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: if we are thinking of `a` as indicating the start of an array we don't want its value wandering about in memory

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: if we are thinking of `a` as indicating the start of an array we don't want its value wandering about in memory

And `b` is explicitly a variable pointer: if we need something variable, use a 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

Clang says:

```
const.c:4:4: error: cannot increment value of type 'int [4]'
  a++;
  ~^
1 error generated.
```

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 = ...  // b gets some value
  ...
  a = b;
  ...
}
```

Bad!

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 = ...  // b gets some value
  ...
  a = b;
  ...
}
```

Bad!

Clearly bad here, but Java allows lots of other types of composite objects (i.e., its “objects”) where this kind of thing is not so visually obviously 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` can be used as a `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` can be used as a `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

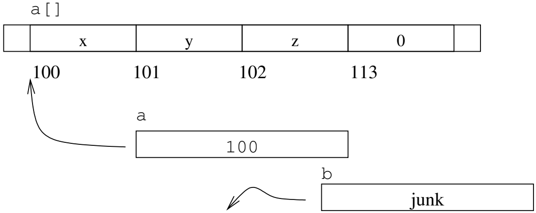
What happens with `b = a`?

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

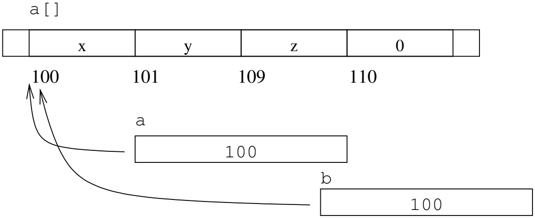
Note there is no copying of characters involved

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 characters from `a` to `b`?

Strings and Pointers

Maybe if we had

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

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

Note `strcpy` will continue copy characters until it hits a 0 in `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 an 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 an 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 an 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

```c
#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` (or `void` arguments if we don't want to pass values into the program)

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` (or `void` arguments if we don't want to pass values into the program)
- `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` (or `void` arguments if we don't want to pass values into the program)
- `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 array of pointer to `char`; namely an array of strings: read this as `char (*argv)[]`, i.e., an array of `char *`

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` (or `void` arguments if we don't want to pass values into the program)
- `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 array of pointer to `char`; namely an array of strings: read this as `char (*argv)[]`, i.e., an array of `char *`
- 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` (or `void` arguments if we don't want to pass values into the program)
- `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 array of pointer to `char`; namely an array of strings: read this as `char (*argv)[]`, i.e., an array of `char *`
- The length of this array is `argc`, of course
- Some people declare `argv` as `char **argv` and play tricks with changing the (now non-constant) variable `argv`

Strings and Pointers

Exercise. Compare the declarations

```
int One(char *one[]) ...
int Two(char *(two[])) ...
int Three(char **three) ...
```

Strings and Pointers

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

Strings and Pointers

- The arguments to `main` are 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 to `main` are 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]`? Or `(a+3)[0]`?

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];
```

Casting

C distinguishes between pointers of different types

Casting

C distinguishes between pointers of different types

Sometimes it is useful to convert a pointer to a different type

Casting

C distinguishes between pointers of different types

Sometimes it is useful to convert a pointer to a different type

C allows us to *cast* many types of objects to other types, not just pointers

Casting

C distinguishes between pointers of different types

Sometimes it is useful to convert a pointer to a different type

C allows us to *cast* many types of objects to other types, not just pointers

There are the automatic coercions mentioned earlier (e.g., integer to floating point), but the programmer can explicitly cast types, too

Casting

C distinguishes between pointers of different types

Sometimes it is useful to convert a pointer to a different type

C allows us to *cast* many types of objects to other types, not just pointers

There are the automatic coercions mentioned earlier (e.g., integer to floating point), but the programmer can explicitly cast types, too

(Coerce: changing type, usually automatic. Cast: changing type, usually programmatic)

Casting

C distinguishes between pointers of different types

Sometimes it is useful to convert a pointer to a different type

C allows us to *cast* many types of objects to other types, not just pointers

There are the automatic coercions mentioned earlier (e.g., integer to floating point), but the programmer can explicitly cast types, too

(Coerce: changing type, usually automatic. Cast: changing type, usually programmatic)

