Programming in C for CM30225

Russell Bradford

2020–2021
Programming in C

A bit of revision
The coursework for this Unit is a couple of programs written in C
The coursework for this Unit is a couple of programs written in C

The first tests shared memory parallelism
The coursework for this Unit is a couple of programs written in C.

The first tests shared memory parallelism.

The second tests distributed memory parallelism.
Programming in C

C is the language chosen because

• it is used a lot in high performance systems (alongside Fortran)
• it is low level, so you get to experience many of the issues of parallel programming directly
C is the language chosen because

- it is used a lot in high performance systems (alongside Fortran)
C is the language chosen because

- it is used a lot in high performance systems (alongside Fortran)
- it is low level, so you get to experience many of the issues of parallel programming directly
Programming in C

It is not the purpose of this Unit to teach you C.
Programming in C

It is not the purpose of this Unit to teach you C

It is something you should know already — or should find it easy to pick up
It is not the purpose of this Unit to teach you C

It is something you should know already — or should find it easy to pick up

So we shall just be covering the relevant features of C very quickly
If you are feeling uncertain about programming in C, as always the solution is to write C programs.
If you are feeling uncertain about programming in C, as always the solution is to write C programs

Lots of C programs
If you are feeling uncertain about programming in C, as always the solution is to write C programs

Lots of C programs

Don’t be afraid to experiment (or play!)
If you are feeling uncertain about programming in C, as always the solution is to write C programs.

Lots of C programs

Don’t be afraid to experiment (or play!)

You can’t break the computer...
Outline

• Why C?
• Resources
• What C looks like
• C is not object oriented
• C is procedural
• Functions
• #include
• Data Types

• Structures
• Arrays
• Pointers
• Memory management
• 2D arrays
• I/O
• Debugging
Why C?

C was designed in the early 70s as a language specifically for implementing an operating system (Unix).
Why C?

C was designed in the early 70s as a language specifically for implementing an operating system (Unix)

Thus its expertise in a range of low-level, close-to-the machine operations
Why C?

C was designed in the early 70s as a language specifically for implementing an operating system (Unix)

Thus its expertise in a range of low-level, close-to-the machine operations

This is what we need for close control over parallelism
C was designed in the early 70s as a language specifically for implementing an operating system (Unix)

Thus its expertise in a range of low-level, close-to-the machine operations

This is what we need for close control over parallelism

The original C, designed by Brian Kernighan and Dennis Ritchie (K&R C) was later modified and updated by the standards organisation ANSI
Why C?

Thus C has a long heritage, with many examples of its use everywhere
Why C?

Thus C has a long heritage, with many examples of its use everywhere.

It has a “you asked for you, you got it” approach.
Why C?

Thus C has a long heritage, with many examples of its use everywhere.

It has a “you asked for you, you got it” approach.

Meaning, for a good programmer, it does exactly what the programmer wants asks for.
Why C?

Thus C has a long heritage, with many examples of its use everywhere.

It has a “you asked for you, you got it” approach.

Meaning, for a good programmer, it does exactly what the programmer wants asks for.

Without trying to be “helpful”.
Why C?

Thus a good C program can be very fast and efficient
Why C?

Thus a good C program can be very fast and efficient

Comparable to writing in assembly for speed
Why C?

Thus a good C program can be very fast and efficient

Comparable to writing in assembly for speed

For our purposes, though, it does not hide any of the problems that parallel programming introduces
Thus a good C program can be very fast and efficient

Comparable to writing in assembly for speed

For our purposes, though, it does not hide any of the problems that parallel programming introduces

So you will get to see and recognise and fix the common parallelism problems
Why Not C?

But C is a very dangerous language
Why Not C?

But C is a very dangerous language

It has a “you asked for you, you got it” approach
Why Not C?

But C is a very dangerous language

It has a “you asked for you, you got it” approach

Meaning it is easy to write bad programs
Why Not C?

But C is a very dangerous language

It has a “you asked for you, you got it” approach

Meaning it is easy to write bad programs

A lot of the security issues with operating systems and apps are directly traceable to poor use of C
Why Not C?

But C is a very dangerous language

It has a “you asked for you, you got it” approach

Meaning it is easy to write bad programs

A lot of the security issues with operating systems and apps are directly traceable to poor use of C

Exercise. Read about buffer overflow bugs, use after free bugs, the Heartbleed bug
Resources

C is highly documented everywhere
C is highly documented everywhere

There are many books: just pick one that suits you!
C is highly documented everywhere
There are many books: just pick one that suits you!
Make sure it describes ANSI, not the old K&R C
C is highly documented everywhere

There are many books: just pick one that suits you!

Make sure it describes ANSI, not the old K&R C

Be just a little skeptical, though: many books contain trivial (and worse) errors of detail
And, of course, there are many resources on the Web
Resources

And, of course, there are many resources on the Web

From introductory to advanced, you can find any level of detail
And, of course, there are many resources on the Web.

From introductory to advanced, you can find any level of detail.

