# Advanced Class Lesson 1

Concepts of the C Programming Language

# Introduction

- Hi, I’m Yi Wei
- I’m your instructor for this thing

# Introduction

- C is a low-level programming language etc. etc.
- Who cares about C's history. Google it yourself if you're curious
- C is just a medium through which I'll introduce you to low-level programming
- C is a simple (**Not easy**) programming language
- In general, it provides you with only a handful of tools to get started. Everything else, you must do yourself.

# Compilation

- C is a compiled language
- What this means is that C must be compiled into an executable file (.exe on windows only) before you can run it
- The process makes use of a software called (Surprise, surprise) a compiler.
- With C, the 3 major compilers are GCC, Clang, and Visual C++
- You don’t have to know the difference between them, other than that GCC is the best.
- For this lesson, we’ll be using Godbolt.org.

# Types

- The basic types provided to you in C are called ‘primitives’.
- These can be split into 3 main categories
  - Integral
  - Floating Point
  - ???
- All types have a set size in bytes, unlike in Python.
- There are also type modifiers that we will get into, soon.
- Note: **There are no string types**.

# Integer

- Integer types just store whole numbers. (Unless you're using j*vascript)
- They do so by just counting in binary. (TODO go through this if someone doesn't understand)
- There are many types within this category.
- Why is that?
- To represent integers of different sizes.
- A computer has limited memory, so to optimise space usage, we don't want to be using 64 bits for everything

# Integer (1)

- An `int` stores data in at least 16 bits (2 bytes)
  - Stores from -32,768 to 32,767
- A `long` (Shorthand for ‘long int’) stores data in at least 32 bits (4 bytes)
  - From -2,147,483,648 to 2,147,483,647
- Guess what’s longer than a long int?

# Integer (10)

- A `long long`, of course.
- This stores data in at least 64 bits (8 bytes)
  - From -(2^63) to (2^63)-1
- A short stores data in at least 16 bits, too. It’s just an alias for int.
- Notice the ‘at least’?
- It’s there because for some reason, ‘int’ is now 32 bits usually, varying from system to system.

# Integer (11)

- What about 1 byte?
- An 1-byte integer is stored as a `char`
- Why is it called a `char`?
- Because of the way a PC encodes text.
- Your PC has to know what character to display.
- You can’t just upload text as images, and then tell it to display them.
- So, we just label each character with a number, and then tell the computer ‘if you see {n}, display character {x}’.
- Very fun. For your sanity, don’t research how this works with Chinese characters. However, it’s called UTF-8 if you really want to know.

# Integer (100)

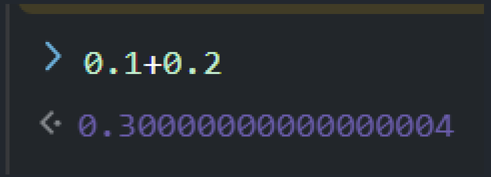
- Integers can be modified with `unsigned`, `const`, `*`, and `[]`.
- We'll ignore `*` and `[]` for now, because they relate to pointers (Not something you need to know now).
- `unsigned` means that we are only allowed to represent integers >=0
- This is to extend the range of numbers that we can represent, if we know for sure we don't have a negative number.
- `const` just means that you can only set its value once (At least, without delving into undefined behaviour)

# Digression: Undefined behaviour

- Something that is not defined within the specifications of the language.
- Can result in errors and crashes.
- Guess what?
- It’s everywhere in C.
- Avoid it where possible.
- Good luck, though. You don’t always know when it’s undefined behaviour :thumbsup:

# Floating Point

- Floating point numbers are just real numbers
- It’s basically just scientific notation for real numbers, but in base 2. (i.e., it’s stored as $m \times 2^E$).
- Most programming languages (Excl. python) use this format as their default way of storing real numbers.
- This format sacrifices some precision, but also increases the range of the values a lot
- An example is the common j*vascript criticism (which for once isn’t j*vascript’s fault).



# Floating Point (1)

- A `float` uses 32 bits.
  - It can store values between $\pm 340282346638528859811704183484516925440.0$ with low precision.
- A `double` uses 64 bits.
  - It can store values between $\pm$(A number so absurdly large that wolfram alpha refuses to process it) with decent precision.
- Note that in floating point land, the further you get from 0 in either direction, the less precise your numbers become.

