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";
```

This declares an array and initialises it

Types

Strings

There is no string type in C

There *are* arrays of `char`

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

This declares an array and initialises it

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";
```

This declares an array and initialises it

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

There is no string type in C

There *are* arrays of `char`

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

This declares an array and initialises it

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'`

Which, of course, is just some integer value

In `printf` use `%s` for strings

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

And `%c` for `chars`

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

It knows where the string starts (at `str`), but not where it ends

Types

Strings

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

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

It knows where the string starts (at `str`), but not where it ends

And other contexts? E.g., `n = strlen(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 `printf` tell how long is the string in `printf("str is '%s'\n", str);`?

It knows where the string starts (at `str`), but not where it ends

And other contexts? E.g., `n = strlen(str);`

All it has is an array of characters of some unknown 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 `printf` tell how long is the string in `printf("str is '%s'\n", str);`?

It knows where the string starts (at `str`), but not where it ends

And other contexts? E.g., `n = strlen(str);`

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

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

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

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

It knows where the string starts (at `str`), but not where it ends

And other contexts? E.g., `n = strlen(str);`

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

Stored as a contiguous sequence of bytes in memory: we need some way to indicate 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

`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

The double quote syntax includes the terminating 0; standard string functions expect the terminating 0

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

Exercise. `sizeof` gives the size in bytes of a C value. Compare

```
sizeof("cat")
```

against

```
strlen("cat")
```

Types

Strings

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

Types
Strings

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

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, as part of the language

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

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

Types

More C types?

Types

Types

More C types?

In a very real sense, there are no more types natively supported in C

Types

More C types?

In a very real sense, there are no more types natively supported in C

Again, C is close to the hardware

Types

Types

More C types?

In a very real sense, there are no more types natively supported in C

Again, C is close to the hardware

However, there are a couple of ways of combining types into compound types for the convenience of programming

Types

More C types?

In a very real sense, there are no more types natively supported in C

Again, C is close to the hardware

However, there are a couple of ways of combining types into compound types for the convenience of programming

And for the convenience of the thought processes of the programmers

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;
```

- 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 in C

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 in C
- The type declaration can only contain names of values, as **there are no methods in C**

Types

Structures

Structure types are just like the built-in types

Types

Structures

Structure types are just like the built-in types

So we can have arrays of structs:

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

Types

Structures

Structure types are just like the built-in types

So we can have arrays of structs:

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

So `numbers[7].num`

Types

Structures

Structure types are just like the built-in 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 built-in 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;
```

Types

Structures

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;
```

Compound Types

In summary: the two main ways in C of collecting things together to make compound things are

- arrays: collections of the same type of things
- structures: collections of different types of things

Compound Types

In summary: the two main ways in C of collecting things together to make compound things are

- arrays: collections of the same type of things
- structures: collections of different types of things

Exercise. Read up on `union` types, another way of making compound types in C

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

Compound Types

Exercise for geeks. What is the difference (at the machine level) between

