Pointers

In the opposite direction to `&`, given a pointer we can get at the value at that address using the `*` operator

```
int n = 1234, *pn = &n; // declaration with initialisation
printf("n has value %d, pn has value %p\n", n, pn);
printf("the value at pn is %d\n", *pn);
*pn = 23;
printf("n has value %d, pn has value %p\n", n, pn);
```

Pointers

In the opposite direction to `&`, given a pointer we can get at the value at that address using the `*` operator

```
int n = 1234, *pn = &n; // declaration with initialisation
printf("n has value %d, pn has value %p\n", n, pn);
printf("the value at pn is %d\n", *pn);
*pn = 23;
printf("n has value %d, pn has value %p\n", n, pn);
```

Produces

```
n has value 1234, pn has value 0x7fff81aa38d4
the value at pn is 1234
n has value 23, pn has value 0x7fff81aa38d4
```

Pointers

The * operator says "the value in this variable is a pointer; operate on the value at that address"

Pointers

The $*$ operator says “the value in this variable is a pointer; operate on the value at that address”

Following a pointer to find the value at the address it indicates is called *indirecting* through that pointer

Pointers

The $*$ operator says "the value in this variable is a pointer; operate on the value at that address"

Following a pointer to find the value at the address it indicates is called *indirecting* through that pointer

So if the variable has type pointer to integer, it will fetch 4 bytes of integer; if the variable has type pointer to double, it will fetch 8 bytes of double; and so on

Pointers

The $*$ operator says “the value in this variable is a pointer; operate on the value at that address”

Following a pointer to find the value at the address it indicates is called *indirecting* through that pointer

So if the variable has type pointer to integer, it will fetch 4 bytes of integer; if the variable has type pointer to double, it will fetch 8 bytes of double; and so on

This is one reason why pointers to different types are distinguished: to determine how many bytes of value to access

Pointers

It is important to realise that `*pn = 99` does not modify the value of variable `pn`, but the value at the address contained by `pn`

Pointers

It is important to realise that `*pn = 99` does not modify the value of variable `pn`, but the value at the address contained by `pn`

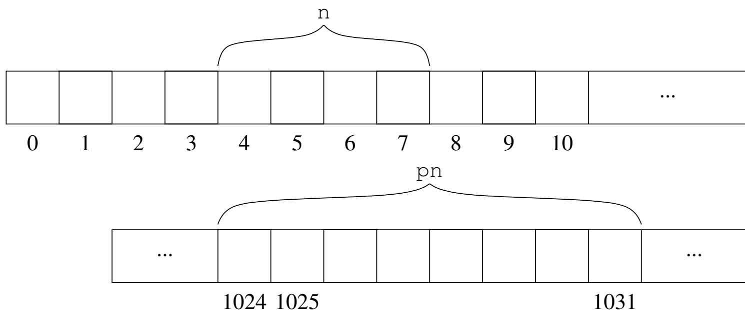
So `*pn = 99` means “get the value of `pn`; store 99 at that address”

Pointers

As `pn` is a perfectly normal variable, it will be associated with some memory location

Pointers

As `pn` is a perfectly normal variable, it will be associated with some memory location



8 bytes of pointer on this 64-bit machine

Pointers

Note: different machines and operating systems may have different size of pointers, according to the needs of the particular machine

Pointers

Note: different machines and operating systems may have different size of pointers, according to the needs of the particular machine

General-purpose 32-bit machines will likely have pointers of size 4 bytes

Pointers

Note: different machines and operating systems may have different size of pointers, according to the needs of the particular machine

General-purpose 32-bit machines will likely have pointers of size 4 bytes

A major gotcha when porting poorly-written programs from 32 to 64 bit architectures

Pointers

In our example the value stored in the 8 bytes starting at byte number 1024 will be the integer “4”, the address of `n`

Pointers

In our example the value stored in the 8 bytes starting at byte number 1024 will be the integer "4", the address of `n`

But we need not stop here: `pn` is a variable, so it has an address: `&pn` is 1024 in this picture

Pointers

In our example the value stored in the 8 bytes starting at byte number 1024 will be the integer “4”, the address of `n`

But we need not stop here: `pn` is a variable, so it has an address: `&pn` is 1024 in this picture

Then `&pn` is a pointer to a pointer to an integer

Pointers

In our example the value stored in the 8 bytes starting at byte number 1024 will be the integer "4", the address of `n`

But we need not stop here: `pn` is a variable, so it has an address: `&pn` is 1024 in this picture

Then `&pn` is a pointer to a pointer to an integer

```
int **ppn = &pn;
```

declares `ppn` to be of type pointer to pointer to integer, and initialises it with a value, namely the address of the variable `pn`

