AURA PRESENTS

ROBOTICS WORKSHOP

PART 1 OF 3 — CRASH COURSE TO PROGRAMMING IN C — ROOM: 405-328

1 HELLO WORLD

Welcome to the world of programming. We will ambitiously try to complete half of the ENGGEN 131 course within three hours. Enjoy!

Note

Since we are speeding things up, these tasks may be challenging to you. Whenever you're stuck, or you feel like you're falling behind, feel free to ask for help!

To begin, hop onto:
https://repl.it/repls/ApprehensiveWellmadeLifecycle

You'll find code already available.

Hit the "run" button at the top. This does the following:

1. Your code gets compiled to a runnable file.
2. The generated file is then executed, and "Hello World" is displayed on the screen.

EXPLANATION

First, look at lines 15 and 24:

```
int main(void) {
  ... lines skipped ...
}
```

These two lines are special. They tell the compiler where the program starts and end. You'll learn more about it in the Functions section.

Programs perform instructions one at a time, typically going down line by line. Each line is called a ***statement***. Statements are ended with a semicolon (the "`;`").

The first instruction that gets executed is line 19:

```
printf("Hello World\n");
```

This `printf` is a function, a.k.a. a command, that instructs the computer to display a piece of text (you'll learn more about functions soon). In this case, this code instructs the computer to display "Hello World" onto the screen.

TASKS

Try changing the "Hello World" text. What happens when you hit "run"?

What happens when you add text after the strange "`\n`" text? What do you think this strange text does?

UNDOING YOUR CHANGES

If you've made some changes to the code we've supplied and you want to revert to our code, simply hop back onto the original link we've supplied in this handout. Your changes will then be erased.

COMMENTS

You can add comments in your code to help explain what a line does, or to make a note. Comments has no effect on the program behaviour – they're purely for us humans to read. In C, you can comment each line individually or comment blocks of code at once:

```
// single line comment

/*
block comment that can
span
several
lines
*/
```

2 VARIABLES & DATA TYPES – STORING DATA

DATA TYPES

Programs tell the computer how to manipulate ***data***. The "Hello World" text from before is an example of such data. There are several kinds of data, known as ***data types***. The data types we will be using are integers, floats, and chars.

Data Types – Integers

Integers are whole numbers. Writing numbers without a decimal point, such as `42` and `-36`, are assumed to be integers.

Data Types – Floats

Floats are numbers that can have fractional parts. Writing numbers with a decimal point, such as `1.0` and `-0.35`, are assumed to be floats.

Data Types – Chars

Chars are single characters. These are written by wrapping the character in single quotes, such as `'c'` and `'\n'`.

What about the text "Hello World"? It's not a single character, but a sequence of characters, and you'll learn about that soon when we get to the Arrays topic.

VARIABLES

It'll be useful to be able to store data while the program runs. To do this, we can put data into ***variables***. There are three things we'll be doing to variables: creating them in the first place (called ***declaring variables***), putting data into variables (called ***assigning variables***), and reading data from variables.

Declaring Variables

Variables are declared by writing

```
<variable type> <variable name>;
```

where `<variable type>` can be, for example, `int`, `float`, or `char`.

For example:

```
// Here we declare an integer called x,
// and a float called myFloat.
int x;
float myFloat;
```

Assigning Variables

Variables can be assigned by using the assignment operator "`=`". The value on the right of the equals sign is then written into the variable on the left of the equals sign.

Here's an example:

```
// Here we assign the integer 2 to the
// variable x, and the value 3.0 to the
// variable myFloat.
x = 2;
myFloat = 3.0;

// You can also assign and declare at the
// same time:
float z = -2.8;
```

Using Variables

Just as *numbers* can be put to the right side of the equals sign, we can also use *variables* on the right side. Things on the right side are called ***expressions***. Expressions are things that can have its value calculated in the program. Data and variables can be used as expressions.

```
// Here we assign the value of z to myFloat.
myFloat = z;
// so myFloat now has the value of -2.8
```

We can also combine numbers and variables together to form larger expressions using *operators*. Standard mathematical operators are available (`+`, `-`, `*`, `/`, and many more).

```
// Here we declare a variable
// called sum, and we initialize it
// by using the previous variables
// and adding them together.
float sum = x + z + myFloat;

// Here we display the sum onto the
// screen using the printf
// function. The weird %f specifier
// tells printf where to display
// the float.
printf("The sum is: %f\n", sum);
```

DEMO – USING VARIABLES

Clear the existing code and then run the following.

```
#include <stdio.h>

int main(void) {
  int x;
  float myFloat;

  x = 2;
  myFloat = 3.0;
  float z = -2.8;

  float sum = x + z + myFloat;
  printf("The sum is: %f\n", sum);
}
```

You should then see “The sum is: 2.200000”, because

$$2 + 3 - 2.8 = 2.2$$

TASK – CREATING VARIABLES

Create a program that declares an integer, a float, and a char, assigns values to them (to your liking), then displays them onto the screen.

