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`,

	range
signed	$-128 \ldots 127$
unsigned	$0 \ldots 255$

Types

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

So for an 8-bit `char`,

	range
signed	$-128 \ldots 127$
unsigned	$0 \ldots 255$

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`,

	range
signed	$-128 \ldots 127$
unsigned	$0 \ldots 255$

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

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

So for an 8-bit `char`,

	range
signed	$-128 \ldots 127$
unsigned	$0 \ldots 255$

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

There is a `signed` keyword if you want to be explicit: e.g., `signed char` and `signed int`

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 operators that operate on the bits of the integer types: `&`, |, <<, >>, etc.

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

Types

Integers

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

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

To reiterate: `'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 C syntax for characters is single quotes: `'A'` is the integer value that encodes for the character "A"

To reiterate: `'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

Exercise. Find out which character encoding your machine uses

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

Exercise. Is `'A'` < `'B'` always true?

Exercise. What about `'A'` < `'a'` or `'a'` < `'A'`?

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` is sometimes supported

Types
Floating Point

C has a few floating point types

- `float` also called “single precision float”
- `double` also called “double precision float”
- `long double` is sometimes supported

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` is sometimes supported

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 sizes works against you when you want to do numerical analysis with floating point, so 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 sizes works against you when you want to do numerical analysis with floating point, so 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, e.g., *fixed-point* arithmetic

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 number”s

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 number”s

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 number”s

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

Exercise. Look up the documentation on the functions `atan` and `atan2`

Exercise. Read up on IEEE 754 features

Types

Floating Point

To write a `double` in C, use the familiar `1.234` and `-2.3e-5` formats

Types
Floating Point

To write a `double` in C, use the familiar `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` (64 bit)

Types

Floating Point

To write a `double` in C, use the familiar `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` (64 bit)

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

Types
Floating Point

So in those kinds of machines

```
float f = 1.0f * 2.0f
```

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

Types

Floating Point

So in those kinds of machines

```
float f = 1.0f * 2.0f
```

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

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

Types

Floating Point

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

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

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` argument will be automatically widened to a `double` before being passed into `printf`

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 (widened) 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 (widened) to an `int`

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

Types

Note there is a significant difference between coercion of ints to doubles and coercion of chars to ints

Types

Note there is a significant difference between coercion of ints to doubles and coercion of chars to ints

Widening a char to an int just takes the bit pattern that represents the char and puts it in a bigger, int-sized box

Types

Note there is a significant difference between coercion of ints to doubles and coercion of chars to ints

Widening a char to an int just takes the bit pattern that represents the char and puts it in a bigger, int-sized box

The bit pattern is not changed, just extended

Types

Coercing an int to a double takes the bit pattern that represents the int (2s complement, perhaps), calculates the bit-pattern that represents the closest numerically equivalent floating point (IEEE, probably) and returns that

Types

Coercing an int to a double takes the bit pattern that represents the int (2s complement, perhaps), calculates the bit-pattern that represents the closest numerically equivalent floating point (IEEE, probably) and returns that

This will an entirely different bit pattern

Types

Coercing an int to a double takes the bit pattern that represents the int (2s complement, perhaps), calculates the bit-pattern that represents the closest numerically equivalent floating point (IEEE, probably) and returns that

This will an entirely different bit pattern

Usually you don't have to care that this is happening, but you should be aware that it is

Types

On some classes of hardware, this is actually a very expensive (slow) operation!

Thus for `double x;`

```
x = 1;
```

could be a lot slower than

```
x = 1.0;
```

Types

On some classes of hardware, this is actually a very expensive (slow) operation!

Thus for `double x;`

```
x = 1;
```

could be a lot slower than

```
x = 1.0;
```

Though this is relatively rare

Types

Exercise. Assuming standard IEEE and 2-s complement representations:

```
long int n = 42;
double x = n;
```

What is the bit pattern stored in the 8-byte integer `n`?

