Types

Arrays

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

Types
Arrays

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

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

Types

Arrays

Given a type in C, we can have arrays 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 of things are a type, so we can have arrays of them

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

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 `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

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

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

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

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:

Types

Arrays

So:

- Arrays can be passed as arguments to functions

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)

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 crash

Types

Arrays

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

Types
Arrays

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

Types

Arrays

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

Types
Strings

There is no string type in C

Types
Strings

There is no string type in C

There *are* arrays of `char`

Types

Strings

There is no string type in C

There *are* arrays of `char`

```
char str[] = "hello world";
```

C is clever enough to work out the size of the array needed here, to save you a bit of counting

Types

Strings

There is no string type in C

There *are* arrays of `char`

```
char str[] = "hello world";
```

C is clever enough to work out the size of the array needed here, to save you a bit of counting

Then `str[4]` is the character `'o'`

Types

Strings

In `printf` use `%s` for strings

```
printf("str is '%s'\n", str);
```

And `%c` for `char`s

```
printf("char is '%c'\n", str[4]);
```

Types

Strings

There is nothing special about strings that distinguishes them from other arrays, apart from having a special syntax using quotes

Types

Strings

There is nothing special about strings that distinguishes them from other arrays, apart from having a special syntax using quotes

```
char str[] = { 'h', 'e', 'l', 'l', 'o', ' ', 'w',
'o', 'r', 'l', 'd' };
```

Types
Strings

There is nothing special about strings that distinguishes them from other arrays, apart from having a special syntax using quotes

```
char str[] = { 'h', 'e', 'l', 'l', 'o', ' ', 'w',
'o', 'r', 'l', 'd' };
```

There are two reasons why you wouldn't normally write code like this:

- it's easier to use normal quoted string syntax
- this code is semantically incorrect

Types

Strings

Just like other arrays, C does not store the length of a string in the string, only the characters

Types
Strings

Just like other arrays, C does not store the length of a string in the string, only the characters

So how can it tell how long is the string in
`printf("str is '%s'\n", str);`?

Types

Strings

Just like other arrays, C does not store the length of a string in the string, only the characters

So how can it tell how long is the string in
`printf("str is '%s'\n", str);`?

All it has is an array of characters of some undetermined size

Types

Strings

Just like other arrays, C does not store the length of a string in the string, only the characters

So how can it tell how long is the string in `printf("str is '%s'\n", str);`?

All it has is an array of characters of some undetermined size

Stored as a sequence of bytes in memory: we need some way to mark the end of the string

Types

Strings

Just like other arrays, C does not store the length of a string in the string, only the characters

So how can it tell how long is the string in `printf("str is '%s'\n", str);`?

All it has is an array of characters of some undetermined size

Stored as a sequence of bytes in memory: we need some way to mark the end of the string

Thus, in C, all strings are conventionally terminated by a (character value/byte) 0

Types
Strings

```
char str[] = { 'h', 'e', 'l', 'l', 'o', ' ', 'w',
'o', 'r', 'l', 'd', 0 };
```

is the correct version of the simpler

```
char str[] = "hello world"
```

Types

Strings

```
char str[] = { 'h', 'e', 'l', 'l', 'o', ' ', 'w',
'o', 'r', 'l', 'd', 0 };
```

is the correct version of the simpler

```
char str[] = "hello world"
```

So `sizeof("hello world")` is 12 bytes, including the terminating 0

Types

Strings

```
char str[] = { 'h', 'e', 'l', 'l', 'o', ' ', 'w',
'o', 'r', 'l', 'd', 0 };
```

is the correct version of the simpler

```
char str[] = "hello world"
```

So `sizeof("hello world")` is 12 bytes, including the terminating 0

This is another favourite source of bugs!

Types

Strings

```
char str[] = { 'h', 'e', 'l', 'l', 'o', ' ', 'w',
'o', 'r', 'l', 'd', 0 };
```

is the correct version of the simpler

```
char str[] = "hello world"
```

So `sizeof("hello world")` is 12 bytes, including the terminating 0

This is another favourite source of bugs!

