you have:

- Editor.
- A C compiler (there are some free).
- Documentation (lecture notes/exemplars from this course, recommended book, online tutorials...).
- Plenty of time to practice.

Licensed C Compilers

- `Intel` - for Windows and Linux. Compiles highly optimised code for Intel and AMD processors. Free for personal use and academic use by students. Full-academic and commercial licenses obtainable from: `http://www.polyhedron.com`
- `Microsoft Visual Studio 2010, 2013 or 2015 Professional` - You can find free trials at: `https://www.visualstudio.com/products/visual-studio-professional-with-msdn-vs`

Free C Compilers

Linux

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

Windows

- Microsoft's free compiler
  `https://www.visualstudio.com/vs-2015-product-editions`
- `MinGW` - GNU for windows

Programming in C under Linux

- Getting a program to run:
  - Editing: vim / emacs / kate / gedit etc.
  - Compiling: gcc (GNU C Compiler)
  - Command Interpreter: bash / tcsh etc.
- To compile and link:

```
$ gcc source.c -Wall -Werror -pedantic -o output_executable
```

  - The `-Wall` option to gcc enables warnings and it's strongly recommended that you use it.
  - The `-Werror` option to gcc treats all warnings as errors and it's recommended that you use it.
  - Use `-pedantic` as well for better standards compliance checks.

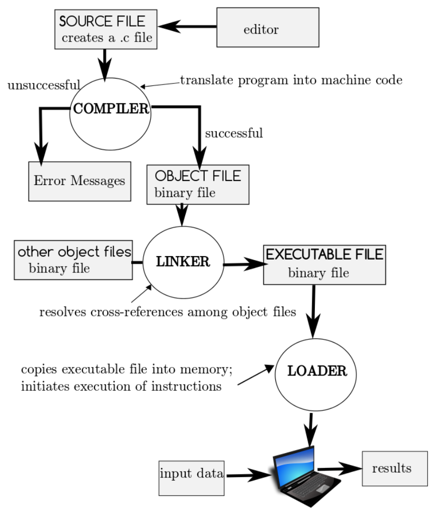
Programming in C under Linux

- To run:

```
$ ./output_executable
```

"./" refers to the current directory and it is required since it is often omitted from the execution path for security reasons.

C Program for execution



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

Integrated Development Environment

Integrated Development Environment (IDE) is a graphical application that provides tools to assist with editing, compiling and debugging of code. gcc (for Linux or MinGW) is a command line driven compiler:

```
$ gcc source.c -Wall -Werror -pedantic -o output_executable
```

Some IDEs:

- `Windows` - Visual Studio Professional (this can be obtained through DreamSpark if you're eligible)
- `Windows/Linux/Mac` - `Eclipse` or `Code::Blocks`

C89 or C99?

- C89 has a couple of painful restrictions which are removed in C99, but even now, C99 isn't fully supported by all compilers.
- In the real world, you may encounter code bases using only C89, so it's important you know the differences.
- If you don't specify flags to gcc, you will get a GNU dialect. Commit to a standard by passing `-std=c89` or `-std=c99` to gcc.
- These lecture notes will include examples using C89 syntax, but we will make it clear whenever we use C99-specific functionality.

C99 Syntactic Changes

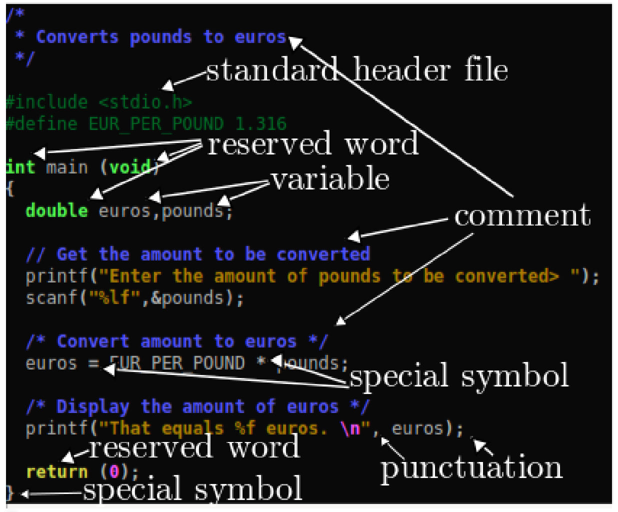
Couple of the most notable differences between C89 and C99 now:

- Variable declarations may only appear at the start of a block in C89. They may intermingle with code in C99.
- For-loop index variables can be declared at the site of the loop itself.
- Single line comments beginning with // are only valid in C99. Multi-line comments (`/* ... */`) are valid in both C89 and C99.
- C99 introduces a `bool` type (although implemented differently from C++'s `bool`).

