C/C++ Programming Module
Sem I 2023-24

July 2023

Introduction

This module is targeted to kick-start students in programming in C and C++ by introducing them to the basic syntax and essential features of the languages. For a new programmer, it should provide a good overview of everything they need to start coding up algorithms and creating data structures. For more experienced programmers, it may serve as a reference to quickly review syntax. This is inspired by Prof. Amitabha Bagchi's Java module. [1]

A question which may come to mind - why C? There are languages like Python out there which seem much easier and friendlier (and they have plenty of applications!). A primary reason for learning C/C++ is *performance*. We spend large amounts of efforts optimizing the algorithms we write and the data structures we create to ensure they run as fast as possible. Using a language like C ensures that well-written code designed for high performance purposes runs as fast as it *deserves to*!

Of course it is impossible to cover everything in a few pages. The internet is your best friend in case you feel confused about any of the concepts or you encounter something not present in this module. Good luck!

Contents

[1]C/C++ Module Version 3 by Viraj Agashe, suggestions and corrections by Ganraj Achyutrao Borade, Tanish Gupta and Kanishk Goel. To provide suggestions or point out errors, mail here.

1 Getting Started

Let's start with talking about C. Till now, some of you may have had experience with languages like `Python`, which are interpreted languages, which means they are executed line-by-line by another program called the *interpreter*. A program written in C on the other hand is first converted completely to a file which the machine can execute directly. The program which carries out this conversion is called a *compiler* and hence C is a compiled language.

So, the first requirement to start programming in C is to install a C compiler on your computer. Note that this module will assume you are using a Linux distribution like Ubuntu. A common C compiler is `gcc`, which stands for GNU Compiler Collection. You can check if `gcc` is already installed by running the following command in a terminal:

```
gcc --version
```

If it prints out the version, you are good to go. If not, installing it is as simple as running:

```
sudo apt-get update
sudo apt-get install build-essential
```

Now let us write our first C program. Create a file `hello.c` and open it with an editor of your choice. Let us write the following program:

```c
#include <stdio.h>
int main() {
    printf("Hello world!\n");
    return 0;
}
```

Don't worry about what the different portions of this program mean as of now - we'll tackle these later. (Do notice that each statement in C is terminated by a semicolon) Once you're done, open up the terminal. We are now ready to compile the C file. Run the following command in the terminal (open in the same folder as the C file):

```
gcc hello.c -o hello.out
```

The `-o` flag specifies the output path of the *executable* generated by the compiler. If we don't specify this, it will generate a file `a.out` by default. You can run the executable as:

```
./hello.out
```

This should print `Hello World!` on the terminal. Et voila!

2 Basics of the C Language

2.1 Variables

Variables in C are used to store and manipulate data. They have a specific data type that determines the kind of data they can hold, such as integers, characters, floating point numbers, etc. A **variable declaration** is a statement which specifies the name and type of a particular variable. For example,

```c
int var1; // Declaring an integer variable
```

On the other hand, a statement which assigns a value to a variable is called a **variable definition**. Note that you can also declare and define a variable simultaneously, as:

```c
int x = 3; // Defining an integer variable
```

A use of a variable means when it appears on the RHS of an expression. In C, you must ensure that Variables are declared before they are used (or you will get an error when you try to compile the program).

You may have noticed text (in green) preceded by a `//`. This is a **comment**. This text is ignored by the compiler. Comments are often used for documenting code and making it more readable. You can also write comments across lines by using `/* ... */`. For example,

```c
/* This is a
multi line comment
*/
```

2.2 Functions

Functions in C are reusable blocks of code that perform a specific task. They help organize the program into smaller, manageable units. Functions have a return type, a name, and may take input parameters. Note that we need to specify the types of the input parameters too. Here's an example of a declaration of a function `mul` which takes in two integer inputs and outputs an integer:

```c
int mul(int a, int b); // Function declaration
```

To define a function, we need to provide the implementation of the function's code. Every function, except one with a `void` return type needs to have at least one `return` statement, returning an expression of the return type. For example:

```c
int mul(int a, int b) {
    int product = a*b;
    return product;
}
```

Functions can be called from other parts of the program to execute their code and return a value if necessary. For example, a function call looks like:

```c
int y = mul(2,3); // Stores the value 6 in the integer y
```

2.3 Scope

Scope refers to the region of the program where variables are visible and accessible by their declared names. Usually, a scope is delimited by a set of matching curly parentheses (i.e. `{ }`). In C, variables can have either local or global scope.