```
int a[2];
```

and

```
struct {
  int a1, a2;
} a;
```

?

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

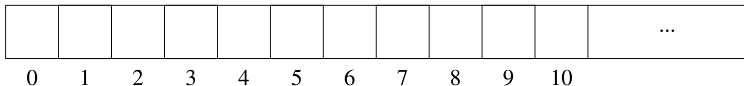
Recall that (thanks to the universal adoption of von Neumann's model) memory can be regarded as a big array of bytes; conventionally numbered from 0 upwards

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



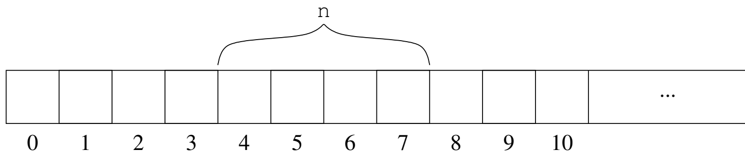
Pointers

When a program is compiled, variables are mapped in some useful way to memory location by the system (compiler and OS program loader)

Pointers

When a program is compiled, variables are mapped in some useful way to memory location by the system (compiler and OS program loader)

So if we have a (4 byte) integer `n` in our code, the system 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: that is, we can find out where a variable has been placed

Pointers

C gives us access to these addresses in our program: that is, we can find out where a variable has been placed

Other languages might not reveal this kind of information, preferring to hide these details from the programmer

Pointers

C gives us access to these addresses in our program: that is, we can find out where a variable has been placed

Other languages might not reveal this kind of information, 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: that is, we can find out where a variable has been placed

Other languages might not reveal this kind of information, 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 (this example: 140732877607788 in decimal) will likely be different on different OSs, 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 (this example: 140732877607788 in decimal) will likely be different on different OSs, 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 when the program is loaded to be run

Pointers

But addresses are just integers

Pointers

Addresses are just integers

Addresses are just integers

C does treat them slightly differently from normal integers to make certain nice things happen, but, at base, they are just integers

Addresses are just integers

C does treat them slightly differently from normal integers to make certain nice things happen, but, at base, they are just integers

The `%p` in `printf` prints addresses in hexadecimal, as that is often useful to the programmer

Addresses are just integers

C does treat them slightly differently from normal integers to make certain nice things happen, but, at base, they 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

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

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

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

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*

(Though it's not the variables that are pointers, but their values. . .)

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 treated as 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 treated as 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 manipulations of them much more convenient

Pointers

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

This is to make manipulations of them much more convenient

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

Pointers

If I gave you `1000000010010010000111111011011` and asked "what does that mean?" you could legitimately say "anything you like"

Pointers

If I gave you `10000000100100100001111111011011` and asked "what does that mean?" you could legitimately say "anything you like"

It is purely the job of the program (and programmer) to say what a particular bunch of bits is supposed to mean

Pointers

If I gave you `1000000010010010000111111011011` and asked "what does that mean?" you could legitimately say "anything you like"

It is purely the job of the program (and programmer) to say what a particular bunch of bits is supposed to mean

The type of a variable or the type of a pointer encodes the information as to what bits they refer to "mean"

Pointers

If I gave you `10000000100100100001111111011011` and asked “what does that mean?” you could legitimately say “anything you like”

It is purely the job of the program (and programmer) to say what a particular bunch of bits is supposed to mean

The type of a variable or the type of a pointer encodes the information as to what bits they refer to “mean”

Thus `int n = 99;` says “allocate four bytes of memory somewhere and (while we access these bytes through this `n`) interpret the bits in those bytes as an integer”

Pointers

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

Pointers

At one point the program might store an integer at a given address; 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

At one point the program might store an integer at a given address; 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

Don't make the mistake of assuming the computer magically "knows" what a bunch of bits means. That's the job of the program

Pointers

At one point the program might store an integer at a given address; 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

Don’t make the mistake of assuming the computer magically “knows” what a bunch of bits means. That’s the job of the program

Note: while C makes this quite plain, the same is true for all computer languages

Pointers

We can declare pointer variables, e.g., `pn`

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

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

Pointers

We can declare pointer variables, e.g., `pn`

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

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

We also say "`pn` is an `int` pointer"; sometimes "`pn` is an integer reference" or even "`pn` is a reference to an `int`"

Pointers

We can declare pointer variables, e.g., `pn`

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

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

We also say "`pn` is an `int` pointer"; sometimes "`pn` is an integer reference" or even "`pn` is a reference to an `int`"

"pointer to" and "reference to" are the same as "address of"

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 ways of writing are treated as exactly the same by the compiler

Pointers

Convention

Note: the convention is to write

```
int *pn;
```

rather than

```
int* pn;
```

Both ways of writing are treated as exactly the same by the compiler

The reason for this slightly awkward convention is that the declaration

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

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

Pointers

Convention

Note: the convention is to write

```
int *pn;
```

rather than

```
int* pn;
```

Both ways of writing are treated as exactly the same by the compiler

The reason for this slightly awkward convention is that the declaration

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

means 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;
```