Pointers

In our example the value stored in the 8 bytes starting at byte number 1024 will be the integer “4”, the address of `n`

But we need not stop here: `pn` is a variable, so it has an address: `&pn` is 1024 in this picture

Then `&pn` is a pointer to a pointer to an integer

```
int **ppn = &pn;
```

declares `ppn` to be of type pointer to pointer to integer, and initialises it with a value, namely the address of the variable `pn`

Use multiple `*`s as appropriate to the number of “pointer to”s

Pointers

An assignment

```
**ppn = 100;
```

Pointers

An assignment

```
**ppn = 100;
```

This will update the value of `n`: `*ppn` gets the address of `pn`, then `**ppn` follows that pointer to the address of `n`, the value 100 is stored there

Pointers

An assignment

```
**ppn = 100;
```

This will update the value of `n`: `*ppn` gets the address of `pn`, then `**ppn` follows that pointer to the address of `n`, the value 100 is stored there

Updating `*ppn` will change the address stored in `pn`, e.g., `*ppn = 8;` is the same as `pn = 8;`

Pointers

An assignment
`**ppn = 100;`

This will update the value of `n`: `*ppn` gets the address of `pn`, then `**ppn` follows that pointer to the address of `n`, the value 100 is stored there

Updating `*ppn` will change the address stored in `pn`, e.g., `*ppn = 8;` is the same as `pn = 8;`

Now `pn` no longer points at `n` but another place in memory

Pointers

Note: these operations never change the locations of variables, merely their values

Pointers

Note: these operations never change the locations of variables, merely their values

The assignment `pn = 8;` does not move `n` at all

Pointers

Note: these operations never change the locations of variables, merely their values

The assignment `pn = 8;` does not move `n` at all

It just means the pointer `pn` now points at some other part of memory

Pointers

Note: these operations never change the locations of variables, merely their values

The assignment `pn = 8;` does not move `n` at all

It just means the pointer `pn` now points at some other part of memory

Pointers can point anywhere in memory, they are not restricted to point at locations of variables

Pointers

Note: these operations never change the locations of variables, merely their values

The assignment `pn = 8;` does not move `n` at all

It just means the pointer `pn` now points at some other part of memory

Pointers can point anywhere in memory, they are not restricted to point at locations of variables

In fact, most useful applications of pointers are *not* pointing at variables

Pointers

Exercise (harder). The following does not work. Explain why and fix it using pointers.

```
void swap(int a, int b)
{
  int tmp = a;
  a = b;
  b = tmp;
}
...
int n = 1, m = 2;
swap(n, m);
printf("n = %d m = %d\n", n, m);
```

Pointers

One of Java's design principles was to eliminate pointers. In reality, it *does* have pointers, it just hides the fact from the naive programmer. And makes it harder for experienced programmers

Pointers

One of Java's design principles was to eliminate pointers. In reality, it *does* have pointers, it just hides the fact from the naive programmer. And makes it harder for experienced programmers

Exercise. Is it possible to write a (primitive) integer swap function in Java?

Exercise. Find out how Java manages pointers

Arrays and Pointers

Pointers are intimately associated with arrays in C

Arrays and Pointers

Pointers are intimately associated with arrays in C

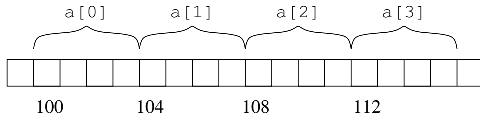
Consider an array `int a[4];`

Arrays and Pointers

Pointers are intimately associated with arrays in C

Consider an array `int a[4];`

In memory, C arrays are laid out simply



To be definite, we fix on using 4 byte (32 bit) integers

Arrays and Pointers

Adjacent members of the array are adjacent in memory

Arrays and Pointers

Adjacent members of the array are adjacent in memory

If the array starts at address 100, so `a[0]` is at address 100,
then `a[1]` is at address $100 +$ `sizeof(int)`;
`a[2]` is at address $100 + 2 \times$ `sizeof(int)`;
and so on

Arrays and Pointers

Adjacent members of the array are adjacent in memory

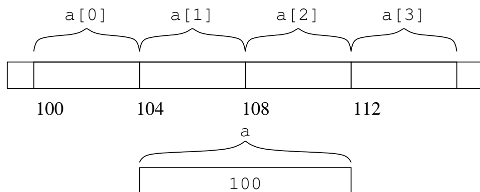
If the array starts at address 100, so `a[0]` is at address 100, then `a[1]` is at address $100 + \texttt{sizeof(int)}$;
`a[2]` is at address $100 + 2 \times \texttt{sizeof(int)}$;
and so on