Local variables are declared within a specific block, such as inside a function. They are only accessible within that block and cease to exist once the block is exited. Typically in C, local variables are declared at the beginning of a function scope (and not mixed with the program logic). Here's an example of a local variable:

```c
void f1() {
    int x = 5; // Local variable
    // Code that uses x
}
void f2() {
    // Code which cannot use x
}
```

What's Different in C++: In C++ you are free to mix declarations with the program logic.

Global variables, on the other hand, are declared outside of any specific block, usually at the top of the program, and are accessible throughout the program. Global variables retain their values until the program terminates. Here's an example:

```c
int n = 101; // Global variable
void random() {
    // Code that uses n
}
```

2.4 Types

C provides several built-in data types to represent different kinds of values. Each data type determines the range of values it can hold and the operations that can be performed on it. Here are some commonly used data types in C:

- `int`: Used to store whole numbers (typical size is 32 bits).
- `long`: Used to store larger whole numbers (typical size is 64 bits)
- `char`: Used to store individual characters.
- `float`: Used to store single-precision floating-point numbers.
- `double`: Used to store double-precision floating-point numbers.
- `void`: Represents the absence of a type. It is commonly used as a return type for functions that don't return a value.

2.5 Structure of a C Program

A C program is usually organized into various sections. Here's a conventional structure of a C program:

```c
// Preprocessor directives
#include <stdio.h>

// Function definitions and Globals

// Main function
int main() {
    // Variable declarations

    // Program logic

    // Return statement
    return 0;
}
```

Let us discuss these one at a time:

1. Preprocessor Directives: These are statements which inform the compiler to perform some specific actions before compiling the code. Briefly, the preprocessor directive `#include <stdio.h>` tells the compiler to make available the functionalities in the `stdio` library for us to use in our C program. [2]
2. Main Function: The main function is essentially the "entry point" into the C program, from where the execution starts. The program logic is written within the main() function, and the return statement signifies the end of the program execution.
3. Function Definitions: Various functions which may be called from main, or from other functions.

2.6 Arrays

Arrays in C are used to store multiple elements of the same data type in contiguous memory locations. They provide a way to efficiently manage and access a collection of elements. To declare an array, we need to specify the data type of its elements and the number of elements it can hold. Here's an example of declaring an integer array:

```
int numbers[5]; // Declares an integer array named 'numbers' that can hold 5 elements
```

We can also initialize an array at the time of declaration by providing the initial values within curly braces:

```
int numbers[] = {1, 2, 3, 4, 5}; // Declares and initializes an integer array with 5 elements
```

Array elements can be accessed using their index. The index starts from 0 for the first element. Here's an example:

```
int firstElement = numbers[0]; // Accesses the element with index 0
numbers[2] = 7; // Sets the element at index 2 to 7
```

2.7 Control Flow

Control flow statements allow us to control the execution of your program based on certain conditions, or to repeat a block of code multiple times. These are essential for writing any non-trivial programs.

2.7.1 if-else Statements

An if-else statement is used to perform different actions based on a condition. Here's an example:

```
if (temperature >= 30) {
    printf("It's a hot day.");
} else if (temperature >= 20) {
    printf("It's a pleasant day.");
} else {
    printf("It's a cool day.");
}
```

2.7.2 For Loop

The for loop is used to iterate over a block of code a specific number of times. For loops use an iteration variable. A for loop definition consists of 3 stages:

- Initialization: This code is used to initialize the iteration variable.
- Termination Check: This code checks if the condition to exit the loop has become true.

[2] stdio.h is actually a header file which contains the declarations of the functions in the stdio library (since we need declaration before use). The implementations of these functions are *linked* later by the compiler into the executable.

- Update: This code typically updates the iteration variable after one iteration of the loop body

After the definition, there is a loop body. Here's an example:

```
int i; // The iteration variable
for (i = 0; i < 5; i++) {
    printf("%d\n", i);
}
```

This for loop will execute the code block five times, with the variable 'i' ranging from 0 to 4.

What's Different in C++: In C++ (and some non-standard versions of C), you can initialize the iteration variable while specifying the for loop. For example,

```
for(int i = 0; i < n; i++) {
    // Loop body
}
```

2.7.3 While Loop

The while loop in C is another control flow statement that allows you to repeatedly execute a block of code as long as a certain condition is true. The while loop is typically used when an expression for the number of iterations is not known beforehand. Example:

```
while (condition) {
    // Code to be executed
}
```

The condition is evaluated before each iteration of the loop. If the condition is true, the loop body is executed. If the condition is false, the loop terminates, and control passes to the next statement after the loop. The while loop continues executing the loop body as long as the condition remains true. It is important to ensure that the condition eventually becomes false to prevent an infinite loop.