Table of Contents

Converter

The canonical C example:



```
/*
 * Converts pounds to euros
 */

#include <stdio.h>
#define EUR_PER_POUND 1.316

int main (void)
{
  double euros,pounds;

  // Get the amount to be converted
  printf("Enter the amount of pounds to be converted> ");
  scanf("%lf",&pounds);

  /* Convert amount to euros */
  euros = EUR_PER_POUND * pounds;

  /* Display the amount of euros */
  printf("That equals %f euros. \n", euros);

  return (0);
}
```

The preprocessor

- The `#include` directive gives the program Converter access to a library `stdio.h`. This directive notifies the preprocessor that the definition of some identifiers used in the program (i.e. `printf` and `scanf`) are found in the standard header file (i.e. `stdio.h`). This is done at the compilation time.

```
#include <stdio.h> /*printf, scanf definitions */
```

- The function `printf` is the most common mechanism for writing data to standard output. The function `scanf` is used to obtained data from the standard input.
- The `#define` directive instructs the preprocessor to replace each occurrence of EUR_PER_POUND **constant macro** by `1.316`.
- There are a number of other directives (e.g. `#define`, `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, `#endif`, `#error`) some of which we'll look at later.

main Function Definition

- The `main` marks the beginning of the main function where the program execution begins. A C program must contain a `main` function. The remaining lines of the program form the *body* of the function which is enclosed in braces {,}.

`int main(void) {` *function body* `}`

- As defined by the C standard, `main`'s type must return an integer value `int` denoting success (zero) or failure (one), and can either take no parameters or a parameter count and an array of strings. We use the first version here.
- If we omit the return statement, `main` will return an undefined value.
- GCC won't even raise a warning for a missing return value unless we pass `-Wall`.

main Function

- A function body has two parts:
  - The **declarations** which tell the compiler what memory of cells are needed in the function (e.g. `euros` or `pounds`)
  - The **executable statements** (derived from the algorithm) are converted to machine code and executed by the computer. THis part is built through the capture requirement process during the problem analysis.

General C program structure

preprocessor directives
main function heading
{
variables declarations
executable statements
}

Identifiers and C Keywords

In Converter program there are words classified as **reserved words**, **standard identifiers** (e.g. `printf` and `scanf`) and **user-defined identifiers** (e.g. `euros` or `pounds`)

- Reserved words: they appear always in lowercase and have a defined meaning in C and cannot be used for other purposes.
- The list of ANSI C reserved words are:

short	do	extern	typedef
char	while	auto	return
float	if	volatile	union
int	else	static	const
long	switch	register	enum
double	case	goto	sizeof
unsigned	default	continue	struct
signed	for	break	void

Variables

- Standard identifiers: are words that define operations in the C Standard Library. Standard identifiers can be redefined and used for other purposes but this is consider bad practice.
- User-defined identifiers:user identifiers to name memory cells that will hold data and program results. The syntax rules for valid user-defined identifiers are:
  - A reserved words cannot be used as identifier and standard identifier should not be modified.
  - Only letters,numbers and underscores can be used to form an identifier.
  - An identifier cannot begin with a number.
  - Bare in mind e.g. POUNDS, pounds and Pounds for the compiler are *different* identifiers.

Table of Contents

Variable declarations and C Types

Variables are used for storing in memory cells the program's data. The values stored in the variables can be changed. The **variable declarations** communicate the compiler what kind of information will be store in each variable and which type of information will be stored in the memory cell:

```
double pounds; /*input data*/
double euros; /*output data*/
```

C requires to declare every variable used in a program. A varible declaration starts with an identifier (e.g. `double`) and communicates the compiler which type of data (e.g. real number) is stored in the varible.

