C

So here is a simpler example. In file `hello.c`

```
#include <stdio.h>

/* This is a
   block comment */
int main(void)
{
  // do something interesting
  printf("hello world\n");

  return 0;
}
```

C

Points of note

C

Points of note

- General syntax — comments, curly brackets, semicolons, quotes, etc. — is pretty familiar

C

Points of note

- General syntax — comments, curly brackets, semicolons, quotes, etc. — is pretty familiar
- Filename can be anything that ends `.c`, no relationship connecting filenames to classes goes on: C does not have classes

C

Points of note

- General syntax — comments, curly brackets, semicolons, quotes, etc. — is pretty familiar
- Filename can be anything that ends `.c`, no relationship connecting filenames to classes goes on: C does not have classes
- The `#include` line we shall describe later: for now just think of it as something to put at the start of every C file

C

Points of note

- General syntax — comments, curly brackets, semicolons, quotes, etc. — is pretty familiar
- Filename can be anything that ends `.c`, no relationship connecting filenames to classes goes on: C does not have classes
- The `#include` line we shall describe later: for now just think of it as something to put at the start of every C file
- The function `main` is the entry point of the program. i.e., when the program is run, it starts executing from here

C

Points of note

- General syntax — comments, curly brackets, semicolons, quotes, etc. — is pretty familiar
- Filename can be anything that ends `.c`, no relationship connecting filenames to classes goes on: C does not have classes
- The `#include` line we shall describe later: for now just think of it as something to put at the start of every C file
- The function `main` is the entry point of the program. i.e., when the program is run, it starts executing from here
- It doesn't have any fancy type: it just returns an integer

C

- In this example `main` has an empty (void) argument list: C likes you to indicate explicitly there are no arguments

C

- In this example `main` has an empty (void) argument list: C likes you to indicate explicitly there are no arguments
- The function `printf` (“print formatted”) prints stuff, here a string with a newline at the end (the `\n`)

C

- In this example `main` has an empty (void) argument list: C likes you to indicate explicitly there are no arguments
- The function `printf` (“print formatted”) prints stuff, here a string with a newline at the end (the `\n`)
- The program exits when you return from `main`

C

- In this example `main` has an empty (void) argument list: C likes you to indicate explicitly there are no arguments
- The function `printf` (“print formatted”) prints stuff, here a string with a newline at the end (the `\n`)
- The program exits when you return from `main`
- `main` returns a value back to the operating system when the program finishes. The OS can use this in various ways, if it wishes. The convention is `0` means “finished successfully”, while non-zero values can signify various kinds of error

C

We can compile this file

```
% gcc -Wall -o hello hello.c
```

C

- The % is a command line prompt

C

- The % is a command line prompt
- `-Wall` is a option to the compiler that tells it to report all warnings. A warning is something in your code that might not be technically wrong, but is sufficiently dodgy to be worth looking at. Always use this option. You should aim to write code with no warnings (and no errors!)

C

- The % is a command line prompt
- `-Wall` is a option to the compiler that tells it to report all warnings. A warning is something in your code that might not be technically wrong, but is sufficiently dodgy to be worth looking at. Always use this option. You should aim to write code with no warnings (and no errors!)
- `-Wextra` gives even more warnings

C

- The % is a command line prompt
- `-Wall` is a option to the compiler that tells it to report all warnings. A warning is something in your code that might not be technically wrong, but is sufficiently dodgy to be worth looking at. Always use this option. You should aim to write code with no warnings (and no errors!)
- `-Wextra` gives even more warnings
- `-Werror` makes warnings into errors: the compiler will refuse to produce any output if there are any warnings

C

- The % is a command line prompt
- `-Wall` is a option to the compiler that tells it to report all warnings. A warning is something in your code that might not be technically wrong, but is sufficiently dodgy to be worth looking at. Always use this option. You should aim to write code with no warnings (and no errors!)
- `-Wextra` gives even more warnings
- `-Werror` makes warnings into errors: the compiler will refuse to produce any output if there are any warnings
- `-o hello` says put the compiled program in the file named `hello`. This filename can be anything you like, not necessarily related to the source code file name

Aside

Note: this is an example of compiling a C program using a command-line compiler (The GNU C compiler `gcc` in this case)

Aside

Note: this is an example of compiling a C program using a command-line compiler (The GNU C compiler `gcc` in this case)

Other compilers will likely take different arguments, e.g., another compiler might not recognise `-Wall` and could have something else equivalent (e.g., `-v`)

