**Lecture 30 – STRUCTS**

Debugging

A good method of debugging is **sticking printf’s** **everywhere** and when running the program, see if the printf’s are also being executed.

Structs

A struct is a **compound data type** which allows the storage of multiple variables under a single name – its another tool for abstracting data. Unlike an array, the variables don’t necessarily have to be of the same type, but can be of different types. In other programming languages, its often called a **class**.

We initialise a struct in a similar way to an array – setting all the values with squiggly brackets.

Typedef

Typedefs are used to **create an alias** **(alternative)** name for a data type. The main role of typedefs are to simplify syntax within a program. It also makes it easier to modify code. To declare a typedef, follow this structure:



Difference between typedef and #define: typedef is **executed by the compiler**, whereas #define is processed by the **pre-processor** which just inserts whatever is in #define into the code.

Extra Functions

You can google any function or header file to get a description of what a specific function does. However, we shouldn’t look at these functions as magic, and we should only never use a library function if we can write that code ourselves.

* **math.h** – Has a lot of mathematical functions
* **limits.h** – Contains all the limits of c of every variable
* **string.h** – *strtold*( ); will turn a string to a floating point number

*atoi*( ); turns a string into an integer

* **stdio.h** – *fprintf*( ); will print data to a file

*snprintf*( ); will print something to a string. The s stands for string and n stands for length of that string.

**Lecture 31 – RISK**

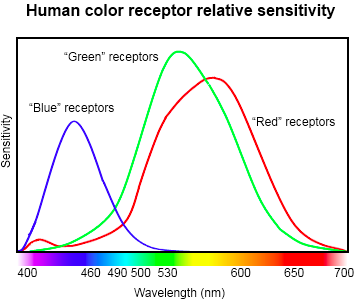
Before writing a program, we should think about the key things we must get right, in order to avoid any potentially catastrophic failures / vulnerabilities. Humans are notoriously bad at evaluating risks – we spend large amounts of time on the “weakest link”, but may forget about some other equally critical problems. E.g. Putting a lot of money into the front door of the house, but not the back door.

Risk assessments in the short-term are often fairly accurate, but we are especially bad in the long term.

* Security is all about **investing sensibly in defending stuff** – not just heaping all cash covering up one weakness, because attackers then have plenty of other weaknesses to attack.
* Risk management is considering risks and how to sensibly deal with them, taking into account the **severity and probability of the risk** and **how hard it is to prevent it**.

**Lecture 32 – COLOR**

Your eyes work through 3 different cones, **(1) small (2) medium (3) large**, which each respond to different wavelengths of light at different levels. Colours you see aren’t just single wavelengths, but a bundle of different wavelengths triggering each cone to a different degree, and how much of each type of cone is triggered will determine the colour you perceive.

* Computers use this fact so they can have three coloured lights per pixel, instead of 16 million and still show 16 million colours with just the three lights.
* The three types of cones roughly map to RED, GREEN, BLUE light so those are our primary colours.
* Many women can see an extra dimension of colour, because they have two XX chromosomes, while males have one XY.

**Lecture 34 – Stacks and Stack Frames**

Frames

Frames have a **STACK-LIKE** structure and are **FILO** – First in, last out.

Main program needs to call function 🡪 execution stops there (+stores details of progress in main program’s frame) 🡪 flow of control jumps to function (+jumps to function’s frame +stores details such as inputs of the function / return address)

Frames and the main program:

* Main needs local space, in it’s own frame, to store its variables.
* Main also needs to tell some of its variables to the next function (i.e. arguments passing into a function)

What is stored in a frame?

* A return address (address to jump back to after executing f’n)
  + When main program calls function in line 5300, return address is 5304 (assuming instructions are 4 bytes long) and 5304 is written in the function frame.
* Arguments that are passed in (information that the calling function has given it)
* The local variables

Disadvantages of knowing the address of frames:

* If two functions are used in the same frame, they will overwrite each other.
* [**Recursion**](http://www.cs.utah.edu/~germain/PPS/Topics/recursion.html) may occur: if a function calls itself, it cannot freeze and call itself.
  + A function that keeps calling itself will keep adding onto the stack, which may cause a potential overflow.
* If you know where things are stored, you can mess with them.
* It is wasteful of space (If there are 500 functions, you have to set up 500 frames which could be really big) when a comparatively small number of functions are called at any time.

Stacks

Frames are not stored in a fixed location in memory but in a stack, which will be called as stack frames.

Supposed that there is a stack pointer (which tells the program where to put the next frame i.e. where free area is) that points to the first cell of the stack.

* Main starts running and it needs space to store its local variables, so C or GCC builds and allocates a frame from the **bottom of the stack** and then **changes the stack pointer to the point above it.**
* E.g. When a frame is needed for a function, it sets aside a new area of memory and adjusts the pointer to the point above this new frame.
* When the function calls itself, another frame will be set up for it:
  + When the function returns, the pointer moves down. Although there is memory still there, the next time a new frame is needed, the memory will be overwritten.  
    **= memory is automatically reused when a function returns.**

Other things about stacks

Frames are made during runtime and doesn’t take much computation time

You can look inside the memory above in a stack.