Be just a little skeptical, though: many Web pages contain trivial (and worse) errors of detail.
What C looks like

/*
multi-line
comment
*/
#include <stdio.h>

// single line comment
int main(void) {
    printf("hello world\n");

    return 0;
}

What C looks like

/* You should always comment your programs */

#include <stdio.h>

int factorial(int n) {
    if (n < 2) {
        return 1;
    }
    return n*factorial(n - 1);
}

int main(void) {
    printf("factorial(%d) = %d\n", 10, factorial(10));

    return 0;
}
What C looks like

You can use a text editor to write the code, then a command-line compile such as

```
c c -Wall -Wextra -o prog1 prog1.c
```
What C looks like

You can use a text editor to write the code, then a command-line compile such as

```
cc -Wall -Wextra -o prog1 prog1.c
```

Then run by

```
./prog1
```
What C looks like

Or you can use your favourite IDE and click on whatever button it wants to compile and then run
What C looks like

Make sure

• you always compile with all warnings on: in the GCC and Clang compilers you can use -Wall, -Wextra and even -Wconversion
• you always read and fix the warnings
What C looks like

Make sure

- you always compile with all warnings on: in the GCC and Clang compilers you can use `–Wall`, `–Wextra` and even `–Wconversion`
What C looks like

Make sure

- you always compile with all warnings on: in the GCC and Clang compilers you can use `-Wall`, `-Wextra` and even `-Wconversion`
- you always read and fix the warnings
What C looks like

The syntactic structure of C is very much like other languages you have come across
What C looks like

The syntactic structure of C is very much like other languages you have come across

Actually, it’s the other way!
What C looks like

The syntactic structure of C is very much like other languages you have come across

Actually, it’s the other way!

Many modern languages (C++, Java, JavaScript, C#, Perl, etc.) based their syntax on C
What C looks like

In one way this is good, as it means C will be familiar to look at
What C looks like

In one way this is good, as it means C will be familiar to look at

In another, it is bad, as C won’t behave as you might expect coming from other languages
What C looks like

In one way this is good, as it means C will be familiar to look at

In another, it is bad, as C won’t behave as you might expect coming from other languages

In particular, C is not object oriented
C is not object oriented

There are no classes in C
C is not object oriented

There are no classes in C

There are no objects in C
C is not object oriented

There are no classes in C
There are no objects in C
There are no methods in C
C is not object oriented

There are no classes in C

There are no objects in C

There are no methods in C

Just plain values and functions
C is not object oriented

There are no classes in C
There are no objects in C
There are no methods in C
Just plain values and functions

This is important to remember: so don’t try to program in C like you might program Java, JavaScript or Python
C is not object oriented

If you are lucky it might work
C is not object oriented

If you are lucky it might work

If you are unlucky it might appear to work
C is procedural

C is a *procedural* language
C is procedural

C is a *procedural* language

So the main way of structuring programs is by the use of functions (procedures)
C is a *procedural* language

So the main way of structuring programs is by the use of functions (procedures)

So: a C program is a big collection of functions that call each other
C is procedural

A large C program will typically consist of several files containing related functions
A large C program will typically consist of several files containing related functions

And will use more functions from libraries supplied by other people
A large C program will typically consist of several files containing related functions.

And will use more functions from libraries supplied by other people.

With separate compilation and linking of all the parts often managed by the IDE or something like a Makefile.
C is procedural

A large C program will typically consist of several files containing related functions

And will use more functions from libraries supplied by other people

With separate compilation and linking of all the parts often managed by the IDE or something like a Makefile

For our purposes: each coursework program will be small enough so a single file is perfectly sufficient
Functions

Functions are defined in a familiar way

```c
int factorial(int n) {
    if (n < 2) {
        return 1;
    }
    return n*factorial(n - 1);
}
```

declares and defines a function named `factorial` that takes an `int` and return an `int`
Functions

Functions are called in a familiar way

```c
int main(void) {
    printf("factorial(%d) = %d\n", 10, factorial(10));

    return 0;
}
```

the function `main` calls the functions `printf`, which takes as argument a call to the function `factorial`
Functions

main is the entry point of the program
Functions

main is the entry point of the program

Every program should contain exactly one function called main
Functions

main is the entry point of the program

Every program should contain exactly one function called `main`

When the program starts, it starts at `main`
Functions

main is the entry point of the program

Every program should contain exactly one function called main

When the program starts, it starts at main

When main exits, the program ends
Functions

main is the entry point of the program

Every program should contain exactly one function called main

When the program starts, it starts at main

When main exits, the program ends

Notice the type of main
Functions

In this example, it has no arguments: this is what `void` means.
Functions

In this example, it has no arguments: this is what \texttt{void} means

Other languages might allow you to write

\texttt{int main() ...}

for a function of no arguments
Functions

In this example, it has no arguments: this is what `void` means.

Other languages might allow you to write:

```c
int main() ...
```

for a function of no arguments.

ANSI C is different: you must be explicit about the lack of arguments.
In this example, it has no arguments: this is what `void` means.

Other languages might allow you to write

```c
int main() ...
```

for a function of no arguments

ANSI C is different: you must be explicit about the lack of arguments.

Later we shall see the other allowed way to use `main`
Functions

In this example, it has no arguments: this is what `void` means.

Other languages might allow you to write

```c
int main() ...
```

for a function of no arguments.

ANSI C is different: you must be explicit about the lack of arguments.

Later we shall see the other allowed way to use `main`.

Exercise. Find out what C does if you leave out the `void`. Then find out what is really happening.
The integer return value from \texttt{main} is passed up to the operating system when the program exits.
Functions

The integer return value from `main` is passed up to the operating system when the program exits.

The OS can choose what to do next.
The integer return value from `main` is passed up to the operating system when the program exits.

The OS can choose what to do next.

Typically used as a message to the shell or other UI to indicate success or failure of the program.
The integer return value from `main` is passed up to the operating system when the program exits.

The OS can choose what to do next.

Typically used as a message to the shell or other UI to indicate success or failure of the program.

0 is conventionally success, while different non-zero values indicate different kinds of failure.
Functions

The integer return value from `main` is passed up to the operating system when the program exits.

The OS can choose what to do next.

Typically used as a message to the shell or other UI to indicate success of failure of the program.

0 is conventionally success, while different non-zero values indicate different kinds of failure.

