C

The C compiler Clang is reasonably widely available

C

The C compiler Clang is reasonably widely available

Its aim is to give detailed and accurate error and warning messages while producing better code than Gcc

C

The C compiler Clang is reasonably widely available

Its aim is to give detailed and accurate error and warning messages while producing better code than Gcc

And to replace Gcc

C

The C compiler Clang is reasonably widely available

Its aim is to give detailed and accurate error and warning messages while producing better code than Gcc

And to replace Gcc

It is still under heavy development

C

The C compiler Clang is reasonably widely available

Its aim is to give detailed and accurate error and warning messages while producing better code than Gcc

And to replace Gcc

It is still under heavy development

It also uses `-Wall` to show warnings

C

```
% 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

```
% 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
```

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

C

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

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

C

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

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

Loops

```
for (i = 0; i < 10; i++) {
  printf("i is %d\n", i);
}
```

C

Loops

```
for (i = 0; i < 10; i++) {
  printf("i is %d\n", i);
}
```

In fact, `for` is quite general

C

Loops

`for (` *initialisation* `;` *test* `;` *iteration* `) { ... }`

C

Loops

`for (` *initialisation* `;` *test* `;` *iteration* `) { ... }`

Initialisation can be any statement (even empty)

C

Loops

`for (` *initialisation* `;` *test* `;` *iteration* `) { ... }`

Initialisation can be any statement (even empty)

Test can be any expression that returns a true/false; or empty for an infinite loop

C

Loops

`for (` *initialisation* `;` *test* `;` *iteration* `) { ... }`

Initialisation can be any statement (even empty)

Test can be any expression that returns a true/false; or empty for an infinite loop

Iteration can be any statement (even empty)

C

Loops

```
for ( ; wibble(z, w+1); i = i*2) { ... }
```

C

Loops

```
while ( test ) { ... }

do { ... } while ( test );
```

Exercise. What about

```
for ( ; ; ) { ... }
```

C

Exercise. Modify `factorial` to print

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

C

```
#include <stdio.h>

int main(void)
{
  double x, y, z;

  x = 1.0e30;
  y = -1.0e30;
  z = 1.0;

  printf("%g and %g\n", (x + y) + z, x + (y + z));