# Floating Point (1) (1)

- Floats can be modified with `const`, `*`, and `[]` only.
- Floats are inherently signed, so you cannot slap an `unsigned` on it.
- `const` does the same thing that it does to an int.

# ???

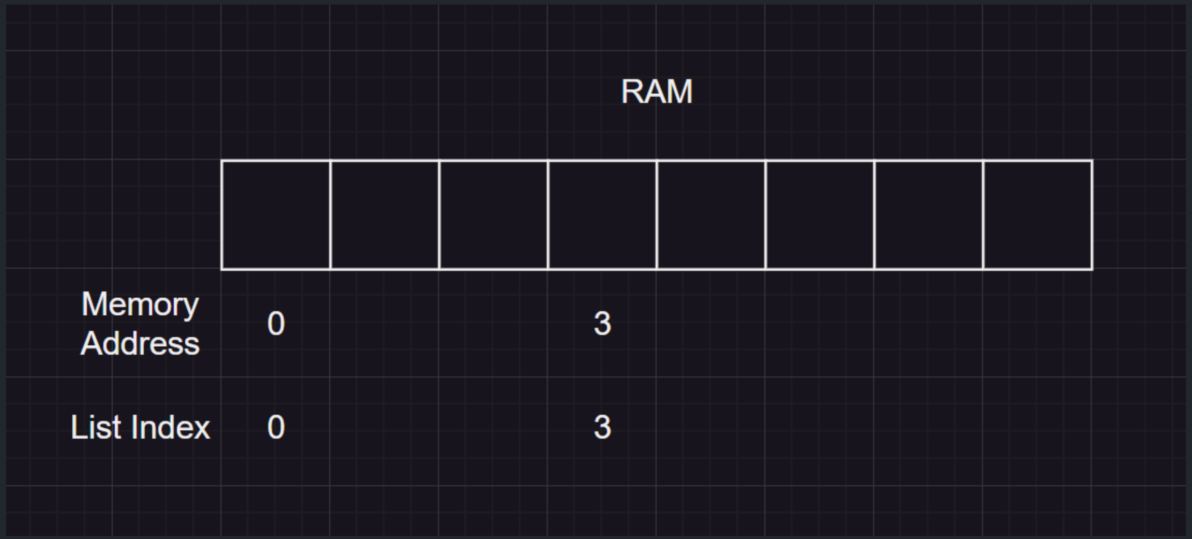
- I have no clue what to name this category.
- The only type in this category is `void`.
- `void` represents that something has no type.
- Its main use is in subroutines and arbitrary type representation.
- More on that in the future.

# Pointers

- A pointer is just a variable that stores the address to a value.
- To understand the true use of pointers, you need to understand memory.

# Memory

- Memory is basically just your RAM.
- If you've used any common programming language, you'll be familiar with arrays or 'lists', as they are called in python.
- You can think of RAM as basically just an array of bytes.
- If you're accessing the data at memory address 3, you can think of it as accessing the 4th element of your list.



# Memory (1)

- Thing is, you can store data in more than 8 bytes.
- So, your code will have to specify how much memory to take.
- If you specify that your data is stored in 2 bytes, your PC will basically just take the byte at the current index and next index of your list, then concatenate (Not add) them together.
- (To concatenate 15 and 20 together, you just stick them next to each other, and it becomes 1520)

# Pointers*

- A pointer is just a variable type that's made specifically to store the addresses that you're using to store things in memory.
- A pointer is denoted by the '*' modifier.
- For example, an `int*` is a pointer to an integer.

# Pointers**

- Remember how you must specify the number of bytes that you're going to access from memory?
- This is how you tell your CPU how many bytes it should read from memory.
- To access the data stored at the address of a pointer, you use the * modifier again, but this time on the pointer itself.
- To find the address of a variable, you use the & operator.

# Pointers***

```
// arbitrary code, does not actually run
int aoeu = 65;
int* boeu = &aoeu; // Gets the address of aoeu.
int x = *boeu; // x now equals to 65.
*boeu = 42;
x = *boeu; // x now equals to 42.
```

# Pointers***

- Why use pointers?
- Using a pointer is like telling someone where your locker is, so that they can go in and take (Or put) stacks of foolscap whenever.
- This means that everyone can access that same locker at once.
- Contrary to this, using actual values is like passing someone a box that mimics your locker in the number of stacks of foolscap in it.

