THE C PROGRAMMING LANGUAGE

An intro to the basics of C

WHO'S THIS PRESENTATION IS FOR

- First years without previous programming experience
- Individuals interested in learning C from a beginner's perspective

WHAT WILL WE BE GOING OVER

- Setting up an environment to build C programs for x86 machines
- Writing a 'Hello world!' Program
- Syntax structure of C programs
- The C compilation pipeline

SETTING UP A DEVELOPMENT ENVIRONMENT

```
sudo apt update && sudo apt upgrade –y
sudo apt-get install build-essential gdb
```

These commands will install necessary updates and build tools we will use for upcoming content in the lecture

- For this lecture we will be writing programs in an Ubuntu WSL instance.
- To install WSL and Ubuntu follow this link https://learn.microsoft.com/en-us/windows/wsl/install
- Once you've installed Ubuntu type the following commands into the Ubuntu terminal

SETTING UP A DEVELOPMENT ENVIRONMENT

- In the Ubuntu terminal you will need to make a directory for where your written code will be saved.
- Commands such as ls and cd will allow you to view the content of the currently accessed directory and change directories, respectively.
- For this lecture run the following command:

```
mkdir -p ~/umsae/C_tutorial

cd ~/umsae/C_tutorial

code .
```

Once these command have ran you should see a Visual Studio Code window open up with the C_tutorial directory open

PART 1: HELLO WORLD

All programs start somewhere

PART 1

- In a vscode window make a new file called 'hello.c' inside this newly made file put the following code in.

```c
#include <stdio.h>

int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

PART 1

- Inorder to compile and debug in vscode you will need to make two files
- Launch.json
- Open the debug tab on the right-side bar and click the link labeled 'create a launch.json file'
- Task.json
- Press the keyboard short cut 'CTRL+SHIFT+P' to open the search menu
- Search for 'Configure Tasks'

PART 1

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "type": "shell",
            "label": "gcc build active file",
            "command": "/usr/bin/gcc",
            "args": [
                "-g",
                "${file}",
                "-o",
                "${fileDirname}/${fileBasenameNoExtension}"
            ],
            "options": {
                "cwd": "/usr/bin"
```

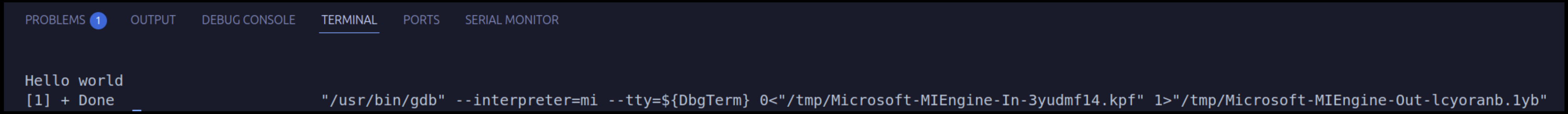
Tasks.json

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "gcc - Build and debug active file",
            "type": "cppdbg",
            "request": "launch",
            "program": "${fileDirname}/${fileBasenameNoExtension}",
            "args": [],
            "stopAtEntry": false,
            "cwd": "${fileDirname}",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "setupCommands": [
                {
                    "description": "Enable pretty-printing for gdb",
                    "text": "-enable-pretty-printing",
```

Launch.json

PART 1

- Now that everything is installed you should be able to click the run button and debug the hello world program.
- The program should output something like this in the terminal



```
Hello world
[1] + Done _                    "/usr/bin/gdb" --interpreter=mi --tty=${DbgTerm} 0<"/tmp/Microsoft-MIEngine-In-3yudmf14.kpf" 1>"/tmp/Microsoft-MIEngine-Out-lcyoranb.1yb"
```

PART 2 SYNTAX STRUCTURE

What do you mean I need to use semicolons!

PART 2

- C is a statically typed language.
- Data types of variables must be expressly stated.
- Failure to give C the proper data type to a variable will often result in the compiler refusing to compile your program
- So, what type of data can C use?

PART 2: PRIMITIVE DATA TYPES

Data Type	Int	Float	Double	Char
Description	Represents whole numbers	Single-precision floating point numbers	Double-precision floating point numbers	Single characters or integers

PART 2: DECLARING VARIABLES

Assigning variables is intuitive. First you declare what data type your variable will be then your variables name.

<Data type> <Variable Name>

If your variable should be initialized with a value, you can use the equals sign to assign the value.

<Data type> <Variable Name> = <Value>

PART 2: DECLARING VARIABLES EXAMPLES