It is the choice of the programmer what these return codes mean.
Functions

C has

• local variables, optionally initialised
  `int n = 0, m;`

• for loops
  ```c
  for (i = 0; i < 10; i++) { ... }
  ```

• while loops
  ```c
  while (n < 10) { ... }
  ```

• Increment `n++` and decrement `x--`

• Conditionals
  ```c
  if (n == 0) { ... } else { ... }
  ```

• and much other familiar stuff
Functions

C has

- local variables, optionally initialised
  ```
  int n = 0, m;
  ```
C has

- local variables, optionally initialised
  
  ```c
  int n = 0, m;
  ```

- for loops
  
  ```c
  for (i = 0; i < 10; i++) { ... }
  ```

- while loops
  
  ```c
  while (n < 10) { ... }
  ```

- Increment `n++` and decrement `x--`

- Conditionals
  
  ```c
  if (n == 0) { ... } else { ... }
  ```

- and much other familiar stuff
C has

- local variables, optionally initialised
  ```
  int n = 0, m;
  ```
- for loops
  ```
  for (i = 0; i < 10; i++) { ... }
  ```
- while loops
  ```
  while (n < 10) { ... }
  ```
- Increment `n++` and decrement `x--`
- Conditionals
  ```
  if (n == 0) { ... } else { ... }
  ```
- and much other familiar stuff
C has

- local variables, optionally initialised
  ```c
  int n = 0, m;
  ```
- for loops
  ```c
  for (i = 0; i < 10; i++) { ... }
  ```
- while loops
  ```c
  while (n < 10) { ... }
  ```
- Increment `n++` and decrement `x--`
Functions

C has

- local variables, optionally initialised
  
  int n = 0, m;

- for loops
  
  for (i = 0; i < 10; i++) { ... }

- while loops
  
  while (n < 10) { ... }

- Increment n++ and decrement x--

- Conditionals
  
  if (n == 0) { ... } else { ... }
Functions

C has

- local variables, optionally initialised
  ```c
  int n = 0, m;
  ```

- for loops
  ```c
  for (i = 0; i < 10; i++) { ... } 
  ```

- while loops
  ```c
  while (n < 10) { ... } 
  ```

- Increment n++ and decrement x--

- Conditionals
  ```c
  if (n == 0) { ... } else { ... }
  ```

- and much other familiar stuff
Functions

When you use a function in C, its type must be known to the compiler.
When you use a function in C, its type must be known to the compiler.

E.g., `factorial` takes an `int` and returns an `int`. 
Functions

When you use a function in C, its type must be known to the compiler.

E.g., `factorial` takes an `int` and returns an `int`.

This is so the compiler can generate the right code to pass the values and receive the result.
Functions

Mostly, you would define the function before using it (e.g., factorial example above) and that is enough.
Functions

Mostly, you would define the function before using it (e.g., factorial example above) and that is enough.

Other times you need to use library functions written by someone else (e.g., printf).
Functions

Mostly, you would define the function before using it (e.g., \texttt{factorial} example above) and that is enough.

Other times you need to use library functions written by someone else (e.g., \texttt{printf})

You can do one of two things:
Functions

Declare the type of a function yourself:

```c
int factorial(int m);
```

just the first line, terminated by a semicolon, no code body
Functions

Declare the type of a function yourself:

```c
int factorial(int m);
```

just the first line, terminated by a semicolon, no code body

This gives the compiler the information it needs
Functions

Declare the type of a function yourself:

```
int factorial(int m);
```

just the first line, terminated by a semicolon, no code body

This gives the compiler the information it needs

The argument variable names in this declaration are irrelevant and can even be omitted: `int factorial(int);`
Functions

Declare the type of a function yourself:

int factorial(int m);

just the first line, terminated by a semicolon, no code body

This gives the compiler the information it needs

The argument variable names in this declaration are irrelevant and can even be omitted: int factorial(int);

Though some people argue you should include them for documentation purposes
Or use `include` to read in a file that contains the declaration(s)

```c
#include <stdio.h>
```
Or use `include` to read in a file that contains the declaration(s)

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

Somewhere in the system there will be a standard file named `stdio.h` that the compiler reads and inserts at that point.
Functions

Such a header file will contain the type declarations of many standard library functions, such as printf.
Functions

Such a *header file* will contain the type declarations of many standard library functions, such as `printf`.

Exercise. Find and look at the `stdio.h` file on your system. What is the type of `printf`?
Functions

Such a *header file* will contain the type declarations of many standard library functions, such as `printf`

Exercise. Find and look at the `stdio.h` file on your system. What is the type of `printf`?

Exercise. Read the documentation for various library functions (use `man` pages on Linux)
C natively supports very few types of data

- **char** and **unsigned char**: Usually 8 bits
- **short** and **unsigned short**: Usually 16 bits
- **int** and **unsigned int**: Usually 32 bits
- **long int** and **unsigned long int**, usually abbreviated to just **long** and **unsigned long**: Usually 64 bits

"Usually" as these are typical sizes, not set in the C standard.