# Pointers****

- When you do this, you create a copy of everything that you store within the locker (Which is honestly not a problem, unless you're trying to pass a 10MB struct)
- Putting a pointer into a subroutine is called **passing by reference**, whilst passing a value into a subroutine is called **passing by value**.
- Pointers can also be interpreted as a box. More on that when I teach you about structs.

# Arrays

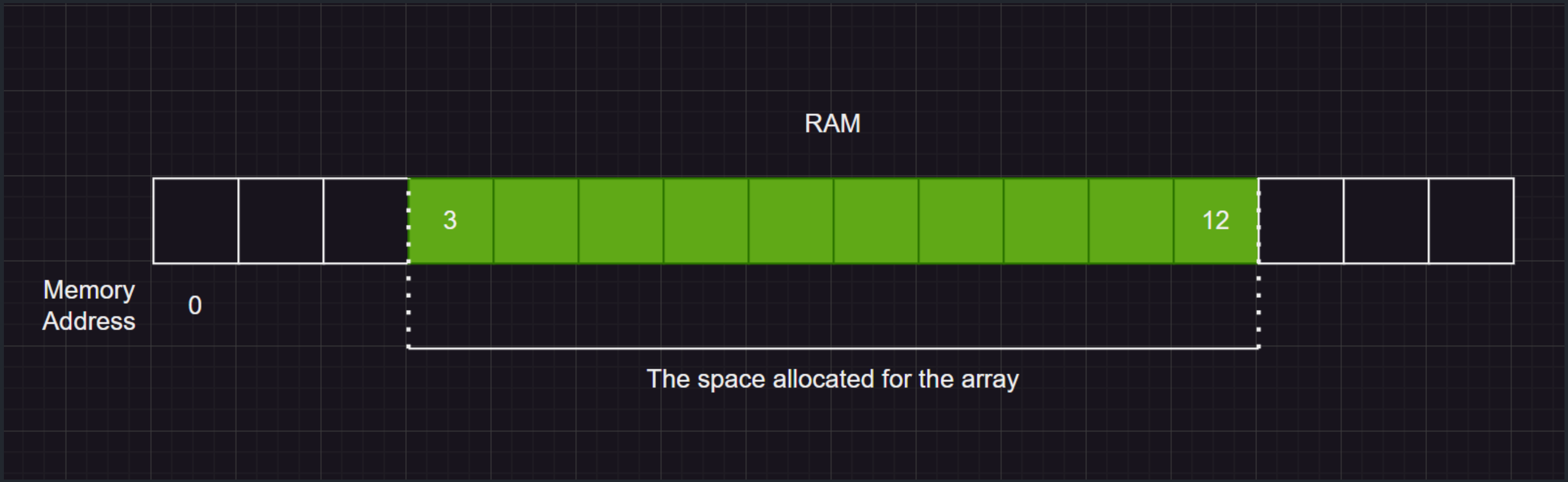
- Arrays are basically what you’re familiar with.
- In Python, they’re also called lists.
- However, in C, they can only contain elements of the same type.
- **Arrays in C are just pointers**. More on that later.
- For those who know at least a bit of Python or whatever, you’ll know that ‘[]’ is the array access operator.
- In C, ‘[]’ is sort of a quality-of-life feature, rather than a necessity.