Basic C Types

The integral types are (more information see `<limits.h>`):

Type	Size (minimum)	Range (minimum)
`char`	8 bits	unsigned: 0 to 255
		signed: -128 to 127
`short`	16 bits	signed: -32768 to 32767
`unsigned short`		unsigned: 0 to 65535
`int`	16 bits	signed: -32768 to 32767
`unsigned int`		unsigned: 0 to 65535
`long`	32 bits	signed: -2147483648 to 2147483647
`unsigned long`		unsigned: 0 to 4294967295
`long long`	64 bits	signed: -2^{63} to $2^{63}-1$
(C99)		unsigned: 0 to $2^{64}-1$

C also has floating point types. The standard says little about them, but on most platforms they usually follow IEEE 754 and have the following sizes and ranges:

Type	Size	Range	Precision (decimal)
`float`	32 bits	1.2e-38 to 3.4e+38	6 places
`double`	64 bits	2.3e-308 to 1.7e+308	15 places
`long double`	80 bits	3.4e-4932 to 1.1e+4932	19 places

Data models

Type	Data model			
	LP32	**ILP32**	**LLP64**	**LP64**
`(unsigned) short`	16 bits	16 bits	16 bits	16 bits
`(unsigned) int`	16 bits	32 bits	32 bits	32 bits
`(unsigned) long`	32 bits	32 bits	32 bits	64 bits
`(unsigned) long long`	64 bits	64 bits	64 bits	64 bits

- 32-bit systems:
  - **LP32** or **2/4/4** (int is 16-bit,long and pointer are 32-bit): Win16 API
  - **ILP32** or **4/4/4** (int,long and pointer are 32-bit): Win32 API, Unix, Linux and MAC OS X
- 64-bit systems:
  - **LLP64** or **4/4/8** (int and long are 32-bits and pointer is 64-bit): Win64 API
  - **LP64** or **4/8/8** (int is 32-bit,long and pointer are 64-bit): Unix, Linux and MAC OS X

Integer Types

- As seen in previous table, two main subtypes `signed` and `unsigned`. Signed types use a sign bit.
- For signed types:
  - minimum value: -2^{size-1}
  - maximum value: $2^{size-1}-1$
- For unsigned types:
  - minimum value: 0
  - maximum value: $2^{size}-1$
- `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.

stdint.h

The sizes and ranges of integer types can vary by platform in C. C99 introduced the header `stdint.h` which provides exact width integer types.

Type	Size	Range
`uint8_t`	8 bits	0 to 255
`uint16_t`	16 bits	0 to 65535
`uint32_t`	32 bits	0 to 4294967295
`uint64_t`	64 bits	0 to 18446744073709551615
`int8_t`	8 bits	-128 to 127
`int16_t`	16 bits	-32768 to 32767
`int32_t`	32 bits	-2147483648 to 2147483647
`int64_t`	64 bits	-9223372036854775808 to 9223372036854775807

Integer Arithmetic

Base Operators

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

Modulo Operator

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

Float Types

These are much more flexible numbers, but still NOT the same as $\mathbb{R}$. (See `<float.h>`)

- Because of the way the number is stored, even fairly simple-looking base-10 numbers (0.1, 0.2, 0.3, ...) are only stored as an approximation.
- Because numbers are kept as approximations: associativity and commutativity don't always apply and multiplicative inverses don't always exist.
- The way in which the numbers are stored can result in a loss of the least significant parts of numbers. This causes problems when adding/subtracting big and small numbers together.
- Programming floats well for numerical problems, especially with large/small numbers, is an art form!
- Four special numbers: `-Inf` ($-\infty$), `Inf`(∞), ϵ, `NaN`(Not exist number)

Float Arithmetic

Base Operators

Like integers, four usual operators are defined +, -, * and /

Floating point code

- It looks like integer code but with a decimal point suffix
- Scientific notation is achieved with `e`:
  `double massofearth = 5.976e24;`

Float Arithmetic

- Division is not always the reverse of multiplication.
- Operators may not be commutative!
  $A + B + C \neq A + C + B$

Integral Types

- Remember, sizes are platform dependent. Don't make assumptions about the width or ranges of integral types.
- The C operator `sizeof()` can be applied to any type or value in C to determine its width in `char`s:

```
sizeof(unsigned long)
```

- We'll see that `sizeof()` is vital for writing many C programs later.
- All integral types are signed by default, except for `char`. Whether `char` is signed or unsigned is platform-dependent.
- Integers types in C represent whole numbers. The integers lose all fractional data, multiplication is not always the inverse operation. Operations results higher than the max range will wrap to the smallest range value.

