Diploma in

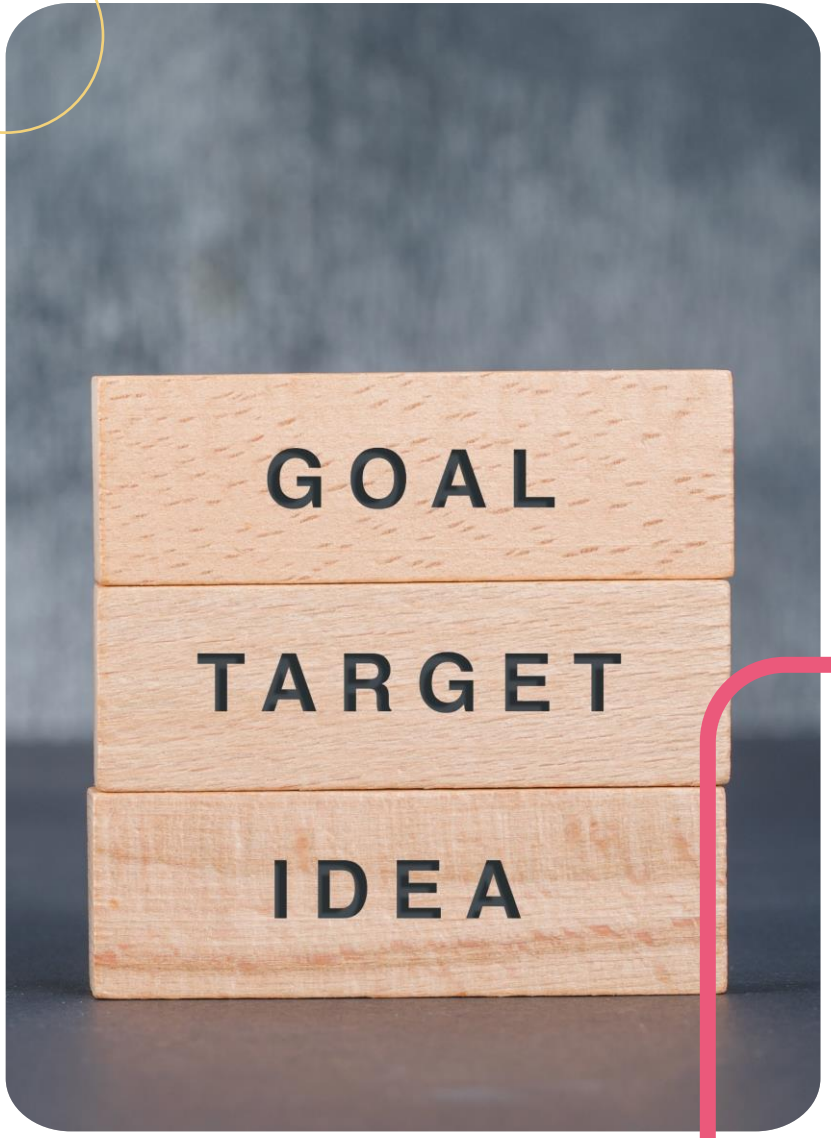
Computer Science

Memory Management

Lesson 7 Challenge Feedback

Which C standard does your compiler conform to?

- Go to HELP
- Click ‘About’ and you will see all the info you need to know
- For example: GCC conforms to the C99 standard