We normally use **int** unless there are good reasons to use another size of integer.
Types

C natively supports very few types of data

Integers of various sizes

- char and unsigned char: Usually 8 bits
- short and unsigned short: Usually 16 bits
- int and unsigned int: Usually 32 bits
- long int and unsigned long int, usually abbreviated to just long and unsigned long: Usually 64 bits
C natively supports very few types of data

Integers of various sizes

- char and unsigned char: Usually 8 bits
- short and unsigned short: Usually 16 bits
- int and unsigned int: Usually 32 bits
- long int and unsigned long int, usually abbreviated to just long and unsigned long: Usually 64 bits

“Usually” as these are typical sizes, not set in the C standard
Types

C natively supports very few types of data

Integers of various sizes

- char and unsigned char: Usually 8 bits
- short and unsigned short: Usually 16 bits
- int and unsigned int: Usually 32 bits
- long int and unsigned long int, usually abbreviated to just long and unsigned long: Usually 64 bits

“Usually” as these are typical sizes, not set in the C standard

We normally use int unless there are good reasons to use another size of integer
Types

Floating point of various sizes

- float 32 bits “single precision”
- double 64 bits “double precision”
Types

Floating point of various sizes

- `float` 32 bits “single precision”
- `double` 64 bits “double precision”

And occasionally shorter (“half precision”) or longer (“quad precision”) variants
Types

Floating point of various sizes

- float 32 bits “single precision”
- double 64 bits “double precision”

And occasionally shorter (“half precision”) or longer (“quad precision”) variants

We normally use double unless there are good reasons to use another size of float
Types

Floating point of various sizes

- float 32 bits “single precision”
- double 64 bits “double precision”

And occasionally shorter (“half precision”) or longer (“quad precision”) variants

We normally use double unless there are good reasons to use another size of float

Exercise. 16 and 8 bit floats have recently become popular. Find out why
Types

C is weakly typed and by default does a lot of implicit coercion of types to other types.
C is weakly typed and by default does a lot of *implicit coercion* of types to other types.

For example poor code like:

```c
double x = 1;
```

might not even produce a warning from the compiler.
Types

C is weakly typed and by default does a lot of implicit coercion of types to other types

For example poor code like:

```c
double x = 1;
```

might not even produce a warning from the compiler

Or worse:
```c
int n = 0.1;
```
Types

C is weakly typed and by default does a lot of *implicit coercion* of types to other types.

For example poor code like:

```c
double x = 1;
```

might not even produce a warning from the compiler.

Or worse:

```c
int n = 0.1;
```

To make sure you catch these kinds of things, always put the decimal point in floating point constants and compile using `-Wconversion`.
Exercise. List all the things that are poor practice in this code:

```c
#include <stdio.h>

int main(void) {
    float den = 1.0/3.0;

    int div1 = (int)(4.0/den);
    int div2 = (int)(4/den);

    printf("%d and %d\n", div1, div2);

    return 0;
}
```
Other *composite* types are built from these basic types by using `struct`
#include <stdio.h>

struct intpair { // declare a new structure type
    int a;
    int d;
};

int main(void) {
    struct intpair p;
    p.a = 1;
    p.d = 99;

    printf("%d\n", p.a + p.d);

    return 0;
}
Structures

Note that

• you need to write `struct` everywhere, not just the type name
• `intpair` is not a class
• you can’t define methods
• `struct` values can be used just like any other type (e.g., passed to functions and so on)
Structures

Note that

- you need to write `struct` everywhere, not just the type name
Structures

Note that

- you need to write `struct` everywhere, not just the type name
- `intpair` is not a class
Structures

Note that

• you need to write \texttt{struct} everywhere, not just the type name
• \texttt{intpair} is not a class
• you can’t define methods
Structures

Note that

- you need to write `struct` everywhere, not just the type name
- `intpair` is not a class
- you can’t define methods
- `struct` values can be used just like any other type (e.g., passed to functions and so on)
Types in struct can be arbitrarily nested
Types in \texttt{struct} can be arbitrarily nested

As long as you don’t try to define a type that contains an instance of itself!
Structures

Types in `struct` can be arbitrarily nested

As long as you don’t try to define a type that contains an instance of itself!

Exercise. Read about `union`
Arrays

Another kind of composite type in C is the *array*
Arrays

Another kind of composite type in C is the *array*

These look much like arrays in other languages

```c
int a[100], b[100];
...
a[i] = a[i] + 2*b[i];
```
Arrays

Another kind of composite type in C is the *array*. These look much like arrays in other languages.

```c
int a[100], b[100];
...
a[i] = a[i] + 2*b[i];
```

Arrays are indexed from 0 to length-1.
Arrays

Another kind of composite type in C is the *array*

These look much like arrays in other languages

```c
int a[100], b[100];
...
a[i] = a[i] + 2*b[i];
```

Arrays are indexed from 0 to length-1

`a[0]` to `a[99]` in this example
Another kind of composite type in C is the *array*

These look much like arrays in other languages

```c
int a[100], b[100];
...
a[i] = a[i] + 2*b[i];
```

Arrays are indexed from 0 to length-1

a[0] to a[99] in this example

Given any C type, we can make arrays of that type
Arrays

Another kind of composite type in C is the *array*

These look much like arrays in other languages

```c
int a[100], b[100];
...
a[i] = a[i] + 2*b[i];
```

Arrays are indexed from 0 to length-1

a[0] to a[99] in this example

Given any C type, we can make arrays of that type

So we can have arrays of structures; and structures containing arrays
Arrays

Beware!

Array access is not checked in C
Beware!

Array access is not checked in C

This may well compile and do something:

```c
int a[100];
...
printf("Off the end: %d\n", a[100]);
```
This example is visually obvious, but in real code, of course, it’s much harder to spot

```c
int a[100];
...
// increment array
for (i = 0; i <= 100; i++) {
    a[i]++;
}
```
Arrays

This is one of C’s tradeoffs: speed (lack of checking) for safety
Arrays

This is one of C’s tradeoffs: speed (lack of checking) for safety
And it a big problem in many poorly-written C programs
Arrays

This is one of C’s tradeoffs: speed (lack of checking) for safety
And it a big problem in many poorly-written C programs
At worst, it can lead to a systems hack
Arrays

If you are lucky, the access will be to unmapped memory, and your program will crash
Arrays

If you are lucky, the access will be to unmapped memory, and your program will crash

If you are unlucky your program will seem to work
Arrays

If you are lucky, the access will be to unmapped memory, and your program will crash.