  return 0;
}
```

Explain the output

Types

C has relatively few built-in types (remember it's low level!), mostly versions of types supported by hardware

Types

C has relatively few built-in types (remember it's low level!), mostly versions of types supported by hardware

Integers of various kinds and sizes:

Types

C has relatively few built-in types (remember it's low level!), mostly versions of types supported 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 C compiler supports all these types, particularly `long long`

Types

Integers

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

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

Types

Integers

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

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

These are all sizes of integer that have proved to be useful in real programs

Types

Integers

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

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

These are all sizes of integer 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 C compiler supports all these types, particularly `long long`

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

These are all sizes of integer 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

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

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

All the C standard says is that

$$\texttt{sizeof(char)} \leq \texttt{sizeof(short)} \leq \texttt{sizeof(int)} \leq \texttt{sizeof(long)}$$

Types

Typically, on modern PCs we have

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

But you should not rely on this in a portable program

Types

There are also *unsigned* variants of the integer types: `unsigned char`, `unsigned int` and so on

Types

There are also *unsigned* variants of the integer types:
`unsigned char`, `unsigned int` and so on

So for an 8-bit `char`, the signed version has range $-128 \ldots 127$.
The unsigned version $0 \ldots 255$

Types

There are also *unsigned* variants of the integer types: `unsigned char`, `unsigned int` and so on

So for an 8-bit `char`, the signed version has range $-128 \ldots 127$. The unsigned version $0 \ldots 255$

Again, C has these types as they are useful in real programs

Types

There are also *unsigned* variants of the integer types: `unsigned char`, `unsigned int` and so on

So for an 8-bit `char`, the signed version has range $-128 \ldots 127$. The unsigned version $0 \ldots 255$

Again, C has these types as they are useful in real programs

Unsigned integers are often used as simple bit patterns rather than integers per se, e.g., in cryptography

Types

Integers

Exercise. `%d` is the `printf` specifier for `signed int`. Find the specifiers for the other integer types

Exercise. Find out what happens to the value when you overflow an `unsigned char` and a `signed char`

Exercise. An unadorned `int` is signed. Find out whether an unadorned `char` has a sign or not

Exercise. Find out the sizes of the integer types on machines you have access to

Exercise. Read up on the bit operators that operate on the individual bits of the integer types

Types

Integers

So `char` is an integer type?

Types

Integers

So `char` is an integer type?

Correct: C does not have character as a separate type like some other languages

Types

Integers

So `char` is an integer type?

Correct: C does not have character as a separate type like some other languages

We shall see in a moment that C does not have a string type either!

Types

Integers

So `char` is an integer type?

Correct: C does not have character as a separate type like some other languages

We shall see in a moment that C does not have a string type either!

In fact, it would probably be better to think of `char` as a byte as many compilers have an 8-bit `char`

Types

Integers

So `char` is an integer type?

Correct: C does not have character as a separate type like some other languages

We shall see in a moment that C does not have a string type either!

In fact, it would probably be better to think of `char` as a byte as many compilers have an 8-bit `char`

(Aside: technically a “byte” is not necessarily 8 bits; use the word “octet” to mean precisely 8 bits)

Types

Integers

So `char` is an integer type?

Correct: C does not have character as a separate type like some other languages

We shall see in a moment that C does not have a string type either!

In fact, it would probably be better to think of `char` as a byte as many compilers have an 8-bit `char`

(Aside: technically a “byte” is not necessarily 8 bits; use the word “octet” to mean precisely 8 bits)

But the name “`char`” indicates a popular use of this type: characters encoded as ASCII integers

Types

Integers

The syntax for characters is single quotes: `'A'` is the integer value that encodes for the character “A”

Types

Integers

The syntax for characters is single quotes: `'A'` is the integer value that encodes for the character “A”

Again: `'A'` is a way of writing an integer value, typically 65 when using the usual ASCII encoding; the two ways of writing sixty-five are then more-or-less interchangeable

Types

Integers

The syntax for characters is single quotes: `'A'` is the integer value that encodes for the character “A”

Again: `'A'` is a way of writing an integer value, typically 65 when using the usual ASCII encoding; the two ways of writing sixty-five are then more-or-less interchangeable

```
char c = 'Z' - 'A' + 1;
```
is valid C

Types

Integers

The syntax for characters is single quotes: `'A'` is the integer value that encodes for the character “A”

Again: `'A'` is a way of writing an integer value, typically 65 when using the usual ASCII encoding; the two ways of writing sixty-five are then more-or-less interchangeable