# Arrays[0]

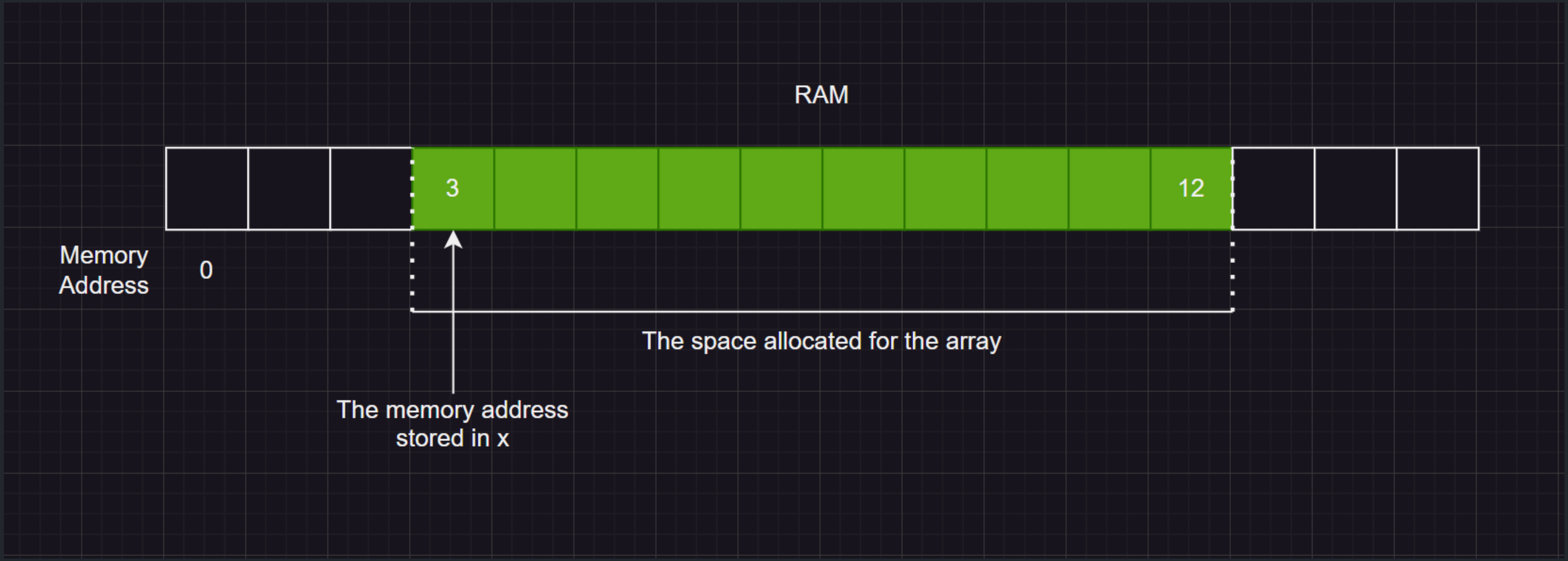
- Let’s say I define an array of 5 ints.
- This is done with the [] modifier (Right).
- What this does is that your program just decides ‘these 10 bytes are belonging to me’, and now, x is just a pointer to the first value of those 10 bytes.

```
int x[5];
// Declares an
// array of 5
// elements named
// x
```