Table of Contents

Conversion of Numeric Types

Conversion of Numeric Types can be either *implicit* or *explicit*.
Implicit Casting
The conversion is done automatically so there is no ambiguity:

```
double x = 6.7; /*conversion from float to double*/
int k = 5, m = 4 , n; double x = 1.5, y =2.1, z;
```

Context of conversion	**Example**	**Cast**
Binary operator and operands of different types.	k+x	k will be cast to `double`
Assigment statement with type `double` and type `int` expression.	z = k / x;	Expression is evaluated first. Result converted to type `double`
Assigment statement with type `int` and type `double` expression.	n = x * y;	Expression is evaluated first. Result converted to type `int`. Fractional part lost

Casting

- In C the explicit conversion is called cast. Cast is used to prevent the lost of fractional part or to simple make clear to the reader the conversions that would occur automatically.
- To cast, we place the destination type in brackets before the varible to be converted:

```
double z = (double) k / (double) m;
double z = (double) ( k / m);
```

- avoid casting if possible.

Numerical inaccuracies

- Remember a fraction such as 1/3 is equal to zero. To get a floating point fraction, this should be rewritten 1.0/3.0.
- Some numbers can be represented in floating point exactly:any integers that fit in the significand (mantissa).
- Some fractions (1/3) cannot be represented exactly as binary numbers in the mantissa (**round-off error**).
- It is possible (though rare) to get exact answers from floating point arithmetic.
- Multiplication and division generally preserve relative error (butcan take us outside the floating point range) (**arithmetic underflow**).
- Addition and subtraction are the largest contributors to floating point error (**cancellation error**).

Table of Contents

Enumerated Types: enum

Enums provide an easy mechanism for creating identifiers:

```c
#include <stdio.h>

enum colour {RED, GREEN, YELLOW};

void printColour(enum colour col) {
  switch(col) {
    case RED: printf("RED\n"); break;
    case GREEN: printf("GREEN\n"); break;
    case YELLOW: printf("YELLOW\n"); break;
  }
}

int main(void) {
  enum colour col = YELLOW;
  printColour(col);
  return 0;
```

Enums as integers

- Enum values are implicitly convertible to integers.
- Unfortunately, arbitrary integers can also be converted to enums.
- Enum constants start at 0 and increment by 1.

```c
#include <stdio.h>

enum colour {RED, GREEN, YELLOW};

int main(void) {
  printf("%i, %i, %i\n", RED, GREEN, YELLOW);
  return 0;
}
```

This outputs:

```
0, 1, 2
```

Specifying enum constants

It's also possible to set the identifier explicitly. This can be helpful when you want the identifier to represent flags.

```c
#include <stdio.h>

enum renderflag {AMBIENT=1, DIFFUSE=2, SPECULAR=4};

int main(void) {
  int flags = AMBIENT | SPECULAR;
  if (flags & AMBIENT)  {printf("Ambient is set.\n")};
  if (flags & DIFFUSE)  {printf("Diffuse is set.\n")};
  if (flags & SPECULAR) {printf("Specular is set.\n")};
  return 0;
}
```

This outputs:

```
Ambient is set.
Specular is set.
```

Table of Contents

Defining Functions

- A Function in C embody a number of statements with a specific functionality.
- The structure of a function is:

```
return type name of the function(argument1, argument2,...)
where argument{x} is form as:  type variable
```

- The C language only provides essential functionality, meaning a lot of functions need to be written by yourself. Here are a few general rules for functions:
  - Functions cannot define other functions within them. But function can call other function within them.
  - Up to one value can be returned from a function using the return statement.
  - If the arguments to functions are passed by value, they remain unaffected by the function.
  - If the arguments to functions are passed by pointers, the arguments can be modified the function body.

Function Definition and Function Declaration

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 implementation of 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)  {body function (implementation code)}
```

What's that void doing in the parameter list?

Will it compile?