Aside

Note: this is an example of compiling a C program using a command-line compiler (The GNU C compiler `gcc` in this case)

Other compilers will likely take different arguments, e.g., another compiler might not recognise `-Wall` and could have something else equivalent (e.g., `-v`)

Many IDEs exist that are supposed to make the management and compilation of large programs easier (e.g., Eclipse, Visual Studio, Xcode)

Aside

Note: this is an example of compiling a C program using a command-line compiler (The GNU C compiler `gcc` in this case)

Other compilers will likely take different arguments, e.g., another compiler might not recognise `-Wall` and could have something else equivalent (e.g., `-v`)

Many IDEs exist that are supposed to make the management and compilation of large programs easier (e.g., Eclipse, Visual Studio, Xcode)

Exercise. Investigate these to find something that suits your personal taste

Aside

NB: there is a difference between an IDE and a compiler

Aside

NB: there is a difference between an IDE and a compiler

Compiler: converts a text source program into executable code

Aside

NB: there is a difference between an IDE and a compiler

Compiler: converts a text source program into executable code

IDE: a tool to help the programmer write better programs

Aside

NB: there is a difference between an IDE and a compiler

Compiler: converts a text source program into executable code

IDE: a tool to help the programmer write better programs

An IDE will generally *contain* a compiler that it calls when you want to produce executable code, but they are very different things

Aside

NB: there is a difference between an IDE and a compiler

Compiler: converts a text source program into executable code

IDE: a tool to help the programmer write better programs

An IDE will generally *contain* a compiler that it calls when you want to produce executable code, but they are very different things

Keep them separate in your mind

More Aside

Many C compilers exist, both paid-for and free. Different compilers may produce more or less efficient compiled code from the same source.

More Aside

Many C compilers exist, both paid-for and free. Different compilers may produce more or less efficient compiled code from the same source.

If all is well, a given standard-compliant (e.g., ANSI C11) C program should compile with a standard-compliant C compiler and should run and produce equivalent results independent of the compiler

More Aside

Many C compilers exist, both paid-for and free. Different compilers may produce more or less efficient compiled code from the same source.

If all is well, a given standard-compliant (e.g., ANSI C11) C program should compile with a standard-compliant C compiler and should run and produce equivalent results independent of the compiler

Exercise. Think about why I said “equivalent results”, not “identical results”

More Aside

Not all (any?) C compilers are fully standard-compliant

More Aside

Not all (any?) C compilers are fully standard-compliant

Not many C programs are fully standard-compliant

More Aside

Some compiler writers deliberately put support for non-standard things in their compilers: for reasons both good and bad

More Aside

Some compiler writers deliberately put support for non-standard things in their compilers: for reasons both good and bad

Good: to add extensions that are genuinely useful to the programmer

More Aside

Some compiler writers deliberately put support for non-standard things in their compilers: for reasons both good and bad

Good: to add extensions that are genuinely useful to the programmer

Bad: to add extensions that the programmer will become reliant on, not realising they are non-standard, so locking them in to using this particular compiler

More Aside

Some compiler writers deliberately put support for non-standard things in their compilers: for reasons both good and bad

Good: to add extensions that are genuinely useful to the programmer

Bad: to add extensions that the programmer will become reliant on, not realising they are non-standard, so locking them in to using this particular compiler

A program that excludes extensions and follows the standard will be much more portable

More Aside

Gcc is a widely available and widely used free compiler on a large number of architectures that produces reasonable (but not the best) code

More Aside

Gcc is a widely available and widely used free compiler on a large number of architectures that produces reasonable (but not the best) code

It's not the case of paying more to get a better compiler...

More Aside

Gcc is a widely available and widely used free compiler on a large number of architectures that produces reasonable (but not the best) code

It's not the case of paying more to get a better compiler...

Many other C compilers exist: Intel; Clang; Microsoft; Norcroft; etc.

More Aside

Also, use a **text editor** (or an IDE) to write programs, not a word processor

More Aside

Also, use a **text editor** (or an IDE) to write programs, not a word processor

You are not that stupid are you?

More Aside

Also, use a **text editor** (or an IDE) to write programs, not a word processor

You are not that stupid are you?