If you are unlucky your program will seem to work.

And give you incorrect results.
Arrays

Note: this problem is made worse in a parallel environment
Arrays

Note: this problem is made worse in a parallel environment where multiple threads can be writing to memory simultaneously.
Arrays

Occasionally, out of bounds array accesses can be useful in low-level code.
Arrays

Occasionally, out of bounds array accesses can be useful in low-level code

Don’t do it
Arrays

Occasionally, out of bounds array accesses can be useful in low-level code

Don’t do it

Be very careful about indexing into arrays
2D arrays are a little harder in C, and we need to revise pointers first
One of the things that makes C so useful is one of the things that some people find hard and thereby write broken programs.
Pointers

One of the things that makes C so useful is one of the things that some people find hard and thereby write broken programs.

A pointer is just an address of a memory location.
Pointers

One of the things that makes C so useful is one of the things that some people find hard and thereby write broken programs.

A pointer is just an address of a memory location.

Remember C was devised for low-level programming.
Pointers

One of the things that makes C so useful is one of the things that some people find hard and thereby write broken programs.

A pointer is just an address of a memory location.

Remember C was devised for low-level programming.

Pointers are sometimes called references in other languages.
int *a;
int b = 99;
a = &b;

declares a as a pointer to memory; the value there is to be interpreted as an int
Pointers

int *a;
int b = 99;
a = &b;

defines a as a pointer to memory; the value there is to be interpreted as an int

Initially, a is unset and points to nowhere in particular
int *a;
int b = 99;
a = &b;

debelares a as a pointer to memory; the value there is to be interpreted as an int

Initially, a is unset and points to nowhere in particular

We set a to the address of b using the reference operator &
Pointers

The value of $a$ is the address of some memory location
Pointers

The value of $a$ is the address of some memory location.

The location of where $b$ lives in memory.
Pointers

The value of \texttt{a} is the address of some memory location.

The location of where \texttt{b} lives in memory.

The value of \texttt{a} is not particularly interesting: the value it refers to is the interesting thing.
Pointers

We get the value that \(a\) points at using the \(*\) operator

```c
printf("The value \(a\) points to is %d\n", *a);
```

It knows to interpret the value it finds at this address as an `int` as the type of \(a\) is `int*` "pointer to int".

Exercise. Think about the games you can play by interpreting the same bits in memory as different types
We get the value that a points at using the * operator

printf("The value a points to is %d\n", *a);
**Pointers**

We get the value that a points at using the * operator

```c
printf("The value a points to is %d\n", *a);
```

It knows to interpret the value it finds at this address as an int as the type of a is int* “pointer to int”
We get the value that \texttt{a} points at using the \texttt{*} operator

\texttt{printf("The value a points to is \%d\n", \*a);}

It knows to interpret the value it finds at this address as an \texttt{int} as the type of \texttt{a} is \texttt{int*} “pointer to int”

Exercise. Think about the games you can play by interpreting the same bits in memory as different types
We may need pointers to structures:

```c
struct intpair {
    int a;
    int d;
};
...
struct intpair p;
...
struct intpair* pp = &p;
...
Pointers

In that case we can get at the struct values either using the * operator

\((*p).a = 123;\)

or, more tidily, the indirection operator ->

\(p->a = 123;\)

The -> is just a shorthand for the combined * and .
In that case we can get at the struct values either using the * operator

\((*p).a = 123;\)

or, more tidily, the indirection operator ->

\(p->a = 123;\)

The -> is just a shorthand for the combined * and .

The use of -> to access values in a (pointer to a) struct is different from many languages: fortunately the compiler will likely spot when you get it wrong
We can take the addresses of variables, and this has some use, but more commonly we want to allocate areas of memory and use them.
Memory Management

We can take the addresses of variables, and this has some use, but more commonly we want to allocate areas of memory and use them.

For this, we have the library functions malloc and free.
Memory Management

We can take the addresses of variables, and this has some use, but more commonly we want to allocate areas of memory and use them.

For this, we have the library functions `malloc` and `free`.

In C, memory is managed by the programmer, not the language.
Memory Management

This differs from other languages like Java, JavaScript, Python and so on that allocate and deallocate memory, or garbage collect inaccessible memory for you.
Memory Management

This differs from other languages like Java, JavaScript, Python and so on that allocate and deallocate memory, or garbage collect inaccessible memory for you.

Another of C’s tradeoffs
Memory Management

This differs from other languages like Java, JavaScript, Python and so on that allocate and deallocate memory, or garbage collect inaccessible memory for you.

Another of C’s tradeoffs

Another of C’s dangers
Memory Management

If the programmer gets the memory management wrong, the program is broken.
Memory Management

If the programmer gets the memory management wrong, the program is broken

If you are lucky, your program will crash near the code where you get it wrong
Memory Management

If the programmer gets the memory management wrong, the program is broken

If you are lucky, your program will crash near the code where you get it wrong

If you are a bit less lucky, your program will crash somewhere else
Memory Management

If the programmer gets the memory management wrong, the program is broken

If you are lucky, your program will crash near the code where you get it wrong

If you are a bit less lucky, your program will crash somewhere else

If you are unlucky, your program will seem to work
```c
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    // allocate memory for 10 doubles
    double *a = malloc(10*sizeof(double));

    // check allocation was successful
    if (a == NULL) { printf("malloc failed\n"); return 1; }

    // now we can use a just like an array
    int i;
    for (i = 0; i < 10; i++) { a[i] = (double)i; }

    for (i = 0; i < 10; i++) { printf("a[%d] = %f\n", i, 2.0*a[i]); }

    // be tidy and deallocate memory
    free(a);

    return 0;
}
```
Strings in C are just arrays of `char`, terminated by a 0 char
Strings in C are just arrays of `char`, terminated by a 0 char

```c
char *s = "hello";
```
allocates and initialises 6 bytes
Strings in C are just arrays of `char`, terminated by a 0 char

```c
char *s = "hello";
```
allocates and initialises 6 bytes

So `s[0]` is the character ’h’
Strings in C are just arrays of `char`, terminated by a 0 char.