The syntax is
`(typename)expression`
to convert the value of the expression to have type `typename`

Casting

So, for example, `int *a = (int*)b;` where `b` is some pointer

Casting

So, for example, `int *a = (int*)b;` where `b` is some pointer

Now `a` points to the same address as `b`; the value of `a` is the same as the value of `b`

Casting

So, for example, `int *a = (int*)b;` where `b` is some pointer

Now `a` points to the same address as `b`; the value of `a` is the same as the value of `b`

As mentioned previously, it is merely the interpretation of the bits at those addresses that may differ

Casting

Indeed, we can convert between integers and pointers:

```
int *a = (int*)42;
```

makes `a` point at address 42 and regard what happens to be there as an integer

Casting

Indeed, we can convert between integers and pointers:

```
int *a = (int*)42;
```

makes `a` point at address 42 and regard what happens to be there as an integer

And `long int n = (long int)p;` where `p` is some pointer casts from a pointer to an integer

Casting

Indeed, we can convert between integers and pointers:

```
int *a = (int*)42;
```

makes `a` point at address 42 and regard what happens to be there as an integer

And `long int n = (long int)p;` where `p` is some pointer casts from a pointer to an integer

So we can now do arithmetic on the integer `n`

Casting

Indeed, we can convert between integers and pointers:
`int *a = (int*)42;`
makes `a` point at address 42 and regard what happens to be there as an integer

And `long int n = (long int)p;` where `p` is some pointer casts from a pointer to an integer

So we can now do arithmetic on the integer `n`

For people who know what they are doing only!

Casting

Indeed, we can convert between integers and pointers:

```
int *a = (int*)42;
```

makes `a` point at address 42 and regard what happens to be there as an integer

And `long int n = (long int)p;` where `p` is some pointer casts from a pointer to an integer

So we can now do arithmetic on the integer `n`

For people who know what they are doing only!

Exercise. Compare `n + 1` and `p + 1`

Casting

Of course, we can explicitly cast between integers and floating point, over and above the usual automatic coercions

Casting

Of course, we can explicitly cast between integers and floating point, over and above the usual automatic coercions

`int a = (int)3.141;` makes `a` have the value 3

Casting

Of course, we can explicitly cast between integers and floating point, over and above the usual automatic coercions

`int a = (int)3.141;` makes `a` have the value 3

`double b = 1.0 + (double)(1 + 1);` makes `b` have the value 3.0

Casting

Of course, we can explicitly cast between integers and floating point, over and above the usual automatic coercions

`int a = (int)3.141;` makes `a` have the value 3

`double b = 1.0 + (double)(1 + 1);` makes `b` have the value 3.0

The latter cast (`int` to `double`) would happen automatically, but it doesn't hurt to make it explicit so we are very clear on what is happening

Casting

Of course, we can explicitly cast between integers and floating point, over and above the usual automatic coercions

`int a = (int)3.141;` makes `a` have the value 3

`double b = 1.0 + (double)(1 + 1);` makes `b` have the value 3.0

The latter cast (`int` to `double`) would happen automatically, but it doesn't hurt to make it explicit so we are very clear on what is happening

Find out what happens with
`int n = (int)1e100;`

Casting

Recall that widening, the kind of type change that merely extends a bit pattern, is quite different from converting integers to floating point, which completely changes the bits

Casting

Recall that widening, the kind of type change that merely extends a bit pattern, is quite different from converting integers to floating point, which completely changes the bits

When we cast between pointer types, the casting actually does nothing at all to the bits!

Casting

Recall that widening, the kind of type change that merely extends a bit pattern, is quite different from converting integers to floating point, which completely changes the bits

When we cast between pointer types, the casting actually does nothing at all to the bits!

In

```
double *a;
...
int *b = (int*)a;
```

the value of `b` is identical to the value of `a`

Casting

Recall that widening, the kind of type change that merely extends a bit pattern, is quite different from converting integers to floating point, which completely changes the bits

When we cast between pointer types, the casting actually does nothing at all to the bits!