What is the bit pattern stored in the 8-byte float `x`?

Types

Exercise. What's happening here?

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

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

Exercise. Some compilers have flags to warn about automatic type coercions. Look this up

Summary: stick to `double` for floating point

Types

Floating Point

Summary: stick to `double` for floating point

When you hear the phrase "floating point" the speaker usually means "double precision floating point"

Types
Floating Point

Summary: stick to `double` for floating point

When you hear the phrase “floating point” the speaker usually means “double precision floating point”

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

Exercise. Think about the difference between
`sqrt(-1.0)`
and
`csqrt(-1.0)`
where `csqrt` is the complex square root function

Types

Floating Point

Exercise. Think about the difference between
`sqrt(-1.0)`
and
`csqrt(-1.0)`
where `csqrt` is the complex square root function

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

Exercise. Let $a = 1.0 \times 10^8$, $b = -1.0 \times 10^8$ and $c = 1.0$. Write code to evaluate and print the result of

$$(a + b) + c$$

and

$$a + (b + c)$$

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 bigger(double a, double b)
{
  if (a > b) return 1;
  return 0;
}
...
if (bigger(x+1.0, y)) ...
```

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 bigger(double a, double b)
{
  if (a > b) return 1;
  return 0;
}
...
if (bigger(x+1.0, y)) ...
```

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 the expression “`a + b`” returns a value, “`a > b`” also 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 the expression “`a + b`” returns a value, “`a > b`” also returns a value, false or true, i.e., zero or non-zero

More idiomatic C would be:

```
int bigger(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

The C standard requires 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

The C standard requires 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

The C standard requires 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

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

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

Exercise. And the shift operators >> and <<. Particularly with regard to signed and unsigned integers

Exercise. Read up on the `?:` operator

Exercise. What happens with `n = 1 + (m = 2)` ?

Types

Boolean

Exercise. Look at what your compiler says about

```
#include <stdio.h>

int main(void)
{
  int s = 0;

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

  return 0;
}
```

Types

Arrays

Given a type in C, we can have an array of things of that type

Types

Arrays

Given a type in C, we can have an array of things of that type

```
int a[5];
double b[1024];
```

Types

Arrays

Given a type in C, we can have an array of things of that type

```
int a[5];
double b[1024];
```

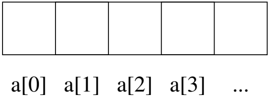
The elements are referenced as you might expect

```
int i;
...
for (i = 0; i < 1024; i++) {
  b[i] += 1.0;
}
```

Indexed from 0 to length − 1

Types

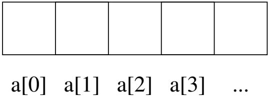
Arrays



Arrays are simply laid out in memory, with successive values next to each other (contiguous) in memory

Types

Arrays



Arrays are simply laid out in memory, with successive values next to each other (contiguous) in memory

The C standard specifies this layout, and this will become important later

Types

Arrays

Arrays of things are a type, so we can have arrays of them

Arrays of things are a type, so we can have arrays of them

So `char d[6][7];` is an array of 6 items; each item is an array of 7 `char`s

Types
Arrays

Arrays of things are a type, so we can have arrays of them

So `char d[6][7];` is an array of 6 items; each item is an array of 7 `char`s

This is how C provides two (and higher) dimensional arrays: as arrays of arrays

Types

Arrays

Arrays of things are a type, so we can have arrays of them

So `char d[6][7];` is an array of 6 items; each item is an array of 7 `chars`

This is how C provides two (and higher) dimensional arrays: as arrays of arrays

But, also, `d[3]` is a valid thing to write: it refers to the 4th array of characters

Types

Arrays

Arrays of things are a type, so we can have arrays of them

So `char d[6][7];` is an array of 6 items; each item is an array of 7 `char`s

This is how C provides two (and higher) dimensional arrays: as arrays of arrays

But, also, `d[3]` is a valid thing to write: it refers to the 4th array of characters

So `d[3][0]`, `d[3][1]`, `...`, `d[3][6]`, are the 7 characters in that array

Arrays of things are a type, so we can have arrays of them

So `char d[6][7];` is an array of 6 items; each item is an array of 7 `chars`

This is how C provides two (and higher) dimensional arrays: as arrays of arrays

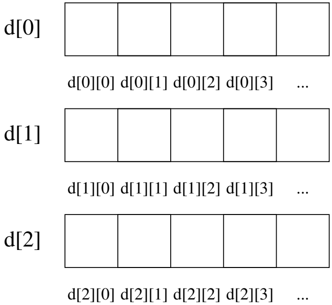
But, also, `d[3]` is a valid thing to write: it refers to the 4th array of characters

So `d[3][0]`, `d[3][1]`, `...`, `d[3][6]`, are the 7 characters in that array

Maybe writing `(d[3])[0]` is clearer?

Types

Arrays



Higher dimensional arrays

Types

Arrays