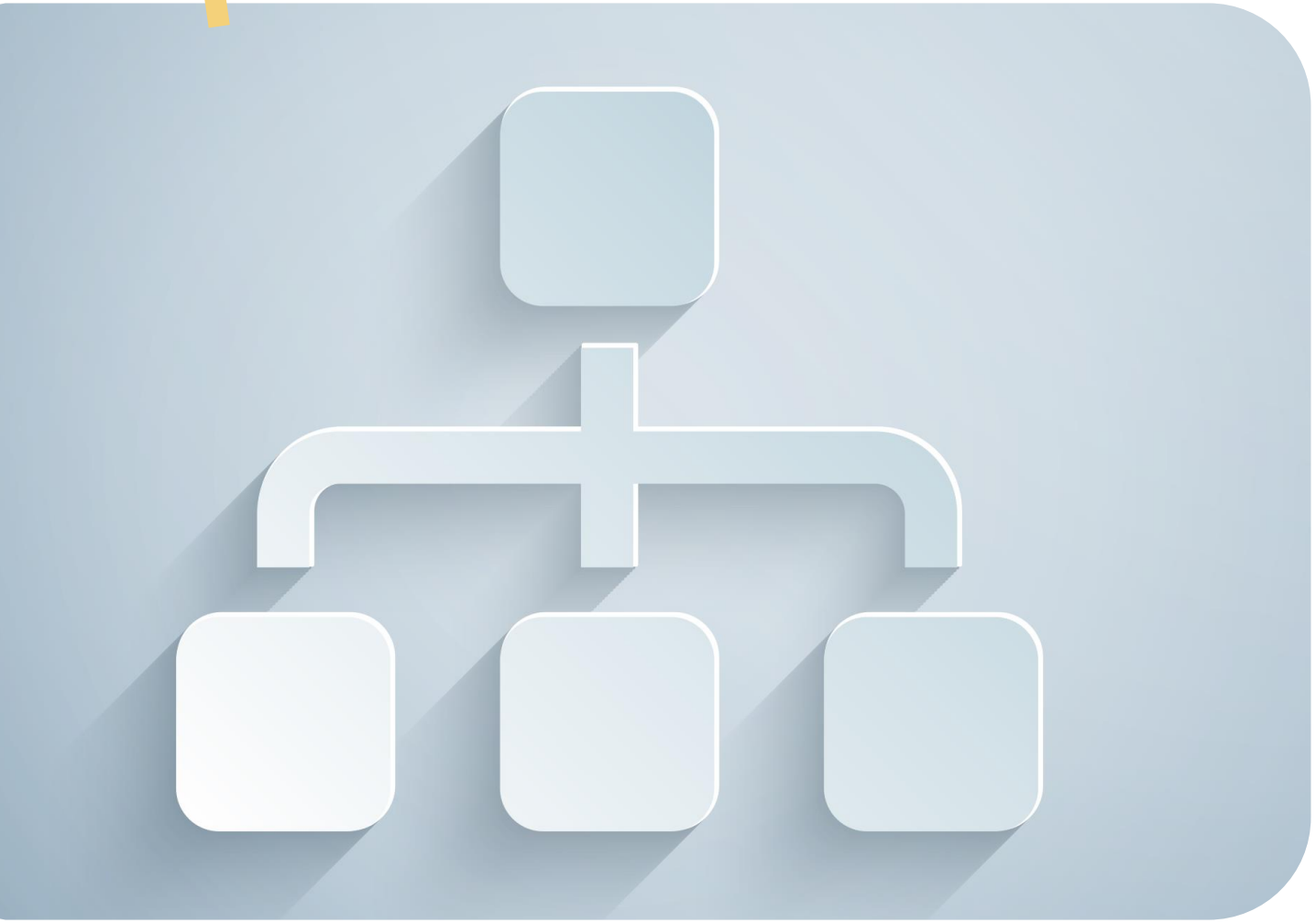
Objectives

- Appreciate the relationship between memory management and pointers
- Become aware of how the operating system handles memory
- Explore certain built-in functions
- Write programs using memory management techniques

Memory

A step back...