In

```
double *a;
...
int *b = (int*)a;
```

the value of `b` is identical to the value of `a`

It is entirely a message to the compiler to interpret the bits at that address differently

Casting

`a` says look at this address and regard the 8 bytes there as a double

`b` says look at this address and regard the 4 bytes there as an int

Casting

`a` says look at this address and regard the 8 bytes there as a double

`b` says look at this address and regard the 4 bytes there as an int

Not often a sensible thing to do!

Casting

`a` says look at this address and regard the 8 bytes there as a double

`b` says look at this address and regard the 4 bytes there as an int

Not often a sensible thing to do!

But vital for some low-level machine operations!

Casting

`a` says look at this address and regard the 8 bytes there as a double

`b` says look at this address and regard the 4 bytes there as an int

Not often a sensible thing to do!

But vital for some low-level machine operations!

Exercise. Read up on automatic pointer coercions, including `void*`

Structures and Pointers

Arrays are fixed-size structures in C

Structures and Pointers

Arrays are fixed-size structures in C

Once declared, their length cannot be altered

Structures and Pointers

Arrays are fixed-size structures in C

Once declared, their length cannot be altered

Some languages allow variably sized arrays: there is a hidden cost to this, though, in speed of access to the elements of the array

Structures and Pointers

Arrays are fixed-size structures in C

Once declared, their length cannot be altered

Some languages allow variably sized arrays: there is a hidden cost to this, though, in speed of access to the elements of the array

Modern programs need dynamic structures, like lists and trees, that can grow and shrink

Structures and Pointers

Arrays are fixed-size structures in C

Once declared, their length cannot be altered

Some languages allow variably sized arrays: there is a hidden cost to this, though, in speed of access to the elements of the array

Modern programs need dynamic structures, like lists and trees, that can grow and shrink

Lists and other dynamic datastructures are made easy in C by the use of structures and pointers

Structures and Pointers

We can define

```
struct intlist {
  int val;
  struct intlist *next;
};
```

This structure contains an integer value and a pointer to the next item in the list

Structures and Pointers

Exercise. Reflect for a moment why

```
struct intlist {
  int val;
  struct intlist next;
};
```

does not make sense

Structures and Pointers

We can define a few values

```
struct intlist a, b, c;
a.val = 12; a.next = &b;
b.val = 34; b.next = &c;
c.val = 56; c.next = 0;
```

N.B. this is *not* the right way to do this kind of thing

Structures and Pointers

We can define a few values

```
struct intlist a, b, c;
a.val = 12; a.next = &b;
b.val = 34; b.next = &c;
c.val = 56; c.next = 0;
```

N.B. this is *not* the right way to do this kind of thing

So a is the head of the list; b is next; then c

Structures and Pointers

We can define a few values

```
struct intlist a, b, c;
a.val = 12; a.next = &b;
b.val = 34; b.next = &c;
c.val = 56; c.next = 0;
```

N.B. this is *not* the right way to do this kind of thing

So `a` is the head of the list; `b` is next; then `c`

We conventionally terminate the list with a 0 pointer as this turns out to be useful later (think about Boolean values)

Structures and Pointers

In fact, C defines a symbol `NULL` that is the same as zero, but visually indicates a null pointer, i.e., end of list:

```
c.next = NULL;
```

Structures and Pointers

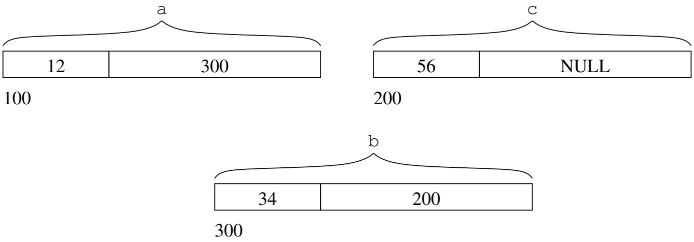
In fact, C defines a symbol NULL that is the same as zero, but visually indicates a null pointer, i.e., end of list:

```
c.next = NULL;
```

In fact, NULL is shorthand for `(void*)0`

Structures and Pointers

In memory, each instance of the structure contains the value and a pointer



Each instance can be anywhere in memory the system wants to put them; they are not necessarily in the order they appear in the code or the order they are created

Structures and Pointers

Note for geeks: there may well be alignment padding between the `int` and the pointer