# Arrays[1]



# Arrays[2]

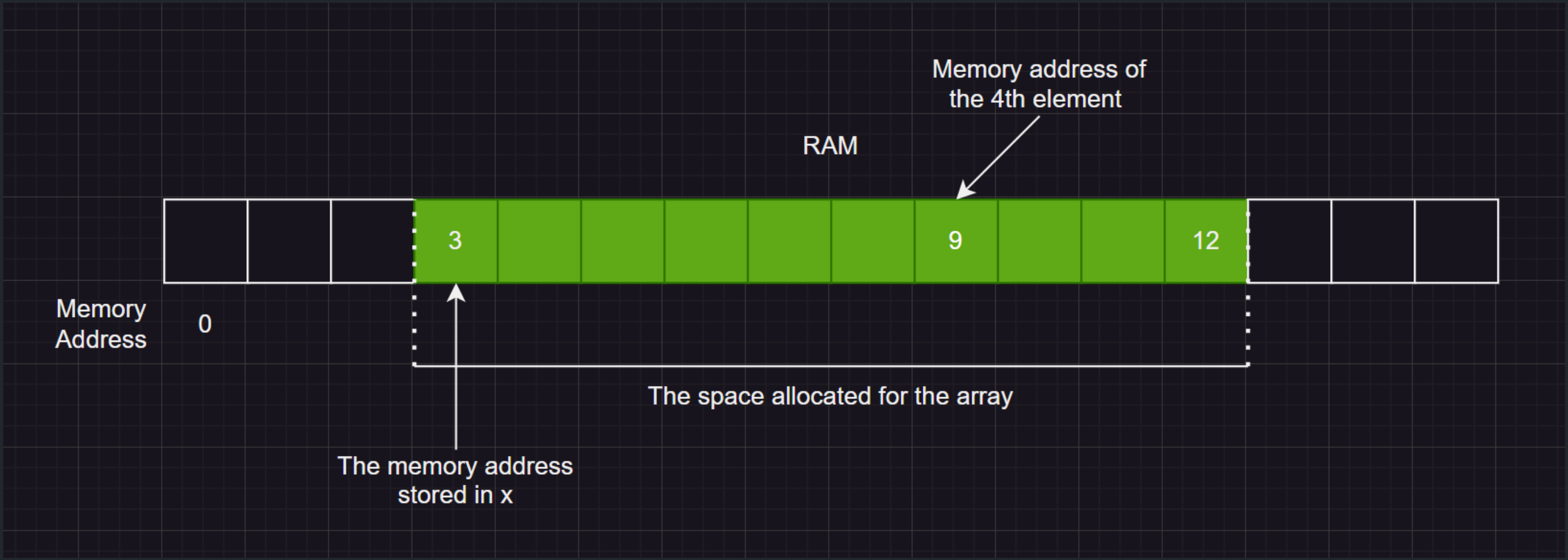

RAM
3
12
Memory
Address
0
The space allocated for the array
The memory address
stored in x

# Arrays[3]

- So how do you access an element in an array?
- You use the `[]` operator on the array itself.
- Python and j*vascript users will be familiar.

```
int x[5];
x[3] = 0;
// Sets the 4th element
// to 0.
```

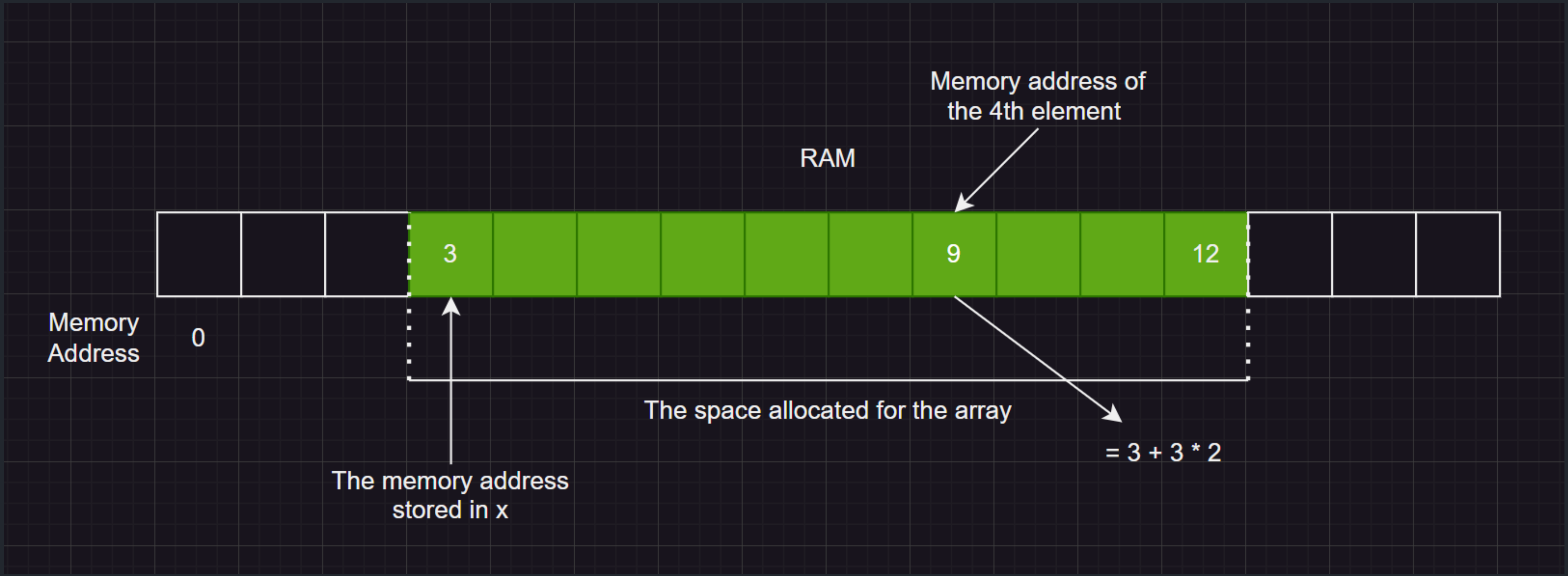
# Arrays[4]


Memory address of
the 4th element
RAM
3
9
12
Memory
Address
0
The space allocated for the array
The memory address
stored in x

# Arrays[5]

- Something you’ll noticed though, is that you can find the memory address in terms of the index and a pointer to the array.