```
#include <stdio.h>

void greet() {
  printf("Hello world!\n");
}

int main(void) {
  greet(42, 5.4, "foo");
  return 0;
}
```

What's that `void` doing in the parameter list?

- In C, a function declaration with an empty parameter list *doesn't* declare a function that takes no parameters.
- Instead, it declares a function whose *parameter list is unknown*.
- To properly declare such a function, place `void` in the parameter list.

Will compile! ☹

```
#include <stdio.h>

void greet() {
  printf("Hello world!\n");
}

int main(void) {
  greet(42, 5.4, "foo");
  return 0;
}
```

Won't compile ☺

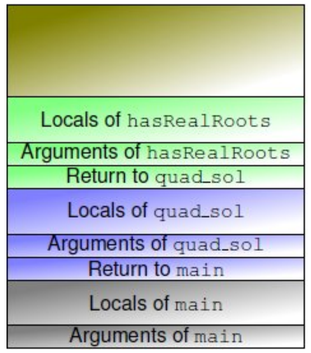
```
#include <stdio.h>

void greet(void) {
  printf("Hello world!\n");
}

int main(void) {
  greet(42, 5.4, "foo");
  return 0;
}
```

The Stack and the Heap

Before we consider arrays or recursive functions, we need to understand a little bit more of the underlying memory model:

- In C memory is allocated from one of two different regions: **the stack or the heap**.
- Values allocated on the stack cease to exist as soon as you leave the enclosing scope.
- Values allocated on the heap are *persistent*. They can exist as long as required.

The stack



- Consider the case where we have `main`, which calls `quad_sol` function, which in turn calls `hasRealRoots` function.
- 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.

Fibonacci Numbers

$\text{Fn} = Fn_1 + Fn_2; \quad F0 = F1 = 1$

Recursive Fibonacci function

```
int fib(int number) {
  if ((number == 0) || (number == 1)) {
    return number;
  } else {
    return fib(number - 1) + fib(number - 2);
  }
}
```

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

Table of Contents

Arrays

- These are blocks of data, all of the same type. Each element is indexed using the array index operator (e.g. `variable[index]`):

  `primes[3]`
- Arrays are declared with types and sizes:

  `double xVector[3];`
- It is possible to initialise an array during its declaration. It is also possible to avoid specifying the size part of the type when it can be inferred.

```
int a[3] = {1, 2, 3};          /* All elements specified */
int b[10] = {1, 2, 3, 4};      /* Some elements unspecified */
int c[] = {1,2};               /* Size inferred */
int d[][2] = {{1}, {9, 10}}; /* 2D, leading dimension inferred */
char str[] = "Hello!";         /* char array of size 7 */
```

Using C Arrays

- Arrays in C are indexed from 0!
- C arrays don't record their sizes. The size information is carried around in the array's type[1].
- The size of the array is incorporated into the type of the array. In C89, this meant that array sizes had to be constant. C99 supports variable length arrays – we'll cover that in a bit.
- **Remember**, like almost all other variables in C, until you initialise it, an array's value is undefined.

[1] C99 supports variable length stack-allocated arrays. The programmer is required to pass around the dynamic size information explicitly.

Arrays in C

The C standard says nothing about the way the stack is implemented. On x86 it grows down in memory so we draw it this way. Some platforms grow upwards or even use linked lists. Similarly, the compiler is free to reorder local variables in the way it thinks is most efficient.

Memory

A C program using arrays:

```
int main(void) {
  int q = 5;
  int r = 12;
  int arr[3][2];