Memory



- Comes in different types
- Made up of transistors and capacitors representing 1 or 0 using charge
- Categorised according to size and distance from CPU

Memory hierarchy

Registers – closet to CPU and small but fast

Cache – close to CPU but not as fast as registers and used to store data to be used shortly

RAM – largest and slowest type of working memory

Factors affecting memory

- Speed (latency)
- Size
- Cost
- Position

Memory hierarchy in action

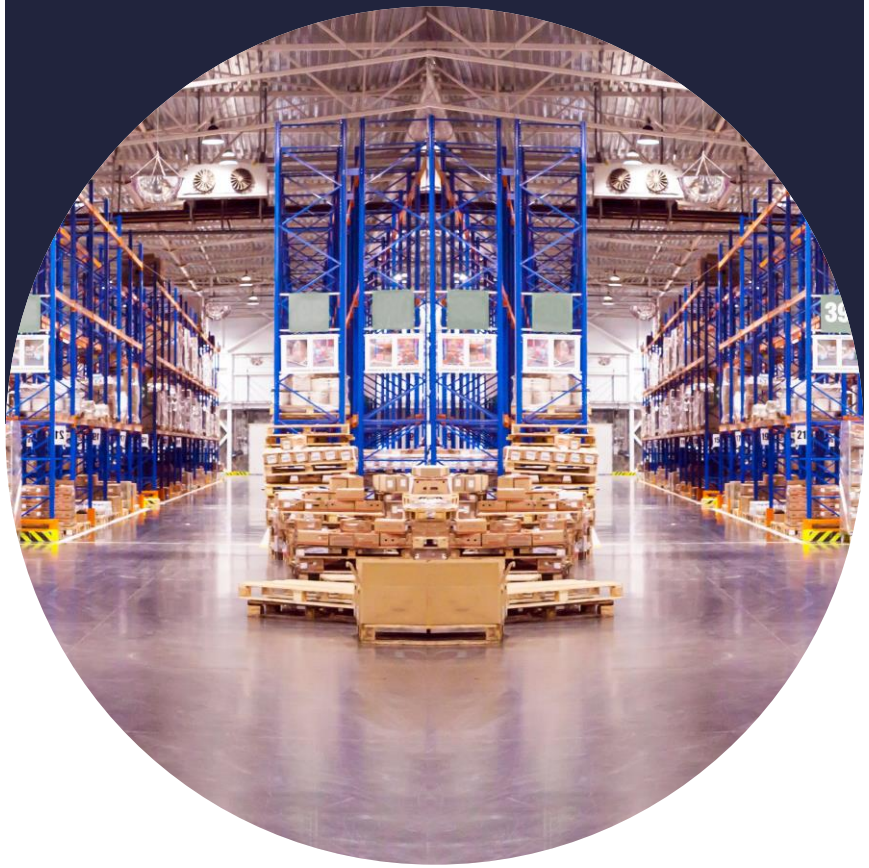
Computer can be told to store variables in registers to make it run faster

Comparable to getting cooking ingredients from the pantry

Operating system memory management

OS memory management

- OS handles movement of data and programs between memory and hard drive
- Memory manager keeps track of all memory locations in main memory (RAM)



OS memory manager is like a warehouse manager!

- Decides which process gets memory
- Monitors de-allocation and reallocation of memory