# Arrays[6]



Memory address of
the 4th element
RAM
3
9
12
Memory
Address
0
The space allocated for the array
= 3 + 3 * 2
The memory address
stored in x

# Arrays[7]

- More formally, you can generalise this equation as:

$$E_n = *\,(A_o + kn)$$

Where,

$E_n$ is the nth element of the list,

$A_o$ is the address of the array,

$k$ is the size of the elements in the array,

$n$ is the value in the subscript operator

# Arrays[8]

- This also explains why to access the first element of a list, you use an index of 0.
- Yay

# Arrays[9]

- A string is just an array of chars.
- It makes sense, right?
- That’s literally what strings are.
- Since arrays are pointers, you can define strings as char pointers.
- The problem is now that you don’t know how big the array is.

# Arrays[10]

- So, all strings in C have a special character called a **null terminator** at the end.
- That way, you can just go through the string one by one until you hit that character.
- This causes undefined behaviour when you don’t have that character to end the string.
- It can also cause undefined behaviour when you have that special character in the middle of the string.
- So basically, strings are hard.

# Structs

- Arrays provide you with the ability to store multiple values of a single type efficiently.
- However, what if you want to store multiple values of different types?
- This is where structs come to help.

# Structs.first

- A struct is basically a type that is defined by the user.
- If you’ve studied OOP before, this is not an object.
- An object contains **both** behaviour **and** data.
- However, a struct can contain only data (Without going into undefined behaviour).
- It’s just a way to bundle data such that it’s easy to access each field individually.

# Structs.second

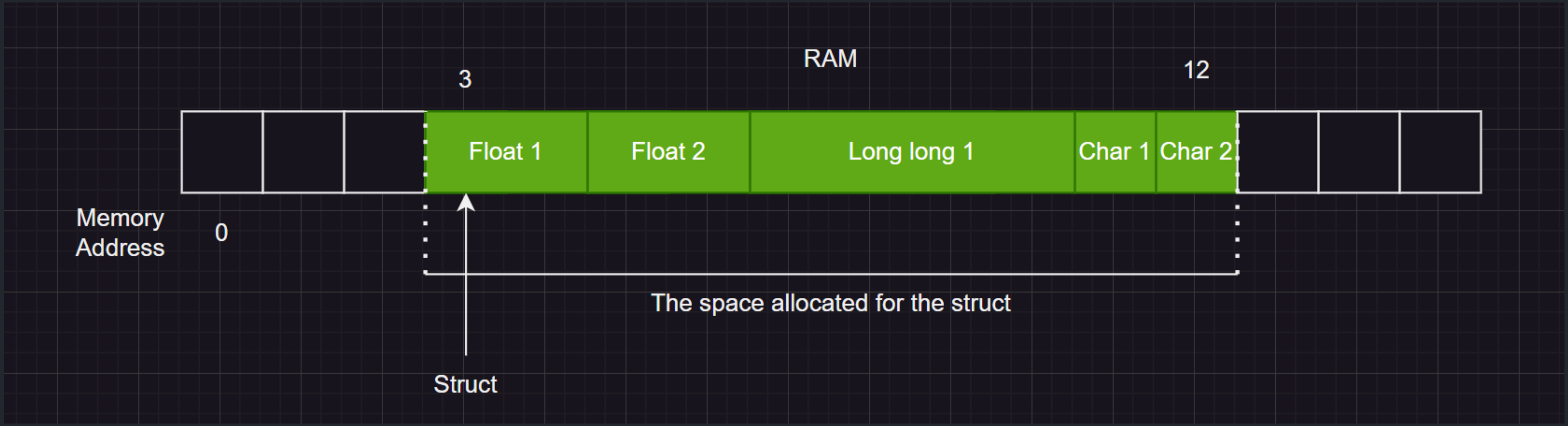
- The data stored within structs is called a **field**.
- Fields can contain primitives, pointers, and other structs.
- You can access each field with a dot operator.

# Structs.third

- Why use structs?
- Structs are an easy way to pass large amounts of data into subroutines.
- Just create a struct that stores all your data, and pass a pointer to that struct into the subroutine.
- That’s much better than passing every single data field into the subroutine manually, right?

# Structs.fourth

- Structs are, in reality, the same as arrays.
- Let’s say that the struct stores 2 floats, 1 long long, and 2 chars.




- This is obviously not how the data is actually laid out in memory.

# Structs.fifth

- And with that, that means that structs are just pointers…
- Or not.
- In reality, they can’t be because structs don’t have a uniform size for the array.
- So yay, less pointer bullshit.