  for(int row=0; row<3; ++row) {
    for(int col=0; col<2; ++col) {
      arr[row][col] = row*2 + col;
    }
  }
}
```

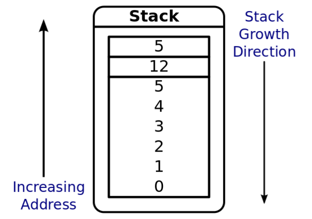
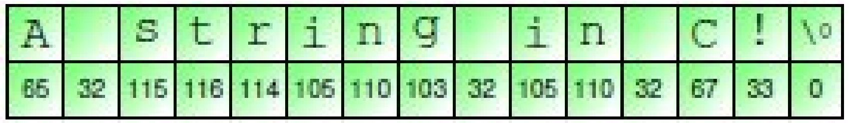


Table of Contents

Declaring and initilising string variables

- A string in C is declared as an array of type char. vspace2mm char stringvar[12]; vspace2mm
- **All** strings in C terminates with zero character (or '\0').



A		s	t	r	i	n	g		i	n		C	!	\0
65	32	115	116	114	105	110	103	32	105	110	32	67	33	0

- char values of 0-127 correspond to ASCII codes, their use should be relatively consistent between different compilers.
- All other values correspond to extended ASCII codes, the representations of which vary considerably between compilers.

Table of Contents

Flow Control

The flow control constructs control the flow of execution in a C program or function. The C control structures enable you to combine individual instructions into a single local unit with one entry point and one exit point. Instructions are orginased into three kinds of control structures to control execution flow: *sequence*,*selection* and *repetition*.

Recursive Fibonacci function

```
int helperfib(int a, int b, int n)
{
  if( n < 1) { return b;}
  return helperfib(b, a+b, n);
}
int goodfib(int n)
{
  return helperfib(0, 1, n);
}
```

Flow Control - while and do -then- while()

A while loop is used to repeatedly execute code as long as a logical expression is true.
If *condition* (logical expression) is false, then the statements are never executed.

```
while(condition) {
  /* Some statements */
}

do {
  /* Some statements */
} while (condition);
```

With do while the statements are executed at least once.

Flow Control - if-then-else Statements

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

```
if (condition) {
  /* Some statements */
} else if (condition) {
  /* Some statements */
} else {
  /* Some statements */
}
```

Flow Control - switch Statements

switch statements can only act on numeric values. Control flow falls through cases unless the break statement is used. The default case also exists.

```
switch(expression) {
  case constant:
    /* Statements */
    break; /* (optional) */
  case constant:
    /* Statements */
  default:
    /* Statements */
}
```

Flow Control - for Statements

For statements resemble their Java counterparts:

```
for(initialiser; condition; increment_expression) {
  /* Some statements */
}
```

Note:

- The initial statement, the condition (defaults to true) and the increment expression may all be omitted:

```
for (;;)
```

- Declaring a loop variable inside the initaliser is only supported in C99.

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.

Table of Contents

Logical expressions

Simple logical expressions

- These conditions are used to selected branches in conditional statements (`if`) and loops (e.g. `for`, and `while`). They establish criterion for either executing or skipping a group of statements.
- They evaluate either *true* (non-zero `int`) or *false* (zero).
- Logical operators:

Operator	**Meaning**
x > y	x greater than y
x >= y	x greater than or equal to y
x < y	x less than y
x <= y	x less than or equal to y
x == y	x equal to y
x != y	x not equal to y

Equality Operator

The following logical expression:

```
if (x = 3) {Statement;}
```

It will always evaluate true and execute the statement. It will also overwrite x with 3. This is not only undesirable as it is will not be testing the desired expression, but it is valid code so will not always throw an error - making debugging very tricky.

Way around

If we swap the variables `x` and 3:

```
if (3 == x) {Statement;}
```

Then if = was used rather than == it would cause an error as a value cannot be assigned to 3, but it keeps the expression logically equivalent. Getting into the habit of using the variables this way round can save hours of debugging!

Compound Logical Expressions

More compicated conditions can be built with three logical oprators: && (and), | | (or), ! (not).

- The && Operator (and)

Operand1	**Operand2**	**operand1 && operand2**
non-zero (true)	non-zero (true)	non-zero (true)
non-zero (true)	zero (false)	zero (false)
zero (false)	non-zero (true)	zero (false)
zero (false)	zero (false)	zero (false)

- The | | Operator (or)

Operand1	**Operand2**	**operand1 | | operand2**
non-zero (true)	non-zero (true)	non-zero (true)
non-zero (true)	zero (false)	non-zero (true)
zero (false)	non-zero(true)	non-zero (true)
zero (false)	zero (false)	zero (false)

- The ! Operator (not)

Operand1	**!Operand2**
non-zero(true)	zero(false)
zero(false)	non-zero(true)

Operator precendence and associativity

```
a - b * c / d
```

- * and / are evaluated before - due to precedence.
- * is evaluated before / due to (left to right) associativity.