If you stick to simple uses of strings, this all just works without you having to think

Types
Strings

Exercise. Look up the ASCII encoding for characters

Exercise. Characters really are integers. What about the following?

```
char message[] = { 104, 101, 108, 108, 111, 32, 119,
  111, 114, 108, 100, 0 };
```

Exercise. And what about

```
printf("A has value %d\n", 'A');
printf("A has value %c\n", 'A');
```

Types
Strings

Since strings are not a proper type in C it does not have built-in operations on strings, e.g., concatenate

Types
Strings

Since strings are not a proper type in C it does not have built-in operations on strings, e.g., concatenate

This is provided by library functions, if you need them. They all assume strings are zero-terminated

Types

Strings

Since strings are not a proper type in C it does not have built-in operations on strings, e.g., concatenate

This is provided by library functions, if you need them. They all assume strings are zero-terminated

Exercise. Look up the various functions that operate on strings, e.g., `strlen`, `strcpy`, `strcat`, `strcmp` and lots more

Types

Structures

C has a simple *structure* type constructor, used when we need to manage more complicated combinations of values

Types

Structures

C has a simple *structure* type constructor, used when we need to manage more complicated combinations of values

```
struct rational {
  int num, den;
};

...

struct rational r;
r.num = 1;
r.den = 2;
```

Types

Structures

- Don't forget the `;` at the end of the declaration

Types

Structures

- Don't forget the `;` at the end of the declaration
- They may look like Java classes, but they are not

Types
Structures

- Don't forget the `;` at the end of the declaration
- They may look like Java classes, but they are not
- The type name is "`struct rational`", always including the word "`struct`"

Types

Structures

- Don't forget the `;` at the end of the declaration
- They may look like Java classes, but they are not
- The type name is "`struct rational`", always including the word "`struct`"
- The elements of the struct are accessed using the dot notation

Types

Structures

- Don't forget the `;` at the end of the declaration
- They may look like Java classes, but they are not
- The type name is "`struct rational`", always including the word "`struct`"
- The elements of the struct are accessed using the dot notation
- `r` is *not* an object in the OO sense

Types

Structures

- Don't forget the `;` at the end of the declaration
- They may look like Java classes, but they are not
- The type name is "`struct rational`", always including the word "`struct`"
- The elements of the struct are accessed using the dot notation
- `r` is *not* an object in the OO sense
- There are no classes, no objects, no methods

Types

Structures

- Don't forget the `;` at the end of the declaration
- They may look like Java classes, but they are not
- The type name is "`struct rational`", always including the word "`struct`"
- The elements of the struct are accessed using the dot notation
- `r` is *not* an object in the OO sense
- There are no classes, no objects, no methods
- The declaration can only contain names of values, as **there are no methods in C**

Types

Structures

Structure types are just like the in-built types

Types

Structures

Structure types are just like the in-built types

So we can have arrays of structs:

```
struct rational numbers[10];
```

Types

Structures

Structure types are just like the in-built types

So we can have arrays of structs:

```
struct rational numbers[10];
```

So `numbers[7].num`

Types

Structures

Structure types are just like the in-built types

So we can have arrays of structs:

```
struct rational numbers[10];
```

So `numbers[7].num`

We can declare structs containing arrays

```
struct numb { int nums[10]; int dens[10]; }
```

Types

Structures

Structure types are just like the in-built types

So we can have arrays of structs:

```
struct rational numbers[10];
```

So `numbers[7].num`

We can declare structs containing arrays

```
struct numb { int nums[10]; int dens[10]; }
```

Then

```
struct numb n;
n.nums[7] = 42;
```

Structs of structs, and so on

```
struct inner {
  double first[10];
  char rest;
};

struct complicated {
  int sign;
  struct rational r;
  struct inner blob;
};
...
struct complicated c;
c.sign = -1;
c.r.num = 5;
c.blob.first[3] = 7.0;
```

We can also declare structs “on the fly” as we are using them

```
struct complicated {
  int sign;
  struct rational r;
  struct inner {
    double first[10];
    char rest;
  } blob;
};
...
struct complicated c;
c.sign = -1;
c.r.num = 5;
c.blob.first[3] = 7.0;
```