Array element n is at address

$$100 + n \times \texttt{sizeof(int)}$$

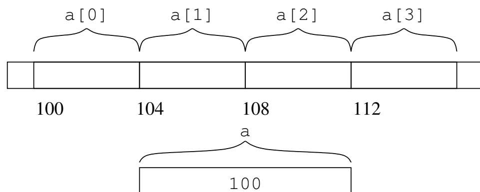
Arrays and Pointers

In fact the variable `a` contains the address of the start of the array



Arrays and Pointers

In fact the variable `a` contains the address of the start of the array



So `a` actually has type `int*` (with a caveat)

Arrays and Pointers

The array index operator `[]` is simply a bit of pointer arithmetic

Arrays and Pointers

The array index operator `[]` is simply a bit of pointer arithmetic

`a[3]` is the same as `*(a + 3)`

Arrays and Pointers

The array index operator `[]` is simply a bit of pointer arithmetic

`a[3]` is the same as `*(a + 3)`

An important point here: arithmetic on pointers is different from arithmetic on integers

Arrays and Pointers

The array index operator `[]` is simply a bit of pointer arithmetic

`a[3]` is the same as `*(a + 3)`

An important point here: arithmetic on pointers is different from arithmetic on integers

For an integer pointer `a` the expression `a + 3` is not simply the address 3 along from the value in `a`, but instead is the address of the *3rd integer* along

Arrays and Pointers

The array index operator `[]` is simply a bit of pointer arithmetic

`a[3]` is the same as `*(a + 3)`

An important point here: arithmetic on pointers is different from arithmetic on integers

For an integer pointer `a` the expression `a + 3` is not simply the address 3 along from the value in `a`, but instead is the address of the *3rd integer* along

If the value in `a` is 100, and integers are of size 4, then `a + 3` is the address $100 + 3 \times 4 = 112$

Arrays and Pointers

This is strange at first, but turns out to be what you always want

Arrays and Pointers

This is strange at first, but turns out to be what you always want

This is another reason to distinguish types of pointers. For a variable `v` of type `T*`, the expression `v + ` n is computed as

$$\texttt{v} + n \times \texttt{sizeof(T)}$$

Giving the address of the nth item along

Arrays and Pointers

This is strange at first, but turns out to be what you always want

This is another reason to distinguish types of pointers. For a variable `v` of type `T*`, the expression `v + ` n is computed as

$$\texttt{v} + n \times \texttt{sizeof(T)}$$

Giving the address of the *n*th item along

Pointer arithmetic counts in items, not bytes

Arrays and Pointers

The result is that for an array `a[]`

- the value of `a` is the address of the start of the array
- `a + 3` is the address of the item 3 further along (the 4th item)
- so `*(a + 3)` is the value there; just like `a[3]`
- `&(a[3])` is the address of that item; as is `a + 3`

Arrays and Pointers

The result is that for an array `a[]`

- the value of `a` is the address of the start of the array
- `a + 3` is the address of the item 3 further along (the 4th item)
- so `*(a + 3)` is the value there; just like `a[3]`
- `&(a[3])` is the address of that item; as is `a + 3`

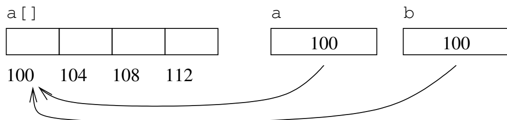
And exactly the same is true for a pointer variable, though this may or may not be pointing at the memory for an array

Arrays and Pointers

Also ++ works well. If we define an integer pointer

```
int *b;
b = a;
```

then `*b` (equivalently, `b[0]`) is the same as `a[0]`



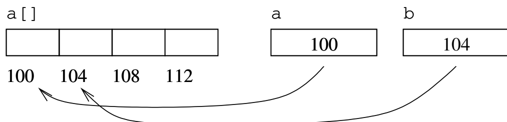
Arrays and Pointers

Also ++ works well. If we define an integer pointer

```
int *b;
b = a;
```

then `*b` (equivalently, `b[0]`) is the same as `a[0]`

After we increment `b++;` which is the same as `b = b+1;` the pointer moves one integer along. Now `*b` is the same as `a[1]`



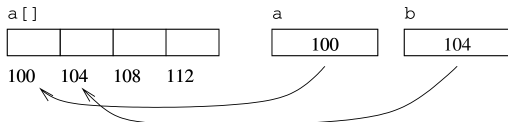
Arrays and Pointers

Also ++ works well. If we define an integer pointer

```
int *b;
b = a;
```

then `*b` (equivalently, `b[0]`) is the same as `a[0]`

After we increment `b++;` which is the same as `b = b+1;` the pointer moves one integer along. Now `*b` is the same as `a[1]`