```
void fill(int arr[], int n)
{
  int i;

  for (i = 0; i < n; i++) {
    arr[i] = 99;
  }
}

...

int a[5], d[6][7];

fill(a, 5);
fill(d[3], 7);
```

Exercise. What about

```
fill(d, 6);
```

Types

Arrays

So:

So:

- Arrays can be passed as arguments to functions

So:

- Arrays can be passed as arguments to functions
- The size of the array need not be specified in the function definition (for simple, 1D arrays)

Types

Arrays

So:

- Arrays can be passed as arguments to functions
- The size of the array need not be specified in the function definition (for simple, 1D arrays)
- An array does not “know its own size”. That information has to be given separately, if needed. This is a common source of bugs

Types

Arrays

Normally, C **does not check** for correct access to arrays

Types

Arrays

Normally, C **does not check** for correct access to arrays

```
int a[5];
...
a[10] = 42;
```

may well compile without error, or even warning

Types

Arrays

Normally, C **does not check** for correct access to arrays

```
int a[5];
...
a[10] = 42;
```

may well compile without error, or even warning

The program might even run, not report an error and return the correct answer

Types

Arrays

Normally, C **does not check** for correct access to arrays

```
int a[5];
...
a[10] = 42;
```

may well compile without error, or even warning

The program might even run, not report an error and return the correct answer

It might run, not report an error and return the wrong answer

Types

Arrays

Normally, C **does not check** for correct access to arrays

```
int a[5];
...
a[10] = 42;
```

may well compile without error, or even warning

The program might even run, not report an error and return the correct answer

It might run, not report an error and return the wrong answer

It might run and return the same answer every time

Types

Arrays

Normally, C **does not check** for correct access to arrays

```
int a[5];
...
a[10] = 42;
```

may well compile without error, or even warning

The program might even run, not report an error and return the correct answer

It might run, not report an error and return the wrong answer

It might run and return the same answer every time

It might run and return a different answer some times

Types

Arrays

Normally, C **does not check** for correct access to arrays

```
int a[5];
...
a[10] = 42;
```

may well compile without error, or even warning

The program might even run, not report an error and return the correct answer

It might run, not report an error and return the wrong answer

It might run and return the same answer every time

It might run and return a different answer some times

It might run and crash

This is one of C's chosen trade-offs

Types

Arrays

This is one of C's chosen trade-offs

More speed for less checking and safety

This is one of C’s chosen trade-offs

More speed for less checking and safety

C allows the programmer to do all kinds of weird stuff, often without warning

This is one of C's chosen trade-offs

More speed for less checking and safety

C allows the programmer to do all kinds of weird stuff, often without warning

This is good for good programmers; bad for bad programmers

Exercise. Implement a function which, given an array of integers fills that array with the squares of 0, 1, 2, and so on

Exercise. Implement a function which, given an array of integers, returns the sum of the values in the array

Exercise. Implement the Sieve of Eratosthenes to find primes