Types

Exercise. Read up on `union` types

Exercise. Read up on `typedef`, a convenient way of abbreviating type names

Pointers

We now turn to one of the features of C that (a) some people find difficult, and (b) makes C so useful: pointers

Pointers

We now turn to one of the features of C that (a) some people find difficult, and (b) makes C so useful: pointers

We start by reviewing the way memory is laid out in hardware

Pointers

We now turn to one of the features of C that (a) some people find difficult, and (b) makes C so useful: pointers

We start by reviewing the way memory is laid out in hardware

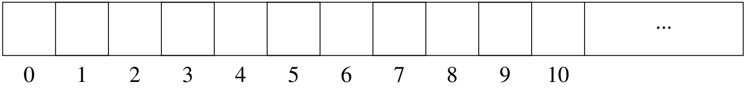
Recall that (thanks to the universal adoption of von Neumann's model) memory can be regarded as a big array of bytes; conventionally numbered from 0 upwards

Pointers

We now turn to one of the features of C that (a) some people find difficult, and (b) makes C so useful: pointers

We start by reviewing the way memory is laid out in hardware

Recall that (thanks to the universal adoption of von Neumann's model) memory can be regarded as a big array of bytes; conventionally numbered from 0 upwards



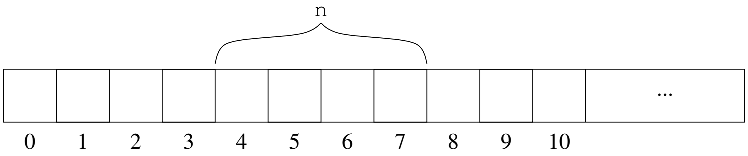
Pointers

When a program is compiled, variables are mapped in some useful way to memory location by the compiler

Pointers

When a program is compiled, variables are mapped in some useful way to memory location by the compiler

So if we have a (4 byte) integer `n` in our code, the compiler might choose to place it at memory address 4 (a very unlikely place in real systems)



Pointers

Then every access of `n` in our code becomes a read or write of bytes 4–7 of memory

Pointers

Then every access of `n` in our code becomes a read or write of bytes 4–7 of memory

We say byte 4 is the *address* of the variable `n`

Pointers

Then every access of `n` in our code becomes a read or write of bytes 4–7 of memory

We say byte 4 is the *address* of the variable `n`

It's where the variable lives in memory

Pointers

C gives us access to these addresses in our program

Pointers

C gives us access to these addresses in our program

Other languages might not do this, preferring to hide these details from the programmer

Pointers

C gives us access to these addresses in our program

Other languages might not do this, preferring to hide these details from the programmer

But for low-level programs that manipulate bits and bytes of memory this is just what they need

Pointers

C gives us access to these addresses in our program

Other languages might not do this, preferring to hide these details from the programmer

But for low-level programs that manipulate bits and bytes of memory this is just what they need

To get the address of a variable use the `&` operator

Pointers

```
#include <stdio.h>

int main(void)
{
  int n = 1234;

  printf("n has value %d and address %p\n", n, &n);

  return 0;
}
```

Pointers

```
#include <stdio.h>

int main(void)
{
  int n = 1234;

  printf("n has value %d and address %p\n", n, &n);

  return 0;
}
```

Produces

```
n has value 1234 and address 0x7fff251f6d5c
```

Pointers

Note the difference between the *value* of `n` and the *address* of `n`

Pointers

Note the difference between the *value* of `n` and the *address* of `n`

The value of `n` will always be 1234; the address will be different on different machines, different on different compilers, possibly different on different runs on the same machine

Pointers

Note the difference between the *value* of `n` and the *address* of `n`

The value of `n` will always be 1234; the address will be different on different machines, different on different compilers, possibly different on different runs on the same machine

It all depends on where in memory `n` happens to be placed

Pointers

So addresses are just integers; the `%p` in `printf` prints addresses in hexadecimal, as that is often useful to the programmer

Pointers

So addresses are just integers; the `%p` in `printf` prints addresses in hexadecimal, as that is often useful to the programmer

Exercise. Compare `%x` with `%p`

Pointers

So addresses are just integers; the `%p` in `printf` prints addresses in hexadecimal, as that is often useful to the programmer

Exercise. Compare `%x` with `%p`

Addresses are first-class values in C: this means you can use and manipulate them just like any other values (like integers, doubles, etc.)