And always use a `fixed width` font when printing out or viewing code. Layout is important in all languages, particularly in C

Running the program

```
% ./hello
hello world
```

C

Running the program

```
% ./hello
hello world
```

I usually include the `./` to ensure I run the program named `hello` that lives in the current directory, not some program of the same name from somewhere else in the system

C

The program is a stand-alone binary (machine instructions) which you can simply run

C

The program is a stand-alone binary (machine instructions) which you can simply run

It is compiled for a specific OS and hardware architecture, generally the machine you used the compiler on

C

The program is a stand-alone binary (machine instructions) which you can simply run

It is compiled for a specific OS and hardware architecture, generally the machine you used the compiler on

So that binary probably won't run on a different OS or different hardware architecture: the program will need recompiling for them

C

The program is a stand-alone binary (machine instructions) which you can simply run

It is compiled for a specific OS and hardware architecture, generally the machine you used the compiler on

So that binary probably won't run on a different OS or different hardware architecture: the program will need recompiling for them

There is no analogue to the `java` runtime you need to run a Java program

C

Java: “write once, compile once, run everywhere”

C: “write once, compile everywhere, run everywhere”

C

Java: “write once, compile once, run everywhere”

C: “write once, compile everywhere, run everywhere”

Another trade-off

C

Java: “write once, compile once, run everywhere”

C: “write once, compile everywhere, run everywhere”

Another trade-off

Compiling Java produces machine-independent (byte)code that will run anywhere — anywhere there is a Java runtime to execute that code

C

Java: “write once, compile once, run everywhere”

C: “write once, compile everywhere, run everywhere”

Another trade-off

Compiling Java produces machine-independent (byte)code that will run anywhere — anywhere there is a Java runtime to execute that code

Compile C produces machine-specific code that only runs on one OS/architecture, but is optimised for that architecture

A **bad** program. `hello2.c`

```
#include <stdio.h>

int main(void)
{
  int n;

  n = n + 1;

  printf("hello world\n");

  return 0;
}
```

C

Compiler Warnings

```
% cc -Wall -o hello2 hello2.c
hello2.c: In function 'main':
hello2.c:7:5: warning: 'n' is used uninitialized
in this function
```

C

Compiler Warnings

A simple example of a warning message

C

Compiler Warnings

A simple example of a warning message

It is **very** important you get used to reading warning and error messages

C

Compiler Warnings

A simple example of a warning message

It is **very** important you get used to reading warning and error messages

You will see loads!

C

Compiler Warnings

A simple example of a warning message

It is **very** important you get used to reading warning and error messages

You will see loads!

Get used to them, and get used to fixing the things they refer to: **don't ignore warnings**

C

Compiler Warnings

A simple example of a warning message

It is **very** important you get used to reading warning and error messages

You will see loads!

Get used to them, and get used to fixing the things they refer to: **don't ignore warnings**

The quality of error messages varies with the compiler. Gcc produces generally reasonable messages

In this case, the compiler happens to generate a runnable executable; for more serious errors it wouldn't

In this case, the compiler happens to generate a runnable executable; for more serious errors it wouldn't

What happens when you run it is difficult to say...

In this case, the compiler happens to generate a runnable executable; for more serious errors it wouldn't

What happens when you run it is difficult to say...

Note that very exceptionally this kind of thing (running with undefined results) it what you want, but only if you are a programmer who is either (a) very clever, or (b) very stupid

C

Compiler Warnings

The C compiler Clang is reasonably widely available

C

Compiler Warnings

The C compiler Clang is reasonably widely available

One of its design aims is to give detailed and accurate error and warning messages

C

Compiler Warnings

The C compiler Clang is reasonably widely available

One of its design aims is to give detailed and accurate error and warning messages

And to replace Gcc

C

Compiler Warnings

The C compiler Clang is reasonably widely available

One of its design aims is to give detailed and accurate error and warning messages

And to replace Gcc

It is still under heavy development

C

Compiler Warnings

The C compiler Clang is reasonably widely available

One of its design aims is to give detailed and accurate error and warning messages

And to replace Gcc

It is still under heavy development

It also uses `-Wall` to show warnings

C

Compiler Warnings

```
% clang -Wall -o hello2 hello2.c
hello2.c:7:7: warning: variable 'n' is uninitialized when
used here
      [-Wuninitialized]
  n = n + 1;
      ^
hello2.c:5:8: note: initialize the variable 'n' to silence
this warning
  int n;
       ^
        = 0
```

C

Compiler Warnings

```
% clang -Wall -o hello2 hello2.c
hello2.c:7:7: warning: variable 'n' is uninitialized when
used here
      [-Wuninitialized]
  n = n + 1;
      ^
hello2.c:5:8: note: initialize the variable 'n' to silence
this warning
  int n;
       ^
        = 0
```