# Subroutines

- Subroutines are what Python and J*vascript etc. tend to call ‘Functions’.
- Subroutines have 2 types
  - Functions
  - Procedures
- The only difference between them is that functions return a value, whilst procedures do not.

# Subroutines ()

- Guess what?
- Subroutines are just pointers, too.
- Calling a subroutine is equivalent to making the program execute code from where the subroutine is located in memory, then returning to its original position after returning.
- Because of that, it is also possible to pass subroutines into other subroutines and return subroutines from functions.

# Subroutines () ()

- Subroutines are a common form of **abstraction**.
- They are usually used to expose a specific **interface** for the user to minimise the mental load on the programmer.

# Interfaces

- Interfaces are methods of using something that aim to minimise the number of things that you must keep track of.
- The most common interface is the Graphical User Interface (GUI).
- It lets users access the features of the application <u>without needing to know about what is happening internally</u>.
- Interfaces need not be complex.
- The array access operator, structs and subroutines are also interfaces, despite being simple in nature.

# If-else statements

- Elementary
- Left as an exercise to the viewers

# Loops

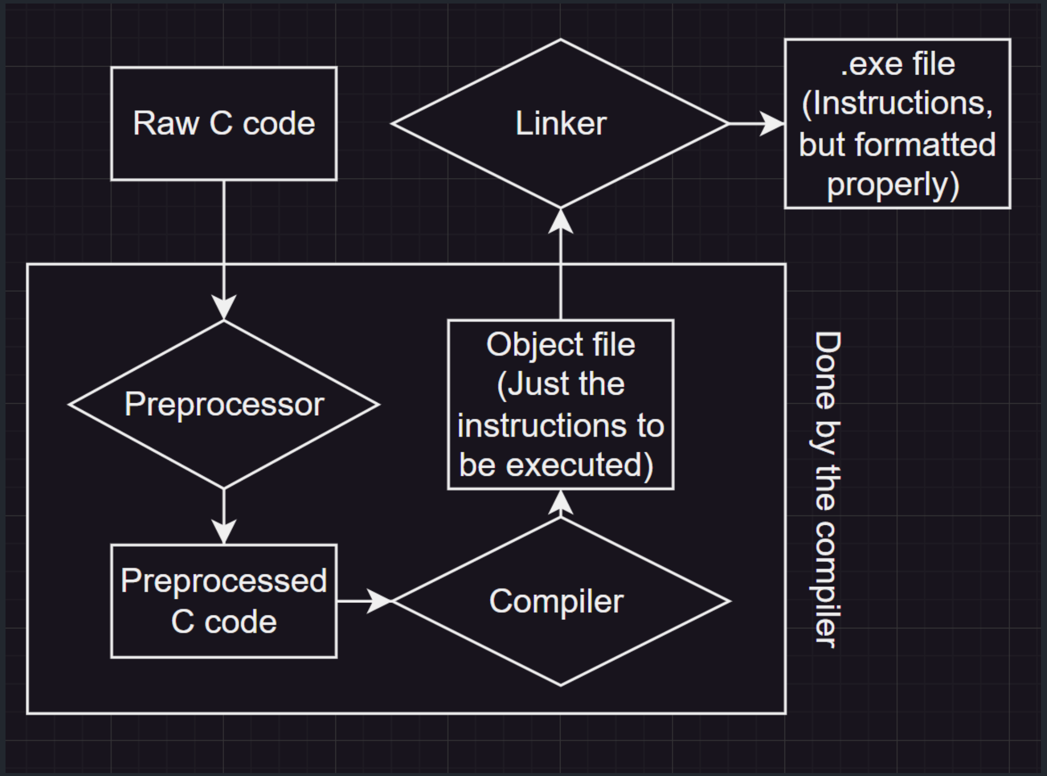
- Elementary
- Left as an exercise to the user

# Switch statement

- Elementary
- Left as an exercise to the viewer

# Preprocessor directives

- Earlier, I mentioned that C is compiled into an .exe.
- However, before that can happen, there are a few steps required.



# Preprocessor directives 1

- The preprocessor is where simple steps can be done before you execute the .exe.
- (At least, that’s how it was until last year, when ANSI and ISO decided that there would be a new keyword that allows more complex tasks to be executed before compilation)
- All preprocessor directives start with a #
- At no point do preprocessor directives affect the actual source code.
- There are officially 3 categories of preprocessor directives
- There are 4 if you’re brain dead

# Macros

- Preprocessor macros are just substituted in at the preprocessing stage.
- They are defined with `#define`, and undefined with `#undef`.
- There are 2 types of macros: Macros, and Parameterised macros.