C program structure

When process launched it gets a place in process address space

↓

Variable names, constants, and instruction labels become memory addresses

↓

Compilation process converts these to a computable address

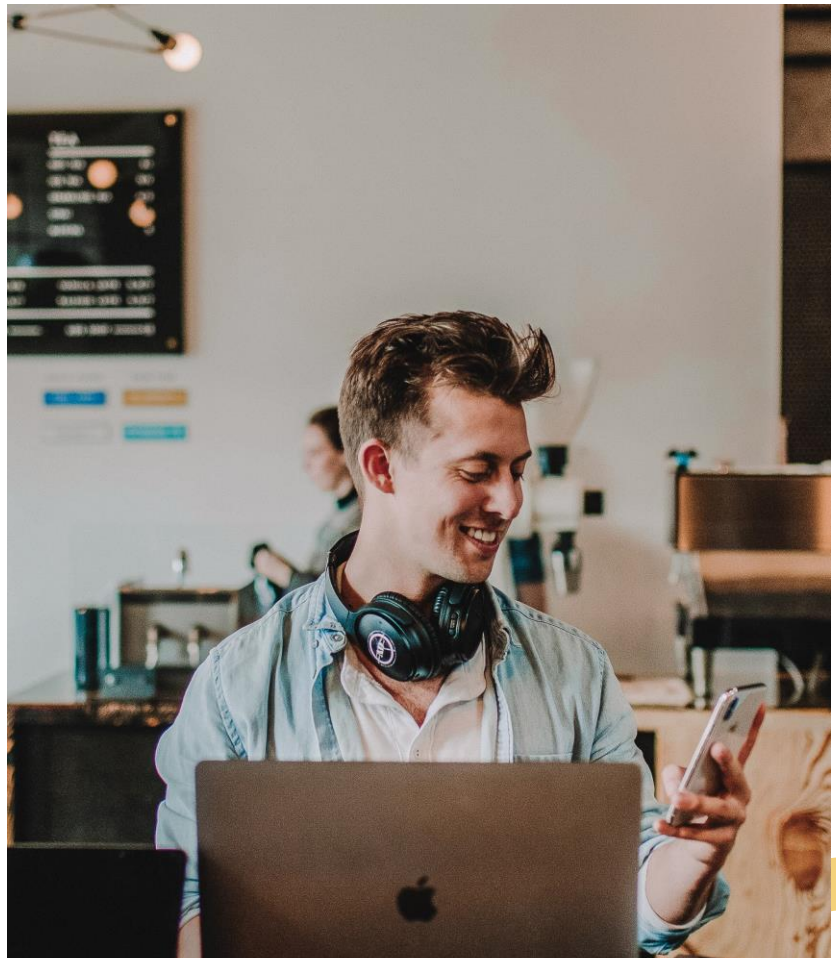
Stack

- Freed when they go out of scope
- Managed automatically
- Compiler needs to know size of variable in advance
- Space in stack is limited