```
if (x & MASK == 0)
```

- == has a higher precedence than & so it is executed first.
- To execute first the bitwise and operation parentheses are needed:

```
if ((x & MASK) == 0)
```

If in doubt always use brackets.

Precedence and associativity of operations

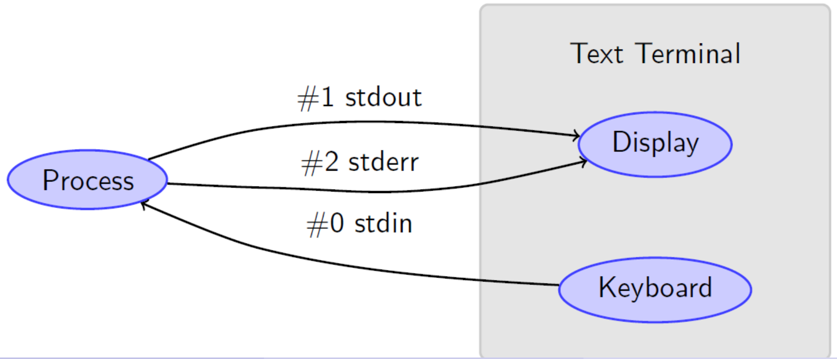
Precedence	**Operation**	**associativity**
highest (evaluate first)	() [] ->.	left to right
	postfix++ postfix–	left to right
	! p̃refix++ prefix– unary + unary - unary * unary & (cast) sizeof	right to left
	* / %	left to right
	binary + binary -	left to right
	≪ ≫	left to right
	<<= >>=	left to right
	== !=	left to right
	binary &	left to right
	binary ˆ	left to right
	binary |	left to right
	&&	left to right
	| |	left to right
	? :	right to left
	= += -= *= /= %= &= ˆ= |= ≪= ≫=	right to left
lowest (evaluate last)	,	left to right

Table of Contents

C Input/Output

C programs typically have three streams opened at start-up from which data can be read or written.

- Standard input (`stdin`) - usually a keyboard.
- Standard output (`stdout`) - usually the terminal.
- Standard error (`stderr`) - used for error or diagnostic information.



C Output (printf)

- The `printf()` or "print-formatted" function is a versatile way of printing text to `stdout`.
- Simplest usage is to print out a literal string:

```
printf("I'm a string!");
```

- Values can be substituted into the supplied string using *conversion specifications*. For example, %d converts the argument to a signed decimal notation, %c to a character:

```
char c = 'Q';
printf("An integer: %d, and a character %c.\n", 42, c);
```

will print:

```
An integer: 42, and a character Q.
```

C Output (printf)

printf is extremely powerful, supporting formatting, rounding and padding of values in multiple ways. We only have time to cover the very basics:

- The conversion specifiers:
  - `c` - character
  - `d, u` - signed(d) & unsigned(u) decimal integer
  - `o, x` - unsigned octal(o) & hexadecimal(x)
  - `f, e` - floating point value(f) in scientific notation(e)
  - `s` - string of characters
  - `p` - pointer value
- C contains a number of escape codes you can use when defining string or character literals in your program:
  - \n - newline, \f - new page
  - \t - tab, \b - backspace
  - \", \', \\ - literal double quote, single quote and backslash

  These aren't interpreted by printf - the actual ASCII character constants are generated in the string or character literals you create.

printf examples

```
#include <stdio.h>

int main(void) {
  int i = 72;
  printf("An integer: %d\n", i);

  float f = 3.1;
  double d = 72.1;
  printf("A double: %f, a float: %f\n", d, f);

  char s[] = "The quick brown fox...";
  printf("A string: %s\n", s);

  char x = 'A', y = 'B', z = '\n';
  printf("Two characters and a newline: %c%c%c", x, y, z);
}
```

A note on the security of `printf`

If you have some string `str` supplied by the user or an untrusted source, never print it using:

Security risk!