# Macros (Macros)

- You can think of them as variables
- However, they are basically searched and replaced during the preprocessing stage.
- That means that you cannot update their values while you execute the .exe.

# Macros (Parameterised Macros)

- You can define macros with parameters, just like functions
- However, one thing that is different is that types are not checked.
- This can cause confusing errors.
- Another issue is that the compiler literally copies and pastes the parameters, **without evaluating them**.
- This means that you can unintentionally execute a subroutine more than once on the same variable.

# File Inclusion

- In Python and J*vascript, you import libraries.
- However, in C, it’s different.
- You use #include to ‘include’ a ‘header’ file in your code.

# Header Files

- Files that contain the **signature** of the subroutines, types and variables that you want to use.
- A signature basically tells the compiler ‘What type goes in this subroutine?’, ‘What fields does a struct have?’ and ‘What type is this variable?’
- They are required for when you want to use libraries that have too much code to compile yourself in a reasonable amount of time.

# File inclusion 1

- ‘Including’ a file basically just opens up that file, and pastes it into your source file.
- Because a header file is just another text file, you can literally just write the full source code of what you want to include into that text file.
- Libraries that use this method are called ‘header-only’ libraries.

# Conditional Compilation

- These directives tell your compiler to basically delete parts of your source code at compilation time.
- These include `#if`, `#ifdef`, `#ifndef`, `#else`, `#endif`.
- #if works just like an if statement. If the condition passed into it is false, it just deletes the part of the code between it and either the next else or endif.
- #ifdef works similarly, but it checks if a macro with that name is defined, instead.
- #ifndef is just the negated version of it.

# Pragmas

- Undefined behaviour
- Compiler specific
- Just avoid them where possible
- They do things like set warnings, do single inclusion checks, and send messages on the compiler’s side

# Translation units

- A translation unit is a concept that is crucial to figure out why pointers are used so often.
- A translation unit consists of **one** source file, and everything that has been included into the source file.
- Each translation unit is compiled to an ‘object file’.
- An object file is basically just raw instructions

# Translation units.o

- A simplified example is a baking recipe.
- A recipe usually consists of 4+ parts, like the exposition, long-ass story about their grandmother, ingredients, and steps.
- An object file is like the raw text of each part, excluding any and all formatting.
- These parts need to be linked together to form a full recipe (Though maybe not the story)
- This is what the linker does.
- In the meantime, it also formats the code to execute on the current machine.

# Translation units.o.o

- Translation units are connected through header files.
- However, header files usually do not contain the actual implementation of structs or subroutines.
- In the case of structs, this means that your compiler **may not know** what size the struct is.
- This means that your compiler cannot tell the OS how much memory it needs to allocate for the program, so it cannot allocate that memory.
- In this way, a pointer is necessary as it acts as a wire.

# The Wire

- Let’s say you have a TV screen connected to a massive box.
- The TV has a wire that enters the box, but you can’t tell what’s inside the box.
- However, you can interact with the TV as per usual, because there’s still buttons and a TV control.
- This means that you can now interact with the TV without knowing the size of the mechanism that controls it.

# Translation units.o.o.o

- In this case, the big box would represent memory, and the wire would be the pointer that leads your inputs into the mechanism that controls it.
- In practise, there would be a subroutine (From a different translation unit) that allocates memory for that struct, which makes the pointer an interface to interact with that struct.

# void* pointers

- Void pointers are used to store, pass, and use things of variable types.
- It’s a void pointer because it’s not a pointer type that’s used anywhere else, to minimise the chances of people mistaking it for an actual pointer.
- Before a void pointer can be used, you must cast it into the type that you actually need.

# Casting

- Casting is the process of converting a thing of one type into another type.
- For example, casting an int to a double means that you're making it a decimal number instead of an integer.

# *pointers

- Because you must cast it every time, it’s also very easy to fuck up.
- Let’s say a type is an int.
- You take its address and stick it into a void pointer.
- However, you mistake it for a long long.
- So, you cast it over, deference it and– Segmentation fault, core dumped.
- How fun.

# The end

- Click to add Text