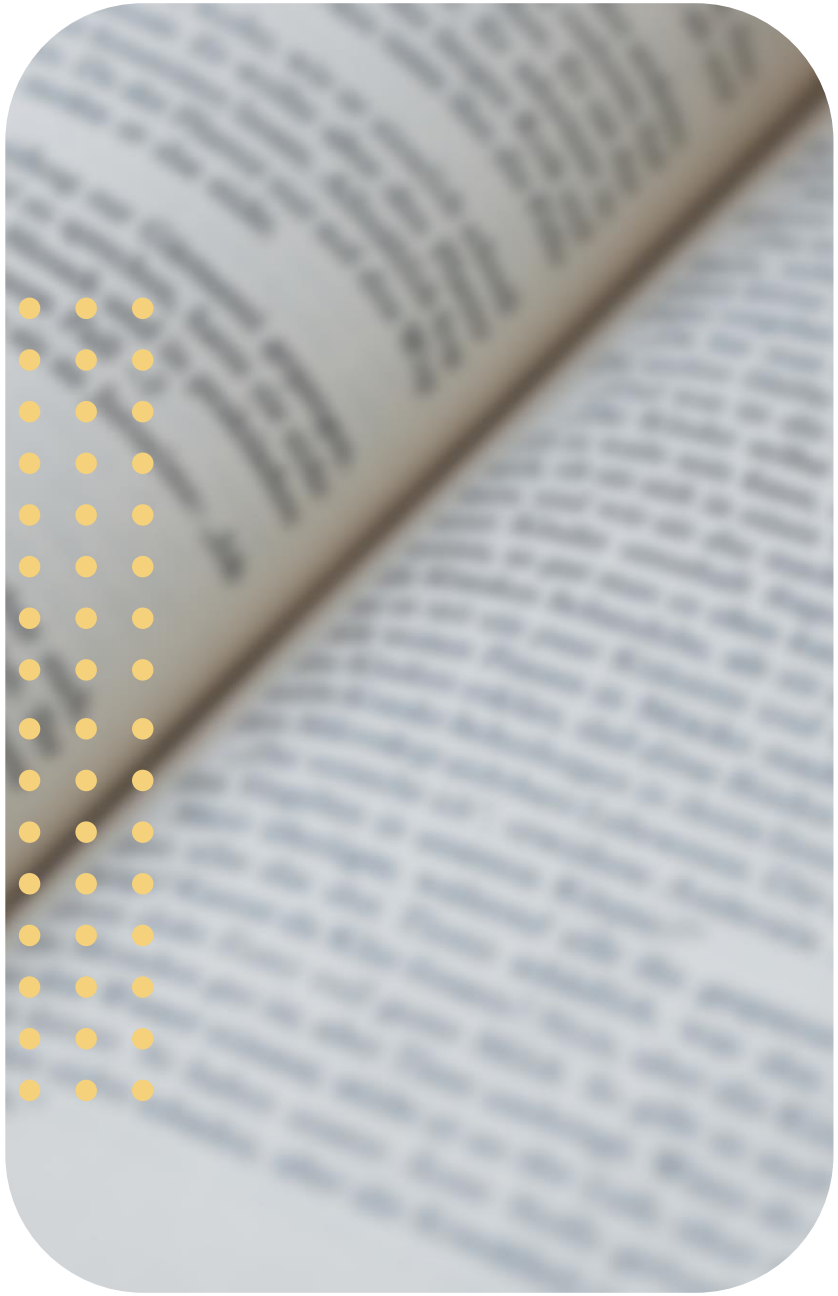
Heap

- Memory to store global variables
- Allocated dynamically
- Must be handled correctly to avoid a memory leak

C program sections



1. Text segment
2. Initialised data segment
3. Uninitialised data segment
4. Stack
5. Heap

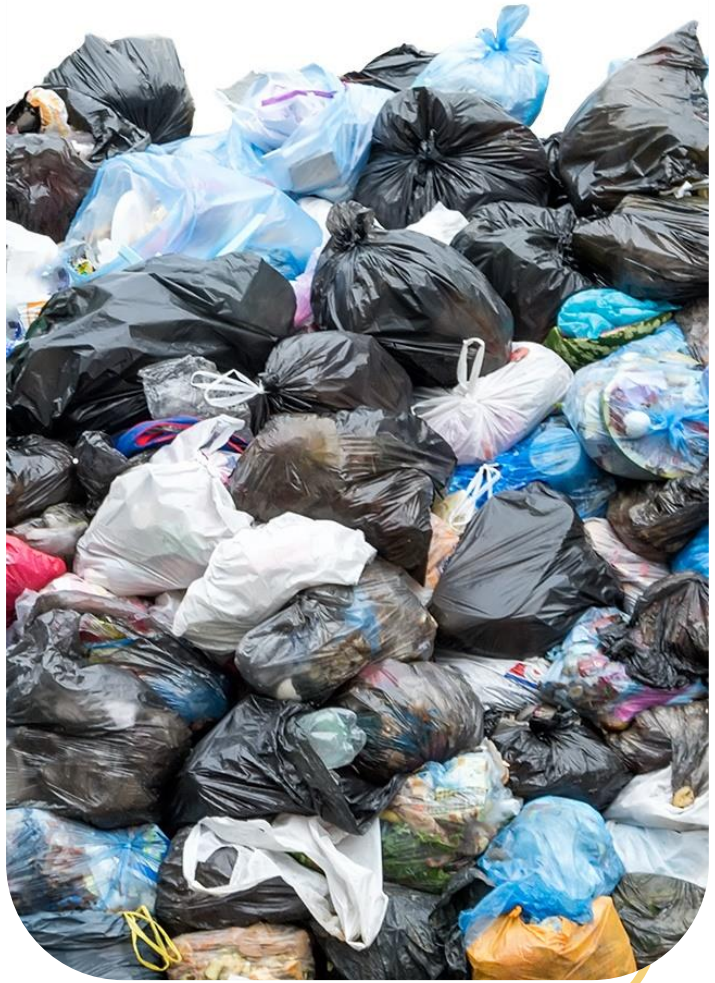


Text segment

- Contains executable code
- Shareable – a single copy kept in memory
- Read-only