```c
char *s = "hello";
```

allocates and initialises 6 bytes.

So `s[0]` is the character 'h'.

And `s[4]` is the character 'o'.

Strings in C are just arrays of char, terminated by a 0 char.

```
char *s = "hello";
```

allocates and initialises 6 bytes

So `s[0]` is the character ‘h’

And `s[4]` is the character ‘o’

And `s[5]` is the value 0
Strings in C are just arrays of `char`, terminated by a 0 char.

```c
char *s = "hello";
```

allocates and initialises 6 bytes

So `s[0]` is the character ’h’

And `s[4]` is the character ’o’

And `s[5]` is the value 0

Note `s[6]` is beyond the end of the memory allocated to this string.
#include <stdio.h>

int main(int argc, char *argv[]) {
    printf("You passed %d arguments to %s\n", argc, argv[0]);

    if (argc > 1) {
        printf("The first was %s\n", argv[1]);
    }

    return 0;
}
Pointers

The is the other use of `main`: it allows us to pass arguments into the program when we run: useful for when we want to run the same program many times with different values
Pointers

The is the other use of main: it allows us to pass arguments into the program when we run: useful for when we want to run the same program many times with different values

argc is the count: including 1 for the program name, which is counted as the 0th program argument
Pointers

The is the other use of `main`: it allows us to pass arguments into the program when we run: useful for when we want to run the same program many times with different values.

`argc` is the count: including 1 for the program name, which is counted as the 0th program argument.

`argv` is an array of `char*`, i.e., an array of strings.
% ./prog7
You passed 1 arguments to ./prog7

% ./prog7 asd ert
You passed 3 arguments to ./prog7
The first was asd
Exercise. Read the documentation for the library functions atoi and atof
You must be very careful in allocating and deallocating memory to avoid danger.
You must be very careful in allocating and deallocating memory to avoid danger.

Another place where bad programming can lead to broken programs.
Memory Management

You must be very careful in allocating and deallocating memory to avoid danger.

Another place where bad programming can lead to broken programs.

For the assignments, we need this for 2D arrays.
C, like many languages, only includes 1D arrays as part of the language
C, like many languages, only includes 1D arrays as part of the language.

So we need to do a little work for 2D arrays.
2D Arrays

C, like many languages, only includes 1D arrays as part of the language.

So we need to do a little work for 2D arrays.

There are several options available.
C, like many languages, only includes 1D arrays as part of the language.

So we need to do a little work for 2D arrays.

There are several options available.

All rely on the fact that given a type, we can make an array of that type.
2D Arrays

C, like many languages, only includes 1D arrays as part of the language

So we need to do a little work for 2D arrays

There are several options available

All rely on the fact that given a type, we can make an array of that type

So, given arrays we can make arrays of arrays
2D arrays, 1st version. Static allocation

```c
int a[100][100];
...
main(void) {
... a[i][j] ...
}
```
2D Arrays

Advantages:

- Easy to write

Disadvantages:

- You will have to recompile your code every time you change the size of the arrays
- Global variables in programs are dangerous
- Global variables in parallel programs are very dangerous
The above allocates an array on the global heap: you can also allocate on the stack (local variable):

```c
main(void) {
    int a[100][100];
    ... a[i][j] ...
}
```

Be careful doing this: preferably don’t do this for anything other than very small matrices
2D Arrays

This is bad because thread stacks are usually limited in size (e.g., 8MB)
This is bad because thread stacks are usually limited in size (e.g., 8MB)

A big array on the stack will extend beyond the end of the memory allocated for the stack
This is bad because thread stacks are usually limited in size (e.g., 8MB)

A big array on the stack will extend beyond the end of the memory allocated for the stack

Thus touching either unallocated memory, or memory allocated for something else
This is bad because thread stacks are usually limited in size (e.g., 8MB)

A big array on the stack will extend beyond the end of the memory allocated for the stack

Thus touching either unallocated memory, or memory allocated for something else

If you are lucky, etc.
The global heap can usually grow to as big as you need, so you can have huge arrays on the heap
The global heap can usually grow to as big as you need, so you can have huge arrays on the heap.

But global variables are bad.
The global heap can usually grow to as big as you need, so you can have huge arrays on the heap.

But global variables are bad.

`malloc` comes to our rescue.
int *a = malloc(10000*sizeof(int));
if (a == NULL) { ...do error case... }

allocates space for 10000 ints in the heap
2D Arrays

```c
int *a = malloc(10000*sizeof(int));
if (a == NULL) {
    ...do error case...
}
```

allocates space for 10000 ints in the heap

And the variable a is local to the block it is defined in
Note: C will not free memory referred to by `a` when we leave the block defining `a`
Note: C will not free memory referred to by \( a \) when we leave the block defining \( a \).

Another way of breaking code: a memory leak.
Note: C will not free memory referred to by \( a \) when we leave
the block defining \( a \)

Another way of breaking code: a memory leak

Memory from `malloc` is only ever deallocated by a call to `free`
2D Arrays

Note: C will not free memory referred to by a when we leave the block defining a

Another way of breaking code: a memory leak

Memory from malloc is only ever deallocated by a call to free

We have two things that are functionally separate: (1) an allocation of memory; (2) a reference to an allocation
2D Arrays

Note: C will not free memory referred to by a when we leave the block defining a

Another way of breaking code: a memory leak

Memory from malloc is only ever deallocated by a call to free

We have two things that are functionally separate: (1) an allocation of memory; (2) a reference to an allocation

A good source of bugs if we don’t match them up properly
2D Arrays

... { int *a = malloc(10000*sizeof(int));
    ... use a ...
} // memory still allocated here, but not accessible
// as variable a is out of scope

Memory leak
2D Arrays