Tips: Printing Data to the Screen

Calling the following
`printf("My numbers are %d and %d\n", 2, 3);`
will display “My numbers are 2 and 3”, where the `2` and `3` are substituted into the `%d` specifiers.

To display integers, use the `%d` specifier.
To display chars, use the `%c` specifier.
To display floats, use the `%f` specifier.

TASK – COURSE MARK CALCULATOR

Imagine you have a university course with three assessments: a project worth 30%, an assignment worth 20%, and an exam worth 50%. The course mark is computed as the weighted average of the assessment marks. Each assessment is assigned a mark out of a hundred.

Write a program that declares a variable for each assessment mark, assigns values for them, then displays the overall course mark based on this information.

3 CONDITIONALS – DECISION MAKING

RELATIONAL OPERATORS

In addition to the basic mathematical operators (`+`, `-`, `*`, `/`), there are also *relational operators* that help you compare data. These, when given two numbers, will evaluate to either `0` or `1`. They ask questions, such as whether one value is greater than another value, from which it evaluates to `0` when the answer is false, and `1` when the answer is true.

Try the following code with different values of `a` and `b`. It illustrates the behaviour of the following operators: `<`, `>`, `==`, `!=`, `>=`, and `<=`. See if you can figure what each of those operators do.

```
#include <stdio.h>

int main(void) {
  // Here we create two integers
  // called a and b.
  int a = 1;
  int b = 2;

  // We will then compare them using
  // operators.
  int c = a < b;
  printf("The value of c is %d\n", c);
  c = a > b;
  printf("The value of c is %d\n", c);
  c = a == b;
  printf("The value of c is %d\n", c);
  c = a != b;
  printf("The value of c is %d\n", c);
  c = a >= b;
  printf("The value of c is %d\n", c);
  c = a <= b;
  printf("The value of c is %d\n", c);
}
```

LOGICAL OPERATORS

There are also three ***logical operators*** that combine different results together. If `x` and `y` were variables:

- Logical negation: `!x` evaluates to `0` when `x` is nonzero, and evaluates to `1` when `x` is zero.
- Logical AND: `x && y` evaluates to `1` when both `x` and `y` are non-zero, and evaluates to `0` otherwise.
- Local OR: `x || y` evaluates to `1` when either `x` or `y` or both are non-zero, and evaluates to `0` otherwise.

For example, we can compare three variables this way:

```
#include <stdio.h>

int main(void) {
  // Here we create three integers
  // called a, b and c.
  int a = 1;
  int b = 2;
  int c = 3;

  // Is b between a and c?
  int d = a < b && b < c;
  printf("The value of d is: %d\n",d);

  // Do one or more of a and b have a
  // value less than c?
  d = a < c || b < c;
  printf("The value of d is: %d\n",d);
}
```

IF…ELSE CONDITIONALS

We can then make decisions based on these comparison results using ***if…else conditionals***. These run different blocks of code depending on the condition.

```
#include <stdio.h>

int main(void) {
  // Try changing these values:
  int x = 3;
  int y = 4;

  if (x == 0) {
    // Statements in here will run if
    // x equals zero.
    printf("A\n");
  } else if (y > 2) {
    // Statements in here will run if
    // x does not equal zero, and y is
    // greater than two.
    printf("B\n");
  } else {
    // Statements in here will run if
    // x does not equal zero and y is not
    // greater than two.
    printf("C\n");
  }
}
```

TASK – COURSE GRADE CALCULATOR

Hop onto this code template:

https://repl.it/repls/DeadlyNegligibleClosedsource

You'll see that an integer, called `courseMark`, has been declared. In the space provided in the template, calculate the grade associated with the mark using the UoA standard grading scale and display this grade onto the screen. The link to the UoA standard grading scale is available in the template.

Change the value of `courseMark` to see if your program behaves correctly.