Initialised data segment

- Contains initialised global variables and static variables
- Not read-only – variables can be altered at runtime



Uninitialised data segment

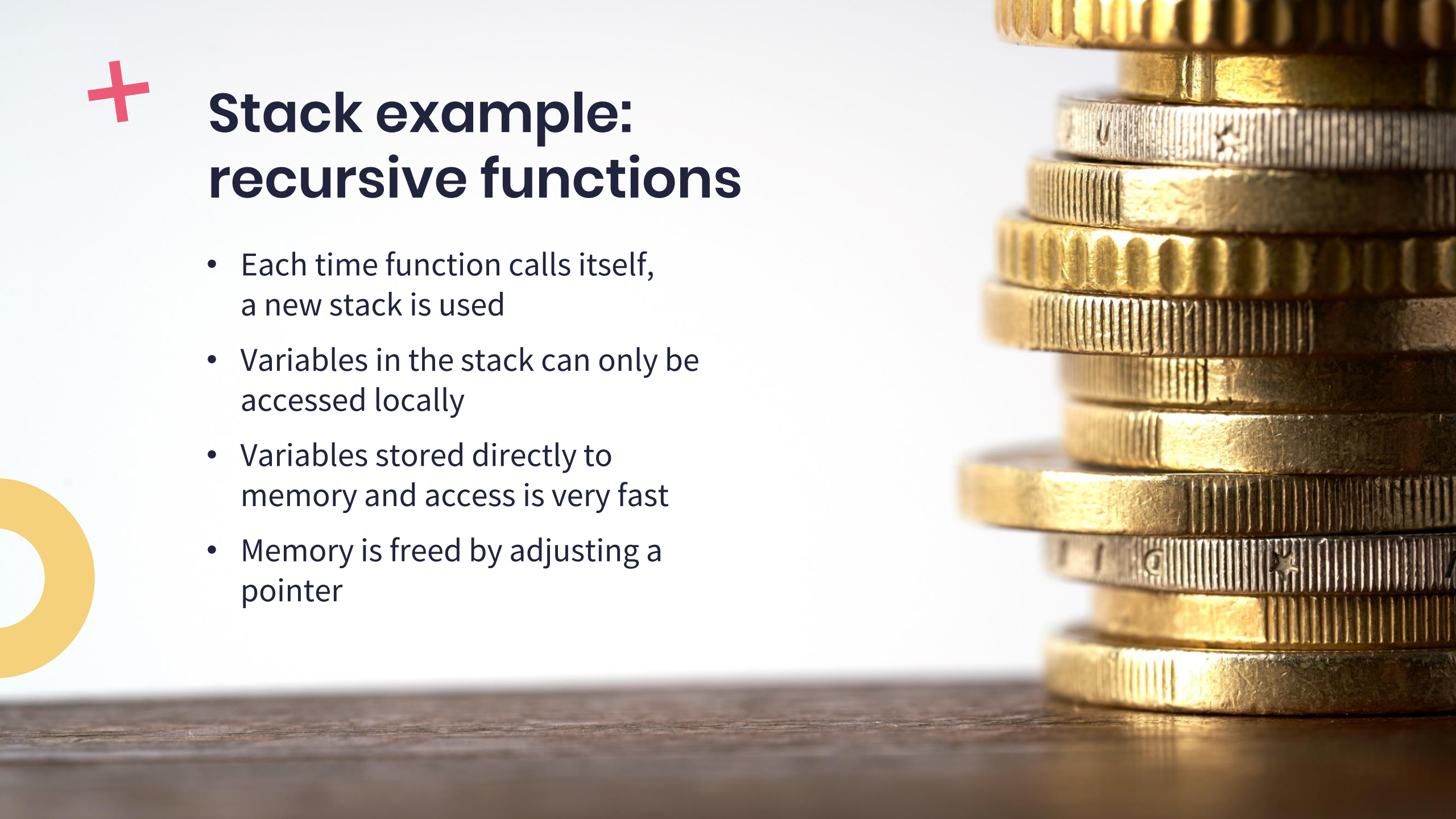
- Contains uninitalised variables
- Starts where initialised data segment ends
- Variables don’t contain any value