Here Clang even gives a suggestion on how to fix the warning

C

Function Definition

```
#include <stdio.h>

int factorial(int n)
{
  if (n < 2) return 1;
  return n*factorial(n-1);
}

int main(void)
{
  printf("factorial of %d is %d\n", 10, factorial(10));
  return 0;
}
```

Produces output

```
factorial of 10 is 3628800
```

C

```
printf("factorial of %d is %d\n", 10, factorial(10));
```

The first argument to `printf` is a *template* for the output

C

```
printf("factorial of %d is %d\n", 10, factorial(10));
```

The first argument to `printf` is a *template* for the output

Everything apart from % and `\n` are copied directly

C

```
printf("factorial of %d is %d\n", 10, factorial(10));
```

The first argument to `printf` is a *template* for the output

Everything apart from % and `\n` are copied directly

The backslash introduces special characters; in particular `\n` means “put a newline here”

C

```
printf("factorial of %d is %d\n", 10, factorial(10));
```

The first argument to `printf` is a *template* for the output

Everything apart from % and `\n` are copied directly

The backslash introduces special characters; in particular `\n` means “put a newline here”

The % says “read the next argument and put its value here”

C

The character after the % indicates how the argument should be treated

C

The character after the % indicates how the argument should be treated

%d means an integer

C

The character after the % indicates how the argument should be treated

`%d` means an integer

`%f` means a floating point number

C

The character after the % indicates how the argument should be treated

`%d` means an integer

`%f` means a floating point number

`%s` means a string

C

The character after the % indicates how the argument should be treated

`%d` means an integer

`%f` means a floating point number

`%s` means a string

Generally it is up to the programmer to get arguments of the right types in the right order

C

```
printf("integer %d\nfloating point %f\nstring %s\n",
       23 + 42, 99.0, "hello world");
```

produces

```
integer 65
floating point 99.000000
string hello world
```

when run

C

Incorrect code:

```
printf("integer %d\nfloating point %f\nstring %s\n",
99.0, 23 + 42, "hello world");
```

produces

```
printf1.c: In function 'main':
printf1.c:9:3: warning: format '%d' expects type 'int',
but argument 2 has type 'double'
printf1.c:9:3: warning: format '%f' expects type 'double',
but argument 3 has type 'int'
```

when you try to compile it

C

Other compilers might not be so helpful and simply do what you ask

C

Other compilers might not be so helpful and simply do what you ask

Giving a floating point number to `%d` the compiler might simply interpret the (bit pattern that represents the) floating point number as (a bit pattern that represents) an integer, and print it

C

Other compilers might not be so helpful and simply do what you ask

Giving a floating point number to `%d` the compiler might simply interpret the (bit pattern that represents the) floating point number as (a bit pattern that represents) an integer, and print it

This is a part of the “you asked for it, you got it” approach of C

C

`printf` is a lot more powerful than this: it does all kinds of formatted output (thus the “`f`” in the name)

C

`printf` is a lot more powerful than this: it does all kinds of formatted output (thus the “`f`” in the name)

Look at the documentation for `printf` for the gory details

C

`printf` is a lot more powerful than this: it does all kinds of formatted output (thus the "`f`" in the name)

Look at the documentation for `printf` for the gory details

`man printf` on Linux/Unix systems

C

`printf` is a lot more powerful than this: it does all kinds of formatted output (thus the "`f`" in the name)

Look at the documentation for `printf` for the gory details

`man printf` on Linux/Unix systems

In fact, there is masses of documentation online, e.g., `man cos` for the cosine function

C

`printf` is a lot more powerful than this: it does all kinds of formatted output (thus the "`f`" in the name)

Look at the documentation for `printf` for the gory details

`man printf` on Linux/Unix systems

In fact, there is masses of documentation online, e.g., `man cos` for the cosine function

These manual pages contain a great amount of detailed information: make sure you read them closely to get the most benefit

C

Exercise. Compile and run `hello.c` on your own machine

Exercise. Modify `hello2.c` to print the value of `n`. Try on a variety of different OSs and compilers and compare the results

Exercise. Read up on `printf`. How do you print a percent (%), a double quote (") and a backslash (\)? What is the difference between `%e`, `%f` and `%g`?