Computers work in segments (areas of memory) and puts a stack at the bottom of memory and it goes upwards.

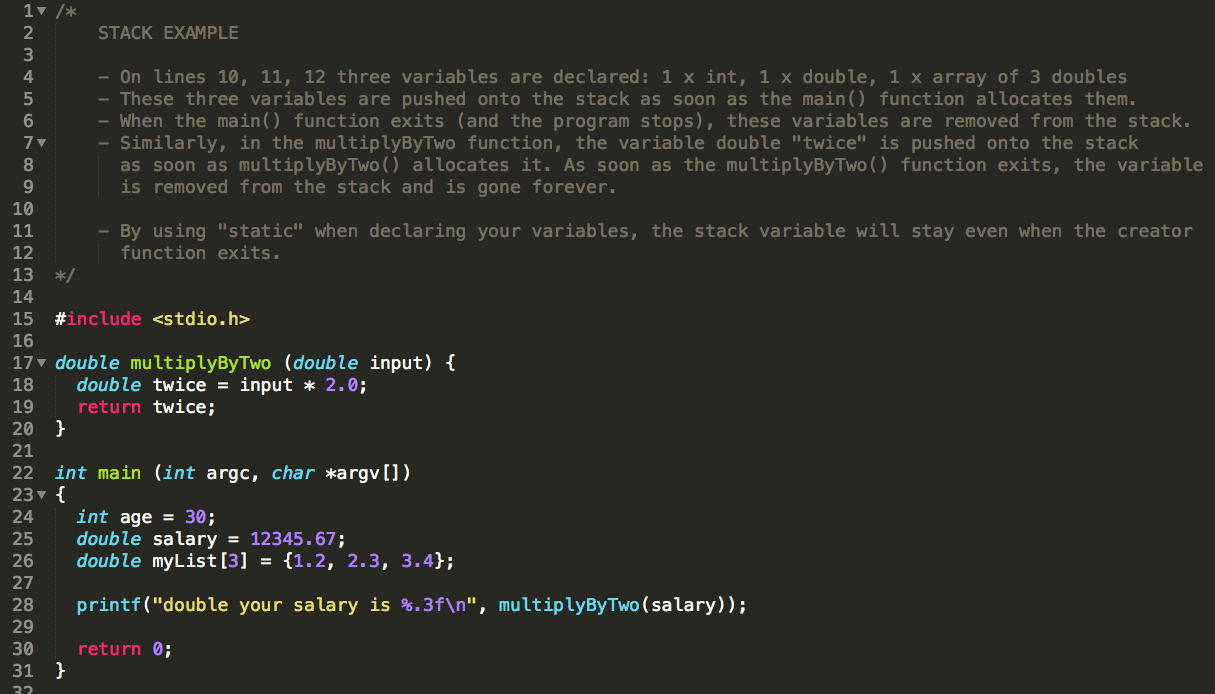
* It has another segment of memory above the stack, called a HEAP which goes downwards.
* A HEAP is a region on your computer’s memory that is not managed automatically for you. It is free-floating.
  + To allocate memory on the heap, you must use **malloc ()** which is an in-built C function.
  + Once you have allocated memory on the HEAP, you are responsible for using **free ()** to deallocate that memory once you don’t need it anymore.
  + If you fail to do this, your program will have a **MEMORY LEAK**: Reduces available memory, reducing computing performance. In worst case scenarios, the system or devices stops working at all.

When to use the STACK vs HEAP?

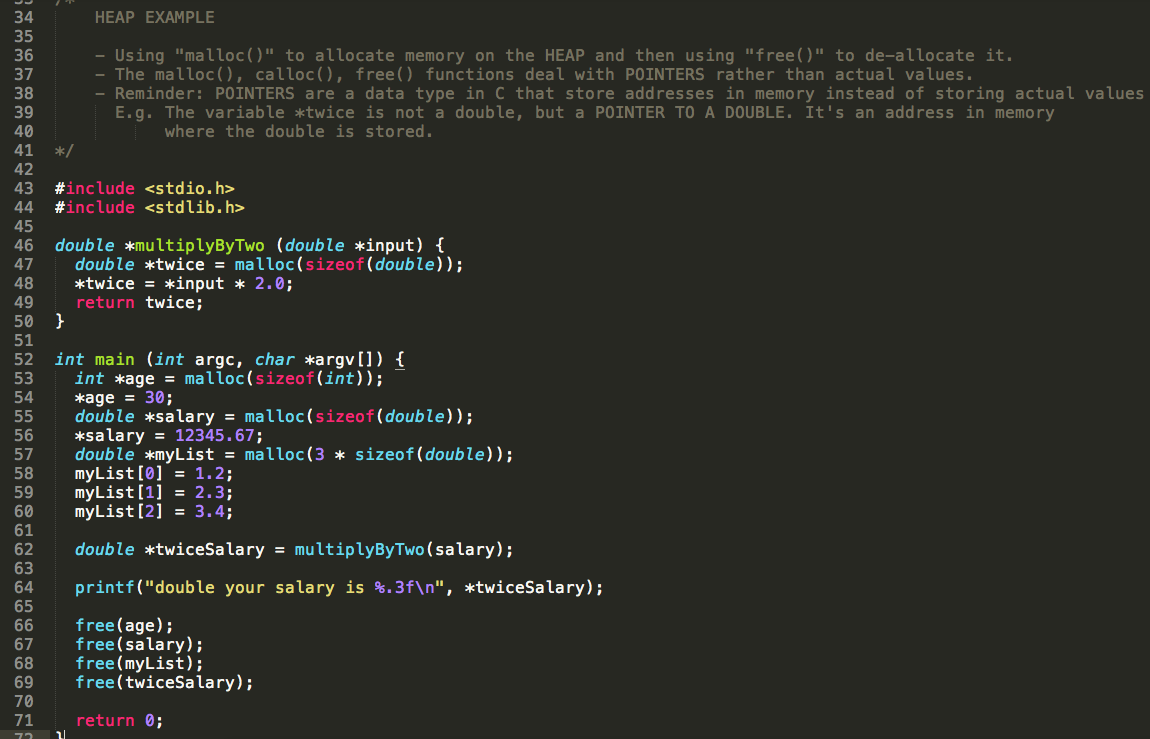
* If you need to allocate a large memory block (e.g. a large array, or big struct) and you need to keep that variable around for a long time (like a global variable), then you should allocate it on the HEAP.
* If you are dealing with relatively small variables that only need to persist as long as the f’n is alive, then use the STACK.