Stack

- Automatic variables stored with function data
- LIFO: last element added to structure is first taken out
- Return address saved each time function is called

Stack example: recursive functions

- Each time function calls itself, a new stack is used
- Variables in the stack can only be accessed locally
- Variables stored directly to memory and access is very fast
- Memory is freed by adjusting a pointer



Heap

- Begins where the stack ends
- Managed by the built-in C functions
- Memory space can be contiguous or non-contiguous and allocated dynamically
- Hierarchical structure
- Larger in size and only limited by size of virtual memory

Cleaning up variables in C

- Other programming languages do this automatically, which impacts performance
- C provides tools that you can use to allocate and reallocate memory as required
- Result is efficient, fast programs with a small system footprint

Memory management in C

Dynamic memory allocation

- Programmer can give the program more memory during runtime as needed
- Overcomes the biggest limitation of arrays – size and number of elements

Memory management functions

Stdlib.h

- malloc.h contains the definitions for the memory allocation functions
- Four functions in C for dynamic memory allocation:
  - malloc() – memory allocation
  - calloc() – contiguous memory allocation
  - free() – frees memory
  - realloc() – reallocating memory

malloc()

- Allocates the requested memory and returns a pointer to it

```
void *malloc(size_t size)
```

Syntax

- Takes the size of the memory block in bytes as a parameter
- Thread-safe

calloc()

- Allocates the requested memory and returns a pointer to it
- Sets the allocated memory to 0

```
Void* calloc( size_t num, size_t size );
```

Syntax

calloc()

- Takes size as a parameter and returns a pointer to the memory location
- Returns null if operation is unsuccessful
- Thread-safe

free()

- Deallocates memory previously allocated

```
Syntax: void free(void*ptr)
```

Syntax

- Takes a pointer as a parameter
- Does not return any value
- Attempting to free deallocated memory causes undefined behaviour

realloc()

- Reallocates given area of memory
- Block of memory must have previously been allocated and not freed using the free function

realloc()

Method 1: expanding or contracting the existing area pointed to by a pointer

Method 2: allocating a new memory block of the required new size, copying memory area with size equal the lesser of the new and the old sizes, and freeing the old block

realloc()

```
void *realloc(void *ptr, size_t size)
```

Syntax

Two parameters:

- Pointer to previous memory block
- New size of memory block

Fragmentation...