4 ARRAYS – SEQUENCE OF DATA

If a variable is a box for storing data, then an ***array*** is a line of boxes next to each other for storing a sequence of data. Arrays are declared in the following way:

```
<type> <name>[<array-size>];
```

For example, here we create an array of five integers:

```
int myArray[5];
```

To read from and write to an array, we use square brackets to index the position of the array we want to access. The ***index*** is a number that starts at zero and identifies the position of each *thing* in the array. Each *thing* in the array is called an ***element***. For example, to set values to each of the five elements of `myArray`, we would write:

```
myArray[0] = 1;
myArray[1] = 42;
myArray[2] = -3;
myArray[3] = 118;
myArray[4] = 11;
```

Similarly, to read from index-3 of `myArray`:

```
printf("Value is: %d\n", myArray[3]);
```

Caution

If you have an array of five integers, don't access from index 5 onwards! Your code will compile fine, but bad stuff will happen. Always make sure that the index is within the bounds of the array.

Arrays can be initialised with data when it is declared, like so:

```
int myOtherArray[4] = { 2, 1, 1, 8 };
```

If you remembered from earlier, we can represent textual data by using a sequence of characters, aka `char` arrays. These textual data are called ***strings***. They work just like normal arrays, except that they end with a special invisible character `'\0'` that marks where the string ends. You can initialise them using double quotes:

```
char myString[80] = "Hello World";
```

That piece of code behaves identically to:

```
char myString[80] = {
  'H',
  'e',
  'l',
  'l',
  'o',
  ' ',
  'W',
  'o',
  'r',
  'l',
  'd',
  '\0',
};
```

Other than that, we won't be using much of strings in robotics.

5 LOOPS – REPEATING INSTRUCTIONS

If you need to assign values to all elements of a small array, that is okay. But what if it was an array of a hundred integers? Thousand integers?

Instead of writing the same line of code over and over again, it is possible to repeat an instruction many times using a construct called `while` loops:

```
while (<condition>) {
  // ...code to be repeated...
}
```

where `<condition>` is an expression that determines whether to keep repeating or not. For example, if we ran the following code:

```
#include <stdio.h>

int main(void) {
  int i = 0;
  while (i < 2) {
    i = i + 1;
  }
  printf("done"\n);
}
```

What would happen?

First, `i` is set to `0`.
Then we reach the while loop.
Is `i < 2`? Yes, so we enter the while loop.
`i` gets incremented to `1`.
We reach the end of the while loop.
We jump back to the condition – is `i < 2`?
Yes, so we enter the while loop once again.
`i` gets incremented to `2`.
We reach the end of the while loop.
We jump back to the condition – is `i < 2`?
No, so instead of entering the while loop, exit it.
Print “done”.

The statements inside the while loop will continue to be repeated until the `<condition>` becomes zero.

Here is an example program to count the length of a word is using a `while` loop.

```
#include <stdio.h>

int main(void) {
  char myString[100] = "robots.";

  int i = 0;

  // While the i’th character is
  // not a period...
  while (myString[i] != '.') {
    // Move on to the next character.
    i = i + 1;
  }

  printf("%d letters", i);
}
```

TASK – GPA CALCULATOR

Hop onto this code template:

https://repl.it/repls/DangerousOrangeredSourcecode

You’ll see that an array of integers, called `courseMarks`, have been declared. In the space provided in the template, calculate the current GPA for a hypothetical student with the given course marks. Use loops to help you.

Assume that the courses use the UoA standard grading scale (link is available in the template).

Try changing the course marks in the array and see if your program still works as you’d expect.

6 FUNCTIONS – REUSING CODE

Do you remember `printf` from the first example? That is a function. A ***function*** (also known as a subroutine) is a group of statements that performs a particular task. Once a function is defined, you can then use the function multiple times in your program.