So we can iterate `b` along the array

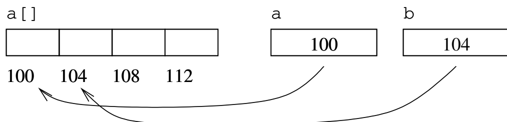
Arrays and Pointers

Also ++ works well. If we define an integer pointer

```
int *b;
b = a;
```

then `*b` (equivalently, `b[0]`) is the same as `a[0]`

After we increment `b++;` which is the same as `b = b+1;` the pointer moves one integer along. Now `*b` is the same as `a[1]`



So we can iterate `b` along the array

The increment operator is an array iteration operator (as well as the normal increase-by-one on usual integers)

Arrays and Pointers

```
for (b = a; ...  ; b++) { ...  }
```

is a common sight in C programs

Arrays and Pointers

```
for (b = a; ...  ; b++) { ...  }
```

is a common sight in C programs

And for a pointer `b` we can write `b[i]` just as if `b` was the name of an array

Arrays and Pointers

```
for (b = a; ...  ; b++) { ...  }
```

is a common sight in C programs

And for a pointer `b` we can write `b[i]` just as if `b` was the name of an array

The `x[n]` syntax really is just a short way of writing `*(x + n)` so is equally useful for both arrays and pointers

Arrays and Pointers

In a very real sense, C does not have arrays as a basic type!

Arrays and Pointers

In a very real sense, C does not have arrays as a basic type!

Just integers, floating point, and pointers

Arrays and Pointers

In a very real sense, C does not have arrays as a basic type!

Just integers, floating point, and pointers

Not necessarily a useful point of view, though!

Arrays and Pointers

Exercise. Explain

```
void copy(int *dst, int *src, int len)
{
  while (len--) {
    *dst++ = *src++;
  }
}
...
int a[128], b[128];
...
copy(b, a, 128);
```

Arrays and Pointers

Exercise. Then explain

```
copy(a, a + 64, 64);
```

Arrays and Pointers

We can now see the close identification of pointers and arrays

Arrays and Pointers

We can now see the close identification of pointers and arrays

In the declaration `int a[4];` we can regard the variable `a` to have type "array of integer", or as type "pointer to integer"

Arrays and Pointers

We can now see the close identification of pointers and arrays

In the declaration `int a[4];` we can regard the variable `a` to have type “array of integer”, or as type “pointer to integer”

In all but a few useful circumstances the two concepts are interchangeable

Arrays and Pointers

We can now see the close identification of pointers and arrays

In the declaration `int a[4];` we can regard the variable `a` to have type “array of integer”, or as type “pointer to integer”

In all but a few useful circumstances the two concepts are interchangeable

We can use pointers as arrays and arrays as pointers

Arrays and Pointers

We can now see the close identification of pointers and arrays

In the declaration `int a[4];` we can regard the variable `a` to have type “array of integer”, or as type “pointer to integer”

In all but a few useful circumstances the two concepts are interchangeable

We can use pointers as arrays and arrays as pointers

As long as you understand what you are doing

Except for a few subtleties

Array types and pointer types are interchangeable

Except for a few subtleties

Array types and pointer types are interchangeable

Array types are a special subset of pointer types

Arrays and Pointers

This identification makes it easy to see what happens when we mis-index an array

Arrays and Pointers

This identification makes it easy to see what happens when we mis-index an array

What is `a[10]` in the above example?

Arrays and Pointers

This identification makes it easy to see what happens when we mis-index an array

What is `a[10]` in the above example?

It is the value stored at address $100 + 10 \times 4 = 140$, regarded as an integer

Arrays and Pointers

This might

Arrays and Pointers

This might

- successfully return some integer value from whatever happens to be at that memory location

Arrays and Pointers

This might

- successfully return some integer value from whatever happens to be at that memory location
- this might be unused and initialised memory, or it might be where some other value (not necessarily an integer) is currently placed

Arrays and Pointers

This might

- successfully return some integer value from whatever happens to be at that memory location
- this might be unused and initialised memory, or it might be where some other value (not necessarily an integer) is currently placed
- or it might refer to an unmapped memory location (think about virtual memory and unmapped pages), when the OS will cause an interrupt and likely terminate your program

Arrays and Pointers

If your program crashes with a *segmentation violation* error, this is likely what is happening: your program is reading or writing to a spurious area of memory