|  |  |
| --- | --- |
| **STACK PROS & CONS** | **HEAP PROS & CONS** |
| * Very fast access * Don’t have to explicitly de-allocate variables * Space is managed efficiently by CPU, memory will not become fragmented * Local variables only * Limit on stack size (OS-dependent) * Variables cannot be resized | * Variables can be accessed globally * No limited on memory size * Relatively slower access * No guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed. * You must manage memory (you’re In charge of allocating and freeing variables) * Variables can be resized using **realloc()** |

STACK EXAMPLE



HEAP EXAMPLE



**Lecture 35 – malloc()**

Revision

* Frames are stored in STACKS while the rest of memory is in the HEAP
* Stacks load from the bottom, while the heap stores from the top
* Once a function ends, the stack pointer goes to the next frame, memory is freed up
* **STATIC MEMORY** – allocated at compile time, doesn’t change

**DYNAMIC MEMORY** – allocated at run-time, this is used when an unknown amount of memory is necessary i.e. because of user output

Malloc ()

* **Malloc (size);** returns a pointer to some free memory in the heap that matches the size (in bytes) you give it.
* Unlike functions, which free the memory they were assigned in the stack on exit, *malloc()* assigns memory **indefinitely**
* Memory that is “malloc-ed” must manually removed

Free (pointer)

* De-allocates the memory that malloc allocated, at that pointer.
* Takes a **pointer to the first addres**s in that memory.
* After you “free” the memory, make sure not to keep using it, as other functions may use that memory.
  + A way to ensure freed memory is not used, is to set the freed pointer to NULL immediately afterwards.

The HEAP manager

The heap manager keeps track of how much memory each function requests, so *free()* knows exactly how much memory to free up and will tell the heap manager what area of the memory is free for use again.

The operating system at the end “should” free everything up.

* The program preserves the memory in *malloc()* while its running in the world that the operating system gives it.
* The program won’t hand its memory back to anyone, when you free memory, as it is in its own bubble inside the operating system. When the program ends, it’s the operating system’s job to clean up that area of memory.
* Memory allocated to the *malloc* doesn’t cause the stack pointer to move back down the stack when a function returns

Notes on compiling

Some warnings and errors don’t appear when compiling unless you add extra flags

* **-Wall**: will display ALL warnings
* **-Werror**: will make warnings appear as ERRORS and stop the program from compiling
* The C compiler just thinks you know what you are doing, without flags.

You need to compile in extra checks such that errors will show up if there is, say, a **buffer overflow** at run-time.

* **LINT** is a program that will look through your code in more detail to check if there are any errors in the code.

Mistakes

1. Memory Leaks

* Keep allocating memory without freeing it

🡪 Run out of memory and your computer crashes.

1. Overflowing

* Accessing wrong memory (like a buffer overflow)

1. Freeing memory allocated by a different function { free(buffer); }
2. Trying to allocate more than is available
3. Assigning an incorrect amount of memory

* Use the **sizeof()** function when assigning memory to reduce the chance that you miscalculate the amount of memory each type of data needs.

**RICHARDS RULE OF THUMB: “Never use *malloc()*” unless you need to.**

Pointers and Malloc()

If *pointer buffer* points to some memory, and you call *free(buffer)*, what happens to buffer?

* It tells the memory allocator that the area can be **reused and overwritten**
* Buffer still **points to the same address**
* Buffer’s **content does not get reset**
* Good practice is to **set the pointer to NULL to avoid accidentally** using it
* NULL – a reserved world in the *stdlib.h* library. It is a pointer that’s guaranteed to point to nowhere.