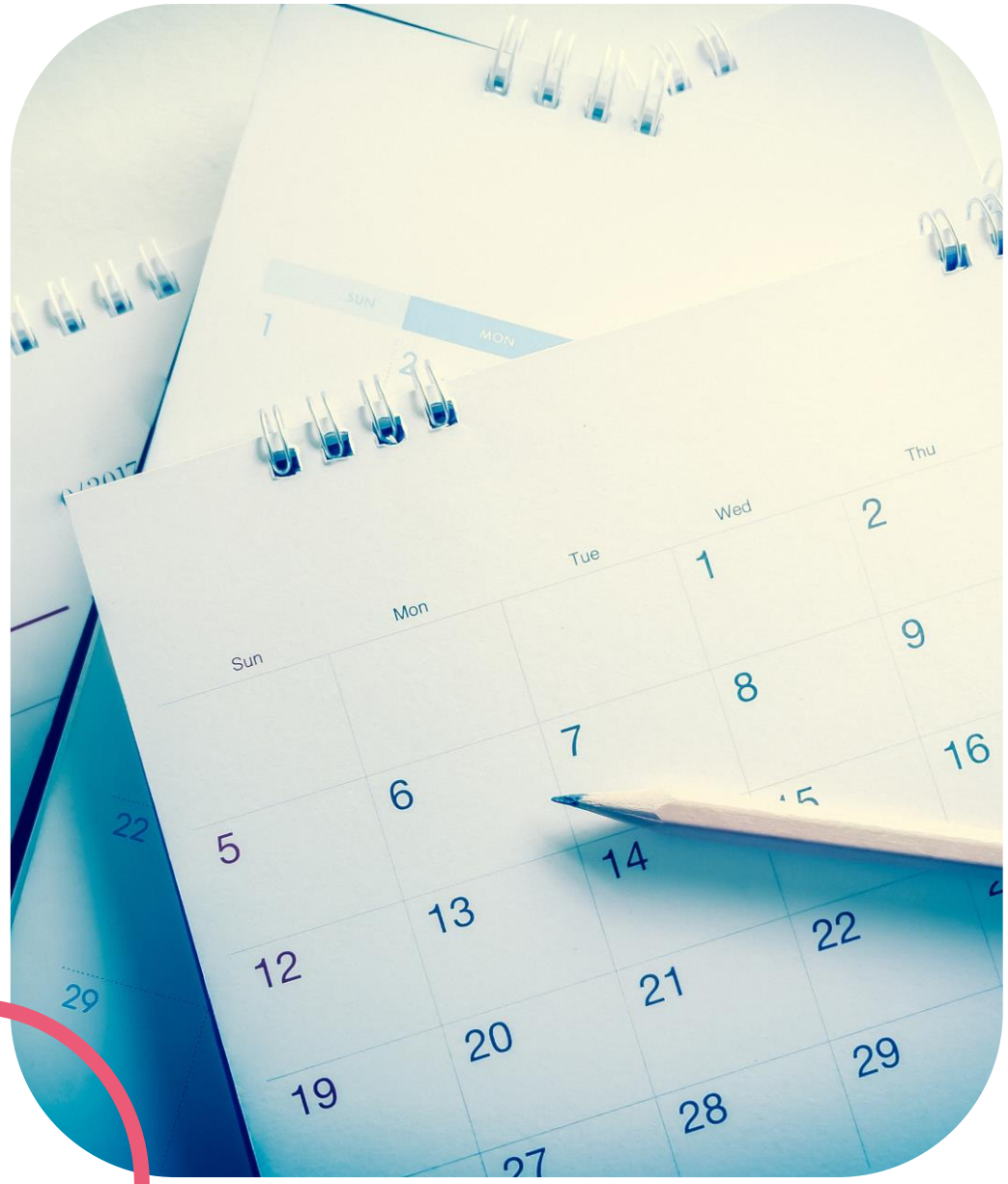
when allocated memory within other allocated memory is deallocated.

- Have to use memory allocation and functions correctly to avoid problems
- C leaves it up to you

Problems with memory management

Dangling pointers

- A pointer that points to nothing
- Will cause computer to crash
- Used in zero-day attacks



Wild pointers

- Caused by not initialising the property on first use
- Points to nothing
- Rather implement an alternative version of the free function

Garbage

- A result of allocating memory and providing no way of reclaiming it when you're no longer using it
- Heap section of the program is managed by the programmer
- Causes problems on constrained systems

Remember...

- Don’t try to free a memory area that was never allocated
- Initialise pointers when you declare them

Did you know?

Google and Facebook employ malloc specialists.