```c
void foo(int n) {
    double *a = malloc(n*sizeof(double));
    if (a == NULL) { ...error... }

    ... use a ...

    free(a);
    // can’t use memory pointed to by a here
    // even though a still points at it
    ...
}

Use after free
```
2D Arrays

Notes:

• try to match free with malloc in your code
• only ever call free on a pointer given to you by malloc
• do not free a given pointer more than once
• free(a) does nothing to the value of a, it still points at the same memory (the memory is still there, it hasn't been destroyed!)
• do not use the memory pointed at by a pointer after it has been freed as the system might have allocated that memory to something else
• this is particularly true in parallel systems
Notes:

- try to match `free` with `malloc` in your code
2D Arrays

Notes:

- try to match `free` with `malloc` in your code
- only ever call `free` on a pointer given to you by `malloc`
2D Arrays

Notes:

- try to match free with malloc in your code
- only ever call free on a pointer given to you by malloc
- do not free a given pointer more than once
Notes:

- try to match \texttt{free} with \texttt{malloc} in your code
- only ever call \texttt{free} on a pointer given to you by \texttt{malloc}
- do not free a given pointer more than once
- \texttt{free(a)} does nothing to the value of a, it still points at the same memory (the memory is still there, it hasn’t been destroyed!)
2D Arrays

Notes:

- try to match `free` with `malloc` in your code
- only ever call `free` on a pointer given to you by `malloc`
- do not free a given pointer more than once
- `free(a)` does nothing to the value of `a`, it still points at the same memory (the memory is still there, it hasn’t been destroyed!)
- do not use the memory pointed at by a pointer after it has been `freed` as the system might have allocated that memory to something else
Notes:

• try to match `free` with `malloc` in your code
• only ever call `free` on a pointer given to you by `malloc`
• do not free a given pointer more than once
• `free(a)` does nothing to the value of `a`, it still points at the same memory (the memory is still there, it hasn’t been destroyed!)
• do not use the memory pointed at by a pointer after it has been freed as the system might have allocated that memory to something else
• this is particularly true in parallel systems
2D Arrays

2D Arrays, 2nd version. Use a 1D array in a 2D manner

```c
int *a = malloc(100*100*sizeof(double));

... a[100*i + j] ...
```

We allocate space for a $100 \times 100$ values, and index into the 1D array ourselves
2D Arrays

Advantages:

- Allocates on the heap
- Local scope
- Easy to allocate and free
- All the array memory is in one contiguous block (good for MPI transfers)

Disadvantages:

- Tedious code to index the array
2D Arrays

2D Arrays, 3rd version. A 2D array is just a list of pointers to 1D arrays

```c
int **a = malloc(100*sizeof(double*));
if (a == NULL) ...

for (i = 0; i < 100; i++) {
    a[i] = malloc(100*sizeof(double));
    if (a[i] == NULL) ...
}