```
char c = 'Z' - 'A' + 1;
```
is valid C

We use the single quote syntax as it is easier (we don’t have to look up the relevant value) and it is portable: not everyone uses ASCII

Types

Integers

Exercise. Find out which character encoding your machine uses

Exercise. Is `'A' + 1` always `'B'`?

Types

Floating Point

C has a few floating point types

Types

Floating Point

C has a few floating point types

- `float` also called “single precision float”
- `double` also called “double precision float”
- `long double` also called “quad precision float”

Types

Floating Point

C has a few floating point types

- `float` also called “single precision float”
- `double` also called “double precision float”
- `long double` also called “quad precision float”

These overwhelmingly conform to a particular standard for floating point representations, namely IEEE 754

Types

Floating Point

C has a few floating point types

- `float` also called “single precision float”
- `double` also called “double precision float”
- `long double` also called “quad precision float”

These overwhelmingly conform to a particular standard for floating point representations, namely IEEE 754

Many machines support `double` in hardware, so this is the “natural” size in programs: but not always

Types

Floating Point

It turns out that the flexibility of having explicitly undefined integers works against you when you want to compute with floating point, so everybody (pretty much all hardware) uses IEEE 754

Type	bytes
float	4
double	8
long double	16

Types

Floating Point

It turns out that the flexibility of having explicitly undefined integers works against you when you want to compute with floating point, so everybody (pretty much all hardware) uses IEEE 754

Type	bytes
float	4
double	8
long double	16

That said, there is a significant class of hardware out there that does it differently. . .

Types

Floating Point

Most general-purpose hardware supports `double` (64 bit) floats with range approximately $\pm 10^{-323}$ to $\pm 10^{308}$

Types

Floating Point

Most general-purpose hardware supports `double` (64 bit) floats with range approximately $\pm 10^{-323}$ to $\pm 10^{308}$

IEEE 754 also has many other curious features, such as support for infinities and “not a numbers”

Types

Floating Point

Most general-purpose hardware supports `double` (64 bit) floats with range approximately $\pm 10^{-323}$ to $\pm 10^{308}$

IEEE 754 also has many other curious features, such as support for infinities and "not a numbers"

These have their expected behaviours, e.g., $1.0/0.0$ returns infinity; `sqrt(-1.0)` returns a NaN

Types

Floating Point

Most general-purpose hardware supports `double` (64 bit) floats with range approximately $\pm 10^{-323}$ to $\pm 10^{308}$

IEEE 754 also has many other curious features, such as support for infinities and "not a numbers"

These have their expected behaviours, e.g., $1.0/0.0$ returns infinity; `sqrt(-1.0)` returns a NaN

Also, there is a *signed zero*, namely ± 0.0. To understand why all these things are desirable you should attend a course on numerical analysis

Types

Floating Point

To write a `double`, use the usual `1.234` and `-2.3e-5` formats

Types

Floating Point

To write a `double`, use the usual `1.234` and `-2.3e-5` formats

For single precision (32 bit) floats, append an `f`, e.g, `3.141f`.
An unadorned `3.141` indicates a `double`

Types

Floating Point

To write a `double`, use the usual `1.234` and `-2.3e-5` formats

For single precision (32 bit) floats, append an `f`, e.g, `3.141f`. An unadorned `3.141` indicates a `double`

There is little use for single precision floats in modern hardware with built-in `double`s: some don't even support `float`s natively

Types

Floating Point

To write a `double`, use the usual `1.234` and `-2.3e-5` formats

For single precision (32 bit) floats, append an `f`, e.g, `3.141f`. An unadorned `3.141` indicates a `double`

There is little use for single precision floats in modern hardware with built-in `double`s: some don't even support `float`s natively

So in `float f = 1.0f; f = f * 2.0f;` the single floats `1.0f` and `2.0f` would be *widened* automatically to `double`; the multiplication computed in double precision; the result then *truncated* to fit back into `f`

Types

Floating Point

This could well actually be slower than plain, hardware supported, double precision computation all the way through

Types

Floating Point

This could well actually be slower than plain, hardware supported, double precision computation all the way through

The only reasons to use `float` are (a) when you are short on space, or (b) the hardware does not support `double` well or at all (embedded chips, graphics cards, etc.)

Types

Floating Point

This could well actually be slower than plain, hardware supported, double precision computation all the way through

The only reasons to use `float` are (a) when you are short on space, or (b) the hardware does not support `double` well or at all (embedded chips, graphics cards, etc.)

The `printf` specifier for both `float` and `double` is `%f`

Types

Floating Point

This could well actually be slower than plain, hardware supported, double precision computation all the way through

The only reasons to use `float` are (a) when you are short on space, or (b) the hardware does not support `double` well or at all (embedded chips, graphics cards, etc.)

The `printf` specifier for both `float` and `double` is `%f`

There is no separate specifier for `float` as any float in a `printf` will be automatically converted to a `double`

Types

A note on mixing values of different types: C (in common with many other languages) has a raft of automatic *coercions* of types of values

Types

A note on mixing values of different types: C (in common with many other languages) has a raft of automatic *coercions* of types of values

In `double x; ... x + 1` the integer `1` is automatically coerced to `double 1.0` (“floating point contagion”)

Types

A note on mixing values of different types: C (in common with many other languages) has a raft of automatic *coercions* of types of values

In `double x; ... x + 1` the integer `1` is automatically coerced to `double 1.0` ("floating point contagion")

In `char c; int n; ... n+c` the `c` is automatically coerced to an `int`

Types

A note on mixing values of different types: C (in common with many other languages) has a raft of automatic *coercions* of types of values

In `double x; ... x + 1` the integer `1` is automatically coerced to `double 1.0` ("floating point contagion")

In `char c; int n; ... n+c` the `c` is automatically coerced to an `int`

Usually it does what you want, but you should always look at mixed-type expressions carefully

Types

Exercise. What's happening here?