Arrays and Pointers

If your program crashes with a *segmentation violation* error, this is likely what is happening: your program is reading or writing to a spurious area of memory

Occasionally you will see *bus error* for the same thing

Arrays and Pointers

If your program crashes with a *segmentation violation* error, this is likely what is happening: your program is reading or writing to a spurious area of memory

Occasionally you will see *bus error* for the same thing

If this happens you need to look carefully at your program to find the error

Arrays and Pointers

Under Linux there is a useful checking program called `valgrind` that is good at finding memory mis-access errors

Arrays and Pointers

Under Linux there is a useful checking program called `valgrind` that is good at finding memory mis-access errors

Compile your program using the `-g` option to include debugging information: `cc -Wall -g ...`

Arrays and Pointers

Under Linux there is a useful checking program called `valgrind` that is good at finding memory mis-access errors

Compile your program using the `-g` option to include debugging information: `cc -Wall -g ...`

`valgrind ./myprog`
will run the program `./myprog` checking *all* the program's accesses to memory

Arrays and Pointers

Under Linux there is a useful checking program called `valgrind` that is good at finding memory mis-access errors

Compile your program using the `-g` option to include debugging information: `cc -Wall -g ...`

```
valgrind ./myprog
```

will run the program `./myprog` checking *all* the program's accesses to memory

This will slow execution horribly, of course, but it will highlight when you do something stupid

Arrays and Pointers

As mentioned, usually C does not check for these kids of errors as

Arrays and Pointers

As mentioned, usually C does not check for these kids of errors as

- the high cost of checking memory accesses

Arrays and Pointers

As mentioned, usually C does not check for these kids of errors as

- the high cost of checking memory accesses
- sometimes the programmer *does* want to write code that accesses off the nominal ends of an array; you can sometimes find code like `a[-1]`. This is valid C, and the programmer will get everything they deserve

Arrays and Pointers

In the declaration of an array and a pointer

```
int a[4]
int *b;
```

we need to be careful about what is happening

Arrays and Pointers

In the declaration of an array and a pointer

```
int a[4]
int *b;
```

we need to be careful about what is happening

`a` is a variable of type pointer to integer **and** a chunk of memory (e.g., 16 bytes) is reserved somewhere for the array; the value of the variable `a` will be the address of that chunk of memory

Arrays and Pointers

In the declaration of an array and a pointer

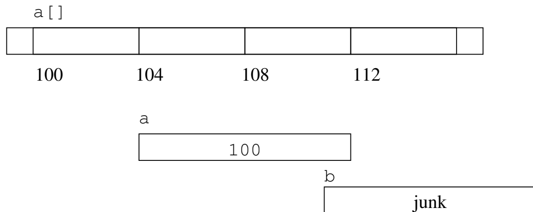
```
int a[4]
int *b;
```

we need to be careful about what is happening

`a` is a variable of type pointer to integer **and** a chunk of memory (e.g., 16 bytes) is reserved somewhere for the array; the value of the variable `a` will be the address of that chunk of memory

`b` is a variable of type pointer to integer, with no particular value, and no chunk of memory is reserved

Arrays and Pointers



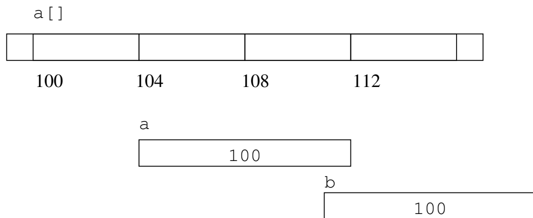
`int a[4]` sets up both `a` and the space for the array; `int *b` just sets up `b`

Arrays and Pointers

`b` is a pointer variable, so we can set its value: `b = a;`

Arrays and Pointers

`b` is a pointer variable, so we can set its value: `b = a;`



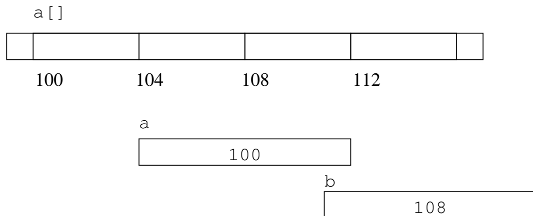
And now `b[1]` makes sense; it is the same as `a[1]`
`b[1]` is at address $100 + 1 \times 4 = 104$