C

Exercise. Modify `factorial` to print

```
 1         1
 2         2
 3         6
 4        24
 5       120
 7      5040
 8     40320
 9    362880
10   3628800
```

Types

C has relatively few built-in types—remember it's low level!—actually just versions of types supported natively by hardware

Types

C has relatively few built-in types—remember it's low level!—actually just versions of types supported natively by hardware

Integers of various kinds and sizes:

Types

C has relatively few built-in types—remember it's low level!—actually just versions of types supported natively by hardware

Integers of various kinds and sizes:

- `char`
- `short int` (or simply `short`)
- `int`
- `long int` (or simply `long`)
- `long long int` (or simply `long long`)

Types

Integers

Not every compiler supports all these types, particularly `long long`

Types

Integers

Not every compiler supports all these types, particularly `long long`

As a language, C is adaptable to many kinds of hardware, from tiny embedded systems to huge mainframes

Types

Integers

Not every compiler supports all these types, particularly `long long`

As a language, C is adaptable to many kinds of hardware, from tiny embedded systems to huge mainframes

These are all ranges of integers that have proved to be useful in real programs

Types

Integers

Not every compiler supports all these types, particularly `long long`

As a language, C is adaptable to many kinds of hardware, from tiny embedded systems to huge mainframes

These are all ranges of integers that have proved to be useful in real programs

Interestingly, the C standard *does not specify how big each of these types are*

Types

Integers

Not every compiler supports all these types, particularly `long long`

As a language, C is adaptable to many kinds of hardware, from tiny embedded systems to huge mainframes

These are all ranges of integers that have proved to be useful in real programs

Interestingly, the C standard *does not specify how big each of these types are*

An `int` is often 32 bits (4 bytes), but it doesn't have to be

Types

Integers

This helps the adaptability of C to many kinds of hardware

Types

Integers

This helps the adaptability of C to many kinds of hardware

But it also introduces a certain amount of extra work in porting a program from one kind of hardware to another

Types

Integers

This helps the adaptability of C to many kinds of hardware

But it also introduces a certain amount of extra work in porting a program from one kind of hardware to another

But this is probably a good thing: you don't want to blindly run your program assuming `int`s are 32 bit on some hardware where they are not

Types

Integers

This helps the adaptability of C to many kinds of hardware

But it also introduces a certain amount of extra work in porting a program from one kind of hardware to another

But this is probably a good thing: you don't want to blindly run your program assuming `int`s are 32 bit on some hardware where they are not

E.g., the processor in an embedded system might not support 32 bit integers, but only 16 bit, perhaps

Types

Integers

All the C standard says is that

$$
\begin{aligned}
1 &= \texttt{sizeof(char)} \\
&\leq \texttt{sizeof(short int)} \\
&\leq \texttt{sizeof(int)} \\
&\leq \texttt{sizeof(long int)} \\
&\leq \texttt{sizeof(long long int)}
\end{aligned}
$$

Types

Integers

All the C standard says is that

$$\begin{aligned} 1 &= \texttt{sizeof(char)} \\ &\leq \texttt{sizeof(short int)} \\ &\leq \texttt{sizeof(int)} \\ &\leq \texttt{sizeof(long int)} \\ &\leq \texttt{sizeof(long long int)} \end{aligned}$$

While $\texttt{sizeof(char)} = 1$ this does not mean a `char` is always one byte

Types

Integers

All the C standard says is that

$$\begin{aligned} 1 &= \texttt{sizeof(char)} \\ &\leq \texttt{sizeof(short int)} \\ &\leq \texttt{sizeof(int)} \\ &\leq \texttt{sizeof(long int)} \\ &\leq \texttt{sizeof(long long int)} \end{aligned}$$

While $\texttt{sizeof(char)} = 1$ this does not mean a `char` is always one byte

CPUs with 4 byte `chars` exist

Types

Integers

All the C standard says is that

$$\begin{aligned} 1 &= \texttt{sizeof(char)} \\ &\leq \texttt{sizeof(short int)} \\ &\leq \texttt{sizeof(int)} \\ &\leq \texttt{sizeof(long int)} \\ &\leq \texttt{sizeof(long long int)} \end{aligned}$$

While $\texttt{sizeof(char)} = 1$ this does not mean a `char` is always one byte

CPUs with 4 byte `chars` exist

They have 32-bit `ints` with $\texttt{sizeof(int)} = 1$

Types

Typically, on modern 64 bit PCs we have

Type	bytes
char	1
short int	2
int	4
long int	8
long long int	8

But you should not rely on this in a portable program

Types

Typically, on modern 64 bit PCs we have

Type	bytes
char	1
short int	2
int	4
long int	8
long long int	8

But you should not rely on this in a portable program

Sizes were indeed a problem when people started moving their C programs from 32 bit processors to 64 bit processors