FUNCTION CALLS

The process of using functions is called "***calling a function***", also known as ***invoking*** or ***dispatching*** the function. To call a function called `aFunction`, you write

```
aFunction();
```

Some functions expect some input data, called ***arguments***. For example, `printf` expects at least one argument (and an additional argument for each specifier). The arguments are given to the function call inside the round brackets one by one, separated by commas. Note that the order that the arguments go in matters.

```
aFunction(arg1, arg2, arg3);

// Example of passing three arguments to
// the printf call.
printf("2nd arg: %d, 3rd arg: %d\n", 2, 5);
```

DEFINING FUNCTIONS

You can create your own functions in the following way:

```
<return-type> <function-name>(<args>) {
  // Write your code here.
}
```

where `<return-type>` can be `void`, `int`, `float`, `char`, and many others. For example, we create a function called `showMeaningOfLife`, which prints 42 onto the screen.

```
void showMeaningOfLife() {
  int yes = 42;
  printf("Meaning of life: %d\n", yes);
}
```

To call this function, you could write:

```
showMeaningOfLife();
```

Functions can give data back to the code that called the function. This is called *returning* data. The `showMeaningOfLife` function doesn't return anything, so we wrote `void` as the return type. We can write the function so it returns the number instead of printing it, using the `return` keyword:

```
int getMeaningOfLife() {
  printf("Returning the meaning of life\n");
  return 42;
}
```

We can then call this function inside an expression:

```
int niceNumber = getMeaningOfLife() * 10;
// niceNumber now has the value of 420
```

Functions can also accept input data as we saw earlier. To create functions that accept arguments, we write the argument types and names within the round brackets like this, just like declaring variables:

```
int absoluteSum(int x, int y) {
  int sum = x + y;
  if (sum < 0) return -sum;
  else return sum;
}
```

As examples of how to call this function:

```
printf(
  "The absolute sum of -3 and 1 is: %d\n",
  absoluteSum(-3, 1)
);
int x = absoluteSum(4, -absoluteSum(-5, -
10));
```

Whoa, that printf code above looks weird

Yes. In C, you can add excessive spaces and new-lines in your code, and it will still be correct and functional. Here, we've split the `printf` function call onto four lines to help readability.

THE PROGRAM ENTRY POINT – INT MAIN

Remember these two special lines in our programs?

```
int main(void) {
  ... lines skipped ...
}
```

This defines a special function called `main`. This is the ***entry point*** of the program, the first function that gets called.

Why does it return an `int`? This is the program's ***exit code*** which signals whether the program succeeded or failed. While it's an important and useful feature, we will not be using it in our robotics programs so don't worry about it for today.

Why Write Functions?

In the next exercise, you will solve a problem by splitting it into smaller problems and solving each subproblem as a function.

In engineering, we work with large, complex systems and we need to think in higher levels of abstraction. Not only is it tidier, easier to understand and easier to maintain, but it also saves you from copying and pasting code each time you want to do the same thing.

7 TASK – A HYPOTHETICAL ROBOT

Imagine, in a hypothetical situation, that your sleep-deprived-self decided that instead of studying, you'll spend the night programming a line-following robot for some strange university club. To help you achieve this, some fellow friends broke down the problem into the following smaller subproblems and asks you to complete them.

First, write a function called `detectLine` that accepts the following two arguments:

1. An integer called `sensorValue`, and
2. An integer called `prevDetected`.

The `detectLine` function should return an integer. When `sensorValue` is greater than `3000`, it should return `1`. When `sensorValue` is less than `2000`, it should return `0`. Otherwise, it should return `prevDetected`.

Next, write a function called `getSteering` that accepts the following four input arguments:

1. An integer called `leftDetected`,
2. An integer called `midDetected`,
3. An integer called `rightDetected`, and
4. An integer called `prevSteering`.

The first three arguments have a value of `0` or `1` depending on whether the left, middle or right sensor has detected a line.

The `getSteering` function should return an integer. When the left or right sensor detects a line, it should return `-1` for left and `1` for right. When only the middle sensor detects the line, it should return `0`. When none of the three sensors detect the line, it should return `prevSteering`.

Finally, write two functions called `driveLeftMotor` and `driveRightMotor`, both of which accepts a single integer argument called steering and both of which should return an integer:

- When steering is `0`, both functions should return `127`.
- When steering is `-1`, `driveLeftMotor` should return `40` while `driveRightMotor` should return `127`.
- When steering is `1`, `driveLeftMotor` should return `127` while `driveRightMotor` should return `40`.

To test your code, follow this link and write your code in the spaces provided:

https://repl.it/repls/OpulentWholeRatio

8 THAT'S ALL FOR THIS SECTION

Congratulations for reaching this far! Don't worry if you're behind or ahead. Everyone has their own pace, and we just hope you've had some fun today.

Please feel free to jump straight to the next section, where we start programming robots.

If you'd like to learn more about C, we recommend you Google some C tutorials online. While you're programming, you might find having a reference nearby handy. For this, we recommend the following resource: https://en.cppreference.com/w/c.