Arrays and Pointers

We could equally do `b = a + 2`

Arrays and Pointers

We could equally do `b = a + 2`

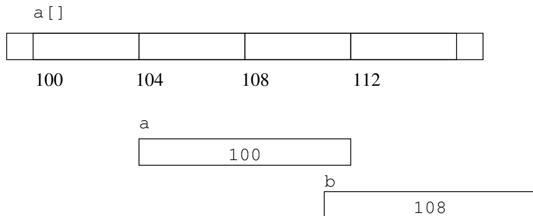


And now `b[1]` makes sense; it is the same as `a[3]`

`b[1]` is at address $108 + 1 \times 4 = 112$

Arrays and Pointers

We could equally do `b = a + 2`



And now `b[1]` makes sense; it is the same as `a[3]`
`b[1]` is at address $108 + 1 \times 4 = 112$

And now `b[-1]` makes sense; it is the same as `a[1]`
`b[-1]` is at address $108 + (-1) \times 4 = 104$

Arrays and Pointers

Another difference in this declaration is that `a` is a *constant* variable (!)

Arrays and Pointers

Another difference in this declaration is that `a` is a *constant* variable (!)

We can't change the value of `a`

Arrays and Pointers

Another difference in this declaration is that `a` is a *constant* variable (!)

We can't change the value of `a`

This is what we usually want from arrays; while `b` is explicitly a variable pointer

Arrays and Pointers

```
void foo(void)
{
  int a[4];
  a++;
}
```

gives an error message in the compiler

```
const.c: In function 'foo':
const.c:5:3: error: lvalue required as increment operand
```

An “lvalue” is a thing that can appear on the left side of an assignment, e.g., an updatable variable

Arrays and Pointers

```
void foo(void)
{
  int a[4], *b = a;
  b++;
}
```

is OK as `b` is allowed to vary

Arrays and Pointers

This may seem trivial but the following is very popular, particularly from Java-trained “programmers”

```
void foo(void)
{
  int a[4], *b;
  ...
  b = ...
  ...
  a = b;
  ...
}
```

Bad!

Strings and Pointers

Strings are just arrays of `char`; string variables thus have type pointer to `char`, i.e., `char *`

We can have

```
char a[4] = "xyz", *b;
```

just as before; `a` has type `char *`

Strings and Pointers

Strings are just arrays of `char`; string variables thus have type pointer to `char`, i.e., `char *`

We can have

```
char a[4] = "xyz", *b;
```

just as before; `a` has type `char *`

Now the value of `a` is a (constant) pointer to an array of 4 characters; the value of `b` is nothing in particular

Strings and Pointers

What happens with `b = a`?

Strings and Pointers

What happens with `b = a`?

Just as before, the variable `b` now points to the same memory as `a`

Strings and Pointers

What happens with `b = a`?

Just as before, the variable `b` now points to the same memory as `a`

Note there is no copying of characters involved

Strings and Pointers

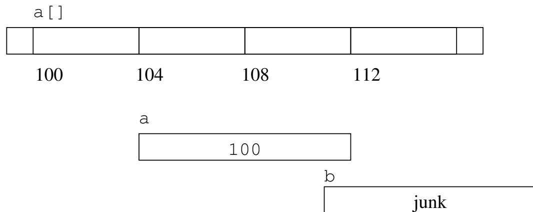
What happens with `b = a`?

Just as before, the variable `b` now points to the same memory as `a`

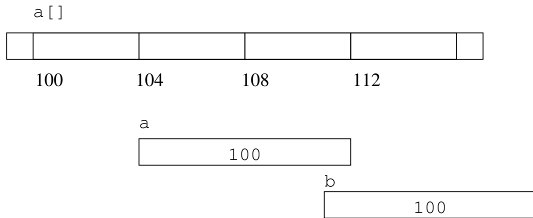
Note there is no copying of characters involved

Just the value in `a` (an address) is copied into `b`, nothing more

Strings and Pointers



Strings and Pointers




```
b = a;
```

Strings and Pointers

Maybe if we had

```
char a[4] = "xyz", b[4];
```

then `b = a;` would copy from characters `a` to `b`?

Strings and Pointers

Maybe if we had

```
char a[4] = "xyz", b[4];
```

then `b = a;` would copy from characters `a` to `b`?

No. `b` is now a constant variable, so this would not even compile as you are still trying to update the value of `b`

Strings and Pointers

Maybe if we had

```
char a[4] = "xyz", b[4];
```

then `b = a;` would copy from characters `a` to `b`?

No. `b` is now a constant variable, so this would not even compile as you are still trying to update the value of `b`

The point is that `a` and `b` are simply pointers to where the characters live: they are not the characters

Strings and Pointers

Maybe if we had

```
char a[4] = "xyz", b[4];
```

then `b = a;` would copy from characters `a` to `b`?