```
void foo( void )
{
    int a = 1;
    float b = 1.0f;
    double c = 2.0;
    char character = 'a';
}
```

PART 2: STRUCTS

- In some cases, you will have data that is made up of different data types but relates to one structure.
- Here we make use of a programming paradigm called structures.
- These structures work similarly to objects in languages like Java or C#
- Members inside of structures can be comprised of primitive data types or other user defined structs

PART 2: STRUCTS EXAMPLES

```
struct Person {
        char name[50];
        int age;
        float height;
        struct Person *friend;
};
int main(void)
{
        // Declare and initialise a struct
        struct Person student = {"Alice",20,5.7f};

        // Accessing to the stucts members
        printf("Name %s\r\n", student.name);
        printf("Age: %d\r\n", student.age);
        printf("Height: %.1f\r\n", student.height);
```

PART 2: ENUMS

- Other times we want to use variables to represent states or key-value pairs.
- In this case we use enumerated values, or enum's for short

PART 2: ENUM EXAMPLES

```c
#include <stdio.h>

// More realistic logging example
enum LoggingLevels {
    DEBUG = 0,
    INFO,
    WARN,
    ERROR,
    ALWAYS
};

void log_message(enum LoggingLevels level, const char* message) {
    static enum LoggingLevels current_level = INFO;

    if (level >= current_level) {
        const char* level_names[] = {"DEBUG", "INFO", "WARN", "ERROR", "ALWAYS"};
        printf("[%s] %s\n", level_names[level], message);
    }
```

PART 2: TYPEDEFS

- Using enums and structs we can make our own datatypes using typedef's

PART 2: TYPEDEF EXAMPLE

```c
typedef enum {

    VEHICLE_CAR,

    VEHICLE_TRUCK,

    VEHICLE_MOTORCYCLE

} VehicleType;


typedef enum {

    STATUS_OFF,

    STATUS_IDLE,

    STATUS_MOVING,

    STATUS_ERROR

} VehicleStatus;


int main(void)

{

    Vehicle my_car = {

        VEHICLE_CAR
```

PART 2:FUNCTIONS

- All function are comprised of two components. Parameters and return values.
- Parameters are variables that a function takes in and uses during it runtime scope.
- Return values are values that the function will send back after being ran.

PART 2: FUNCTION EXAMPLE

```c
// Function with multiple parameters

int add(int a, int b)

{

        return a + b;

}


// Function with no parameters - use void

void print_hello(void)

{

        printf("Hello!\n");

}


// Function with array parameter (requires size)

void print_array(int arr[], int size)
```

PART 2: VARIABLE SCOPE

Variable Scope

- Scope Determines where a variable can be accessed
- Local variables: Declared inside a function, only accessible within that function
- Global variables: Declared outside functions, accessible anywhere

C Functions are Pass by Value

- When passing a variable to a function, a copy is made.
- The original variable remains unchanged
- Changes inside the function do not affect the origninal

PART 2: PASS BY VALUE EXAMPLE

```
void modify_value(int x) {
    x = 100;  // Only changes the copy
    printf("Inside function: %d\n", x);
}

int main(void) {
    int number = 5;
    modify_value(number);
    printf("In main: %d\n", number);
    return 0;
}
```

PART 2: BUT WHAT ABOUT *STATIC*?

Static has different functionality depending where it's being used.

- Inside functions: Preserve value between calls
- Outside functions: Limit scope to current file

STATIC LOCAL VARIABLES

These two snippets of code have different outputs. Do you know what the output will be when they run?

```
void counter_normal(void) {
    int count = 0;
    count++;
    printf("Normal: %d\n", count);
}

void counter_static(void) {
    static int count = 0;  // Preserves value
    count++;
    printf("Static: %d\n", count);
}
```

STATIC GLOBAL VARIABLES AND FUNCTIONS

- Benefits of limiting scope
- Abstract data structures and functionality
- Reduce naming conflicts

PART 2:CONTROL FLOW

Too be or not to be

CONTROL FLOW

- C has 2 control flow conditonals
- If else
- switch

IF VS SWITCH

IF ELSE

- Compare values using boolean expressions
- Act differently depending on the boolean result (else)
- Move on with the rest of the function

SWITCH

- Limited domain of expected values
- Commonly used with enums
- Often calls functions depending on what case was called.

PART 2: LOOPS

- You can do 3 different loops in c
- For
- While
- Do-while

ARRAYS

```
float pool[1000];
```

Looks like the pool has a lot of floaties in it.

POINTERS

```
char *string = "Quebecois";
```

Pointer? I barely know her!