```
printf(str);
```

Instead use:

Safe

```
printf("%s", str);
```

The former version allows the attacker to supply conversion specifiers to `printf` and gives them the means to attack your program[2].

Despite being recognised over a decade ago, similar flaws are still being found.

[2]Tim Newsham, "Format String Attacks", Guardent Inc (September 2000)

C Input

- The C standard library provides the file descriptors `stdin`, `stdout`, `stderr` and the functions to read/write from them in `stdio.h`:
- Some functions require you supply a file descriptor, others use one of the standard ones implicitly.
- These functions deal with strings, but C has no native string type.
- In C, a string is a sequence of `char`s terminated with the character '\0'.
- We'll look at some of the C functions for reading data first:
  - `scanf()` - the input equivalent of `printf()`.
  - `fgets()` - reads a string from an input stream (not necessarily stdin).
  - `getchar()` - reads a single character from standard input.

`scanf()` for reading primitives

- `scanf()` uses a similar format string to `printf`.
- The following code shows how to read an integer from `stdin`:

```
int i;
int ret = scanf("%i", &i);
assert(ret == 1);
```

- The & is a unary operator that returns a reference to `i` called a *pointer*.
- These will be covered in the next lecture – for now, just remember that you will need to use this syntax to read in any primitive type.
- `scanf` returns the number of items matched (or values less than zero to indicate error conditions).

`scanf()` for reading strings

- We can also use `scanf()` to read a string from standard input (until the next whitespace character or newline):
- Since arrays are passed by reference, we don't need to use the & operator.

Security risk!

```
char buffer[50];
int ret = scanf("%s", buffer);
assert(ret == 1);
```

- We have no idea how long the string might be, so it's possible to overwrite the end of the supplied buffer.
- You'll need to use `scanf` in your tutorials, but remember the issues.

On scanf()

- We've covered scanf here because it's a part of the standard library and a common way of reading from input streams.
- It's incredibly easy to use wrongly, and when it is, it can pose security risks.
- P.J. Plauge, chair of the ANSI C library standardisation committee warns in his book "The Standard C Library":

  Be prepared, however, to give up on the scan functions beyond a point. Their usefulness, over the years, has proved to be limited.

fgets()

- fgets() reads a string from a stream into a buffer until the next newline, end of file, or the buffer is full.
- To read from standard in, we can do the following:

```
char buf[100];
int size = sizeof(buf);
fgets(buf, size, stdin);
```

- The size parameter is the maximum number of characters that fgets will write to the buffer. It will always terminate the string with '\0'.
- If a newline is read, it is also stored into the buffer.
- The variable stdin is the standard input stream defined in stdio.h.

fgets() with newline elimination

We can create our own function that calls fgets() and removes the final newline (if present):

```
void getstr(char buf[], int size) {
  fgets(buf, size, stdin);
  int i=0;
  while(buf[i]!='\0') {
    if (buf[i]=='\n') {
      buf[i] = '\0';
    } else {
      ++i;
    }
  }
}
```

getchar()

- getchar() retrieves a single character from standard input. We can use it as follows:

```
#include <stdio.h>

int main(void) {
  char c;
  c = getchar();
  printf("You typed the character %c\n", c);
  return 0;
}
```

- Suppose we want to read a character, then skip all characters up to and including the next newline. We can do:

```
char c = getchar();
while(getchar() != '\n');
```

Printing to standard error

- There's a version of printf called fprintf used for printing to files.
- stdio.h defines the files stdout and stderr for you.
- We can print to standard out and standard error as follows:

```
#include <stdio.h>

int main(void) {
  fprintf(stdout, "Hello standard out!\n");
  fprintf(stderr, "Hello standard error!\n");
  return 0;
}
```

- We can test this on the shell by discarding standard out and standard error:

```
$ gcc -Wall -pedantic -std=c89 fprintf.c -o fprintf
$ ./fprintf 1>/dev/null
$ ./fprintf 2>/dev/null
```

Table of Contents

Assertions

C has support for assertions, but you need to include the header assert.h.

`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.

Assertion example

Example

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

void doSomething(int i) {
  assert(i>1);
}

int main(void) {
  doSomething(1);
  printf("We don't reach this code.\n");
  return 0;
}
```

Summary

In this set of slides, we looked at:

- C is a cross-platform, compiled language which can produce results much quicker than other methods/languages.
- We use an IDE to write source, compile, link and debug our C programs.
- The basic structure of a C program has been demonstrated.
- We have seen the diffent types in C: basic, enumerate, array and string
- There are two categories of number in C: integers and floating point numbers.
- We have seen how logic and statements can control the flow of a program.
- printf and scanf will write and read from the console respectively.