... a[i][j] ...
```

We allocate space for 100 pointers and aim those pointers at 100 1D arrays of 100 values
2D Arrays

Advantages:

- Allocates on the heap
- Local scope
- (allows non-rectangular arrays)

Disadvantages:

- Fiddly code using pointers to pointers
- uses lots of mallocs
- Rows of array not contiguous in memory
2D Arrays

Advantages:

- Allocates on the heap
- Local scope
- (allows non-rectangular arrays)

Disadvantages:

- Fiddly code using pointers to pointers
- uses lots of mallocs
- Rows of array not contiguous in memory

Exercise. Write the code to free all the memory
2D Arrays

2D Arrays, 4th version. A 2D array is just a list of pointers to memory

int **a = malloc(100*sizeof(double*));
if (a == NULL) ...
int *buf = malloc(100*100*sizeof(double));
if (buf == NULL) ...

for (i = 0; i < 100; i++) {
    a[i] = buf + 100*i;
}

... a[i][j] ...

2D Arrays

Advantages:

- Allocates on the heap
- Local scope
- Uses just two `malloc`
- Array memory in one contiguous block
- (allows non-rectangular arrays)

Disadvantages:

- Fiddly code using arithmetic on pointers
2D Arrays

Advantages:

- Allocates on the heap
- Local scope
- Uses just two mallocs
- Array memory in one contiguous block
- (allows non-rectangular arrays)

Disadvantages:

- Fiddly code using arithmetic on pointers

Exercise. Check that I have the i's and j's the right way around in the above examples
2D Arrays

2D Arrays, 5th version. Pun on C types

void fun(int n, int m, double mat[n][m]) {
    ...
}

int r = 3;
int c = 5;

double **mat = malloc(r*c*sizeof(double));
if (mat == NULL) ... 

fun(r, c, (double(*)[c])mat);
2D Arrays

Advantages:

- Allocates on the heap
- Local scope
- Uses just one `malloc`
- Array memory in one contiguous block
- Uses less memory

Disadvantages:

- You can only use indexing on `mat` inside a function call
- You need to understand C type casting
2D Arrays

Note that `malloc` only allocates memory, it does not initialise it to anything.
2D Arrays

Note that `malloc` only allocates memory, it does not initialise it to anything.

So the values in memory can be junk.
2D Arrays

Note that `malloc` only allocates memory, it does not initialise it to anything.

So the values in memory can be junk.

There is the `calloc` function that allocates memory (using `malloc`) and then sets it all to 0.
2D Arrays

Note that `malloc` only allocates memory, it does not initialise it to anything.

So the values in memory can be junk.

There is the `calloc` function that allocates memory (using `malloc`) and then sets it all to 0.

Good for initialising data arrays.
2D Arrays

Note that `malloc` only allocates memory, it does not initialise it to anything.

So the values in memory can be junk.

There is the `calloc` function that allocates memory (using `malloc`) and then sets it all to 0.

Good for initialising data arrays.

Not good (a waste of time) if you need to allocate, then immediately overwrite, as in the `malloc` of `a` in the 4th version above.
Around 70 percent of all the vulnerabilities in Microsoft products addressed through a security update each year are memory safety issues
Matt Miller, Microsoft security engineer, Feb 2019
We have seen `printf` to print text, with various formatting like `%d` for `int`, `%f` for `double`, `%s` for strings.
We have seen printf to print text, with various formatting like %d for int, %f for double, %s for strings

Exercise. Read the documentation for printf
I/O

We have seen `printf` to print text, with various formatting like `%d` for `int`, `%f` for `double`, `%s` for strings

Exercise. Read the documentation for `printf`

Exercise. The reverse is `scanf` to read values from text. Read about it. Don’t read from the keyboard in the assignments. You’ll see why
Files in C are read and written via the FILE type (just structs underneath)
Files in C are read and written via the FILE type (just structs underneath)

- `FILE *fr = fopen("somefilename", "r");`
  - opens the named file for reading. It returns a FILE*
Files in C are read and written via the FILE type (just structs underneath)

- FILE *fr = fopen("somefilename", "r");
  opens the named file for reading. It returns a FILE*
- FILE *fw = fopen("anotherfilename", "w");
  opens the named file for writing. It returns a FILE*
• double a[10];
  fread(a, sizeof(double), 10, fr);

read from the file opened as fr 10 double-sized items into the memory indicated by a, an array in this example

Note: this does not check that there is enough memory allocated at a to store the items and so can write beyond the end of the array
double a[10];
fwrite(a, sizeof(double), 10, fw);

write 10 double-sized items from the memory indicated by a to the file opened as fw

Note: this does not check that the memory allocated at a was that size before reading it and so can read beyond the end of the array
• `fclose(fr);` (similarly `fclose(fw)`) close the file

In the usual way, you should close files cleanly before exiting your program
Exercise. Read the documentation for these functions, in particular their return values (which we have improperly ignored in the above)

Exercise. Read about *unbuffered* file I/O using open, close, read and write
Exercise. Read the documentation for these functions, in particular their return values (which we have improperly ignored in the above)

Exercise. Read about *unbuffered* file I/O using `open`, `close`, `read` and `write`

Note: using files in a parallel system can be interesting…
Debugging C programs can be difficult.
Debugging

Debugging C programs can be difficult

Debugging parallel programs is infinitely more so
Debugging

Simple approach: put `printf`s at appropriate points in your code
Debugging

Simple approach: put `printf`s at appropriate points in your code.

Print out the values of things to see if they are what you would expect; or that things are happening in the order you want.
Debugging

Simple approach: put `printf`s at appropriate points in your code

Print out the values of things to see if they are what you would expect; or that things are happening in the order you want

This is surprisingly good, particularly in parallel systems
Debugging

Use a debugger, such as gdb
Use a debugger, such as gdb
Or whatever your IDE provides
Debugging

Use a debugger, such as `gdb`

Or whatever your IDE provides

You will want to compile your code with the `-g` flag to insert extra debugging information
Use a debugger, such as `gdb`

Or whatever your IDE provides

You will want to compile your code with the `-g` flag to insert extra debugging information

Using a debugger is hard in parallel systems, particularly on a cluster where you might not have interactive access to your running code
You might want to write and debug your code on your own computer before moving it to the cluster.
Debugging

You might want to write and debug your code on your own computer before moving it to the cluster.

This will find some of the biggest bugs, but will miss some that only arise on true parallel systems.
Debugging

You might want to write and debug your code on your own computer before moving it to the cluster.

This will find some of the biggest bugs, but will miss some that only arise on true parallel systems.

Exercise. Read about the Allinea system on the Bath cluster.
Debugging

Use `valgrind` to check for memory access errors

As mentioned, C does not check for dumb memory accesses (beyond the ends of arrays, using memory that has been freed and so on). The `valgrind` program runs your code inside an interpreter and checks all memory accesses. It is helpful to compile with `-g` for this, too.
Use `valgrind` to check for memory access errors.

As mentioned, C does not check for dumb memory accesses (beyond the ends of arrays, using memory that has been freed and so on).
Use `valgrind` to check for memory access errors.

As mentioned, C does not check for dumb memory accesses (beyond the ends of arrays, using memory that has been freed and so on).

The `valgrind` program runs your code inside an interpreter and checks all memory accesses.
Debugging

Use `valgrind` to check for memory access errors

As mentioned, C does not check for dumb memory accesses (beyond the ends of arrays, using memory that has been `freed` and so on)

The `valgrind` program runs your code inside an interpreter and checks all memory accesses

It is helpful to compile with `-g` for this, too
Debugging

This runs much more slowly, but will point out anything that looks dodgy in your use of memory
Debugging

This runs much more slowly, but will point out anything that looks dodgy in your use of memory.

Again, using this is hard in parallel systems, so you might want to debug on your own computer first.