Pointers

So addresses are just integers; the `%p` in `printf` prints addresses in hexadecimal, as that is often useful to the programmer

Exercise. Compare `%x` with `%p`

Addresses are first-class values in C: this means you can use and manipulate them just like any other values (like integers, doubles, etc.)

They are just integers, after all

Pointers

So addresses are just integers; the `%p` in `printf` prints addresses in hexadecimal, as that is often useful to the programmer

Exercise. Compare `%x` with `%p`

Addresses are first-class values in C: this means you can use and manipulate them just like any other values (like integers, doubles, etc.)

They are just integers, after all

Variables that hold addresses are called *pointer variables*

Pointers

So a pointer variable contains a simple integer (the address), but to make things work nicely, C distinguishes between pointers and integers, and also between pointers to different types

Pointers

So a pointer variable contains a simple integer (the address), but to make things work nicely, C distinguishes between pointers and integers, and also between pointers to different types

So a pointer to an integer is treated as different to a pointer to a double

Pointers

So a pointer variable contains a simple integer (the address), but to make things work nicely, C distinguishes between pointers and integers, and also between pointers to different types

So a pointer to an integer is treated as different to a pointer to a double

And both are different from a ordinary integer

Pointers

So a pointer variable contains a simple integer (the address), but to make things work nicely, C distinguishes between pointers and integers, and also between pointers to different types

So a pointer to an integer is treated as different to a pointer to a double

And both are different from a ordinary integer

This is a bit subtle: they are all simple integers underneath; it's just how the compiler manipulates those integers that will be different for different types

Pointers

So the *interpretation* of that pointer integer is what is important

Pointers

So the *interpretation* of that pointer integer is what is important

This is to make later manipulations much more convenient

Pointers

So the *interpretation* of that pointer integer is what is important

This is to make later manipulations much more convenient

Of course, memory doesn't "know" what kind of data is being stored at a particular address; memory is just a bunch of bytes

Pointers

So the *interpretation* of that pointer integer is what is important

This is to make later manipulations much more convenient

Of course, memory doesn't "know" what kind of data is being stored at a particular address; memory is just a bunch of bytes

At one point the program might store an integer at address 4; later it might store a double there

Pointers

So the *interpretation* of that pointer integer is what is important

This is to make later manipulations much more convenient

Of course, memory doesn't "know" what kind of data is being stored at a particular address; memory is just a bunch of bytes

At one point the program might store an integer at address 4; later it might store a double there

It is up to the program to interpret the bits at a given address in whatever way it wants

Pointers

We can declare pointer variables

```
int n;
int *pn;
pn = &n;
```

The * is read as "pointer to"; the variable `pn` has type "pointer to `int`"

Pointers

Convention

Note: the convention is to write `int *pn;` rather than `int* pn;`

Pointers

Convention

Note: the convention is to write `int *pn;` rather than `int* pn;`

Both are treated as exactly the same by the compiler

Pointers

Convention

Note: the convention is to write `int *pn;` rather than `int* pn;`

Both are treated as exactly the same by the compiler

The latter is read as "`pn` has type pointer to `int`" or "`pn` is an `int` pointer"

Pointers

Convention

Note: the convention is to write `int *pn;` rather than `int* pn;`

Both are treated as exactly the same by the compiler

The latter is read as "`pn` has type pointer to `int`" or "`pn` is an `int` pointer"

The reason for this slightly awkward convention is that we can declare

```
int n, *pn;
```

for an `int` `n` and a pointer to `int` `pn`

Pointers

Convention

Note: the convention is to write `int *pn;` rather than `int* pn;`

Both are treated as exactly the same by the compiler

The latter is read as "`pn` has type pointer to `int`" or "`pn` is an `int` pointer"

The reason for this slightly awkward convention is that we can declare

```
int n, *pn;
```

for an `int` `n` and a pointer to `int` `pn`

You can read the above as "`n` is an `int` and `pn` is an `int` pointer"

Pointers

Convention

Exercise. What are the types of the variables in the following?

```
int* a, b;
```