```
int n = 1, m = 2;
double x = n/m;

printf("x is %g\n", x);
```

Types

Floating Point

Summary: stick to `double`

Types

Floating Point

Summary: stick to `double`

The newest C compilers also support a *complex type*, e.g.,

```
#include <complex.h>
...
complex c = 5.0 + 3.0 * I;
c = c + 1.0;
```

The `double 1.0` will be automatically coerced (widened) to a `complex`

Types

Floating Point

Summary: stick to `double`

The newest C compilers also support a *complex type*, e.g.,

```
#include <complex.h>
...
complex c = 5.0 + 3.0 * I;
c = c + 1.0;
```

The `double 1.0` will be automatically coerced (widened) to a `complex`

And they also support *wide characters*, to support character sets from global languages

Types

Floating Point

Exercise. Write a program that evaluates

$$\sum_{n=1}^{1000000} \frac{1}{n} - \frac{1}{n+1}$$

Compare the results using `float` and `double`

Types

Boolean

C does not have a separate Boolean type

Types

Boolean

C does not have a separate Boolean type

Integer 0 plays the role of false, while any non-zero integer is interpreted as true

Types

Boolean

C does not have a separate Boolean type

Integer 0 plays the role of false, while any non-zero integer is interpreted as true

```
int compare(double a, double b)
{
  if (a > b) return 1;
  return 0;
}
```

Types

Boolean

C does not have a separate Boolean type

Integer 0 plays the role of false, while any non-zero integer is interpreted as true

```
int compare(double a, double b)
{
  if (a > b) return 1;
  return 0;
}
```

Though this would not be regarded as a natural C

Types

Boolean

The expression "`a > b`" is just that: an expression

Types

Boolean

The expression "`a > b`" is just that: an expression

Just like "`a + b`" it returns a value, false or true, i.e., zero or non-zero

Types

Boolean

The expression "`a > b`" is just that: an expression

Just like "`a + b`" it returns a value, false or true, i.e., zero or non-zero

More idiomatic C would be:

```
int compare(double a, double b)
{
  return a > b;
}
```

Types

Boolean

You can even write `n = 5 + (a > b);` but that would be questionable style

Types

Boolean

You can even write `n = 5 + (a > b);` but that would be questionable style

Close reading of the C standard reveals that such Boolean expressions should always return 1 or 0; i.e., 1 is specified as the canonical true value

Types

Boolean

You can even write `n = 5 + (a > b);` but that would be questionable style

Close reading of the C standard reveals that such Boolean expressions should always return 1 or 0; i.e., 1 is specified as the canonical true value

So `n` will be 5 or 6

Types

Boolean

You can even write `n = 5 + (a > b);` but that would be questionable style

Close reading of the C standard reveals that such Boolean expressions should always return 1 or 0; i.e., 1 is specified as the canonical true value

So `n` will be 5 or 6

But, again, only mix expressions like this if you really understand what you are doing

Types

Boolean

The equality test is ==, not =

Types

Boolean

The equality test is ==, not =

A common source of bugs is to write

```
if (a = 2) ...
```

rather than

```
if (a == 2) ...
```

Types

Boolean

The equality test is ==, not =

A common source of bugs is to write

```
if (a = 2) ...
```

rather than

```
if (a == 2) ...
```

The first is valid C: it assigns 2 to `a`, and then the expression "`a = 2`" returns the value `2`, i.e., true in a Boolean context

Types

Boolean

Exercise. What is the result of
`printf("%d\n", compare(2, 1));`

Exercise. Read up on the various Boolean connectives `&&`, `||` etc.

Exercise. Compare the Boolean connectives with the *bitwise operators* `&`, `|` etc.

Exercise. Read up on the `?:` operator

Types

Boolean

Look at what your compiler says about

```
#include <stdio.h>

int main(void)
{
  int s = 1;

  if (s = 2) printf("hi\n");
  else printf("lo\n");

  return 0;
}
```