No. `b` is now a constant variable, so this would not even compile as you are still trying to update the value of `b`

The point is that `a` and `b` are simply pointers to where the characters live: they are not the characters

The characters are `a[0]`, `a[1]`, etc.

Strings and Pointers

Maybe if we had

```
char a[4] = "xyz", b[4];
```

then `b = a;` would copy from characters `a` to `b`?

No. `b` is now a constant variable, so this would not even compile as you are still trying to update the value of `b`

The point is that `a` and `b` are simply pointers to where the characters live: they are not the characters

The characters are `a[0]`, `a[1]`, etc.

To copy the characters we can go

```
b[0] = a[0]; b[1] = a[1]; ...
```

more likely using a `for` loop

Strings and Pointers

So to copy the contents of one string to another we can either (a) use a loop, or (b) use the library function `strcpy`

Strings and Pointers

So to copy the contents of one string to another we can either (a) use a loop, or (b) use the library function `strcpy`

This function has type

```
char *strcpy(char *dest, const char *src);
```

copying `src` to `dest`

Strings and Pointers

So to copy the contents of one string to another we can either (a) use a loop, or (b) use the library function `strcpy`

This function has type

```
char *strcpy(char *dest, const char *src);
```

copying `src` to `dest`

So

```
int a[4] = "xyz", b[4]; strcpy(b, a);
```

will copy the contents of the (zero-terminated) string pointed to by `a` to the area of memory pointed to by `b`

Strings and Pointers

So to copy the contents of one string to another we can either (a) use a loop, or (b) use the library function `strcpy`

This function has type

```
char *strcpy(char *dest, const char *src);
```

copying `src` to `dest`

So

```
int a[4] = "xyz", b[4]; strcpy(b, a);
```

will copy the contents of the (zero-terminated) string pointed to by `a` to the area of memory pointed to by `b`

And `strcpy` will copy characters up to and including the terminating 0 byte of the string at `a`

Strings and Pointers

Notes

Strings and Pointers

Notes

- The variable `b` here is constant, not the memory it refers to

Strings and Pointers

Notes

- The variable `b` here is constant, not the memory it refers to
- It is the responsibility of the programmer to ensure the area of memory referred to by `a` does have a terminating 0 in the appropriate place

Strings and Pointers

Notes

- The variable `b` here is constant, not the memory it refers to
- It is the responsibility of the programmer to ensure the area of memory referred to by `a` does have a terminating 0 in the appropriate place
- It is the responsibility of the programmer to ensure the area of memory referred to by `b` is large enough to contain a copy of `a`

Strings and Pointers

Notes

- The variable `b` here is constant, not the memory it refers to
- It is the responsibility of the programmer to ensure the area of memory referred to by `a` does have a terminating 0 in the appropriate place
- It is the responsibility of the programmer to ensure the area of memory referred to by `b` is large enough to contain a copy of `a`

Forgetting these is a popular source of bugs

Strings and Pointers

```
char a[] = "hello world", b[4];
strcpy(b, a);
```

will likely not do what you want

Strings and Pointers

```
char a[] = "hello world", b[4];
strcpy(b, a);
```

will likely not do what you want

Exercise. What is the output from the following?

```
char a[] = "the cat sat on the mat", *b;
b = a;
b[4] = 'r';
printf("a is '%s'\nb is '%s'\n", a, b);
```

Exercise. What is the bug here?

```
char a[] = "the cat sat on the mat", *b;
strcpy(b, a);
```

Strings and Pointers

Exercise. What about

```
char a[] = "hello", b[5];
strcpy(b, a);
```

Strings and Pointers

Exercise. What about

```
char a[] = "hello", b[5];
strcpy(b, a);
```

Look up the function `strlen`. Reimplement it yourself

Strings and Pointers

A good example of the use of strings as pointers is in the way C treats program arguments

Strings and Pointers

A good example of the use of strings as pointers is in the way C treats program arguments

When we run a program we often want to pass some values to that program: `./summit 23 42`

Strings and Pointers

A good example of the use of strings as pointers is in the way C treats program arguments

When we run a program we often want to pass some values to that program: `./summit 23 42`

The arguments passed to the program are presented to the `main` function

Strings and Pointers

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
  int n, m;

  if (argc < 3) {
    printf("Not enough arguments!\n");
    return 1;
  }

  n = atoi(argv[1]);
  m = atoi(argv[2]);
  printf("sum is %d\n", n + m);

  return 0;
}
```

Strings and Pointers

Lots of things here

Strings and Pointers

Lots of things here

- There is another `#include`. We shall discuss these later, but these declare types of functions, `atoi` in this case

Strings and Pointers

Lots of things here

- There is another `#include`. We shall discuss these later, but these declare types of functions, `atoi` in this case
- `main` can take two arguments, conventionally called `argc` and `argv`

Strings and Pointers

Lots of things here

- There is another `#include`. We shall discuss these later, but these declare types of functions, `atoi` in this case
- `main` can take two arguments, conventionally called `argc` and `argv`
- `argc` is the number of arguments, *including the program name*. The program name (`"summit"`) will be argument 0; `"23"` will be argument 1; `"42"` will be argument 2

Strings and Pointers

Lots of things here

- There is another `#include`. We shall discuss these later, but these declare types of functions, `atoi` in this case
- `main` can take two arguments, conventionally called `argc` and `argv`
- `argc` is the number of arguments, *including the program name*. The program name (`"summit"`) will be argument 0; `"23"` will be argument 1; `"42"` will be argument 2
- `argv` has type pointer to pointer to `char`; namely an array of pointer to char; namely an array of strings

Strings and Pointers

Lots of things here

- There is another `#include`. We shall discuss these later, but these declare types of functions, `atoi` in this case
- `main` can take two arguments, conventionally called `argc` and `argv`
- `argc` is the number of arguments, *including the program name*. The program name (`"summit"`) will be argument 0; `"23"` will be argument 1; `"42"` will be argument 2
- `argv` has type pointer to pointer to `char`; namely an array of pointer to char; namely an array of strings
- The length of this array is `argc`, of course

Strings and Pointers

Lots of things here

- There is another `#include`. We shall discuss these later, but these declare types of functions, `atoi` in this case
- `main` can take two arguments, conventionally called `argc` and `argv`
- `argc` is the number of arguments, *including the program name*. The program name (`"summit"`) will be argument 0; `"23"` will be argument 1; `"42"` will be argument 2
- `argv` has type pointer to pointer to `char`; namely an array of pointer to char; namely an array of strings
- The length of this array is `argc`, of course
- Some people declare `argv` as `char *argv[]` to emphasise it is an array of strings

Strings and Pointers

- The arguments are always passed in as strings; we will have to convert a string `"23"` to an integer 23

Strings and Pointers

- The arguments are always passed in as strings; we will have to convert a string `"23"` to an integer `23`
- `if (argc < 3) ...` remember the program name is included in the count

Strings and Pointers

- The arguments are always passed in as strings; we will have to convert a string `"23"` to an integer `23`
- `if (argc < 3) ...` remember the program name is included in the count
- The function `atoi` converts a string containing an integer to an integer. See `man atoi`

Strings and Pointers

C has a huge library of useful functions

Strings and Pointers

C has a huge library of useful functions

Quite often something you thought you might have to write yourself is already in a library

Strings and Pointers

C has a huge library of useful functions

Quite often something you thought you might have to write yourself is already in a library

You will have to just explore!

Arrays and Pointers

Exercise. Look up `strncpy` (extra 'n' in there)

Exercise. What about `3[a]`? Or `0[a+3]`?

Exercise. For `int a[4], *b;` compare `sizeof(a)`, `sizeof(*a)`, `sizeof(b)`, `sizeof(*b)`

Exercise. Read the specification for `atoi` and implement it for yourself (give your version a different name!)

Pointers

One more thing about pointers: the `void` pointer

Pointers

One more thing about pointers: the `void` pointer

You will see code with variables declared as `void *`, e.g.,

```
void *memcpy(void *dest, const void *src, size_t n);
```

Pointers

One more thing about pointers: the `void` pointer

You will see code with variables declared as `void *`, e.g.,

```
void *memcpy(void *dest, const void *src, size_t n);
```

Rather than meaning a pointer to nothing, it means a pointer to something, but we don’t know what type of thing

Pointers

One more thing about pointers: the `void` pointer

You will see code with variables declared as `void *`, e.g.,

```
void *memcpy(void *dest, const void *src, size_t n);
```

Rather than meaning a pointer to nothing, it means a pointer to something, but we don't know what type of thing

This allows us to write functions that act on arbitrary pointers: `memcpy` copies arbitrary blocks of objects, be it `int`s, `double`s, or `struct` whatevers

Pointers

```
int a[10], b[10];
double x[5], y[5];
...
memcpy(b, a, 10*sizeof(int));
memcpy(y, x, 5*sizeof(double));
```

copies 10 integers-worth of bytes from where `a` points to where `b` points; and 5 doubles-worth of bytes from `x` points to where `y` points

Pointers

Exercise. What is the error here?

```
int a[5];
void *b;
...
b = a;
b[0] = b[1] + b[2];
```