2.8 Pointers

Pointers in C are variables that store memory addresses. They allow you to manipulate data indirectly by referring to their memory locations.

2.8.1 Declaration and Initialization

To declare a pointer, you need to specify the data type it points to followed by * symbol and the name. Here's an example:

```
int *ptr; // Declares an integer pointer named 'ptr'
```

To initialize a pointer, you can assign it the address of another variable using the address-of operator &. Here's an example:

```
int num = 10;
int *ptr = &num; // Initializes 'ptr' with the address of 'num'
```

2.8.2 Dereferencing Pointers

Accessing the value stored at a memory address pointed to by a pointer is known as *dereferencing*. We can dereference a pointer using the * *operator* (Note that the purpose is different from the symbol used in the declaration of a pointer). Here's an example:

```c
int num = 10;
int *ptr = &num; // 'ptr' points to 'num'
*ptr = 3; // Changes the value of num to 3
printf("The value of the number is %d", num); // Prints 3
```

This will print the value 3, as the previous statement accesses and modifies the value stored at the memory location pointed to by 'ptr' (which is actually the address of num).

2.9 Input-Output

For input and output, we must include the `stdio.h` header file, which has the declarations for the input/output functions in C. Consider the following example:

```c
int number;
printf("Enter a number: ");
scanf("%d", &number);
```

We have already seen the use of `printf` earlier. For scanf, we write the format specifier `%d` first, which tells us an integer is expected, and we also pass the address of the integer using the `&` operator.

2.10 Strings

In C, a string is represented as an array of characters terminated by a null character (`\0`). C strings are commonly used to store and manipulate text or sequences of characters. We declare a string as a character array. For example:

```c
char str[] = "Hello, World!";
```

In the above example, we declare a character array named 'str' and initialize it with the string "Hello, World!". The size of the array is automatically determined based on the length of the string, including the null character.

C provides several functions in the standard library (`<string.h>`) to perform operations on strings, such as:

- `size_t strlen(const char* str)`
  Returns the length of the string `str` by counting the number of characters before the null character. The return value is of type `size_t`, which represents the size of an object in bytes.

- `char* strcpy(char* dest, const char* src)`
  Copies the contents of the string `src` to the string `dest`. Both `dest` and `src` should be null-terminated character arrays. The return value is a pointer to the destination string `dest`.

- `char* strcat(char* dest, const char* src)`
  Concatenates the string `src` to the end of the string `dest` by appending the characters of `src` to `dest`. Both `dest` and `src` should be null-terminated character arrays.

- `int strcmp(const char* str1, const char* str2)`
  Compares the strings `str1` and `str2` and returns an integer indicating their relationship. The return value is negative if `str1` is less than `str2`, zero if they are equal, or positive if `str1` is greater than `str2`.

Here is an example of usage of strings:

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str1[] = "Hello";
    char str2[20];
    // Length of a string
    int length = strlen(str1);
    // Copying a string
    strcpy(str2, str1);
    printf("Copied string: %s\n", str2);
    return 0;
}
```

3 Advanced Concepts

3.1 Structs

In C, a struct is a composite data type that allows us to group together related variables of different data types. It allows us to create a "blueprint" so that we can hold multiple values under a single struct type. We can use this blueprint to create instances of our struct, just like we create instances of other types when declaring variables. Here's an example of defining and using a struct in C:

```c
struct Point {
    int x;
    int y;
};

// Creating an instance of the struct
struct Point p1;

// Accessing struct members
p1.x = 10;
p1.y = 20;
```

In this example, we define a struct called "Point" that has two members: "x" and "y", both of type int. We create an instance of the struct named "p1" and assign values to its members using the dot operator.

3.2 Malloc

We say the **lifetime** of a variable is the time during which the memory allocated for a variable is accessible during program execution (contrast this with the scope, which is the time during which we can refer to the variable by name). The lifetime of local variables is simply the scope it is contained in. As soon as control exits the scope, the variable is no longer accessible. The memory which contained the variable is considered to be deallocated.

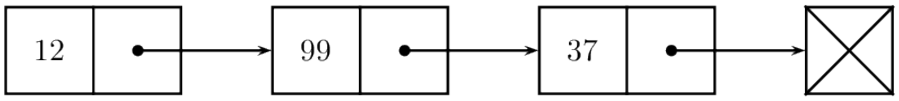

Figure 1: A Linked List

What if we need variables to last beyond their scope? Consider a basic data structure like a Linked List. In such a structure, each node of the linked list contains a pointer to the next node, as shown in the figure. Each node of the linked list would be a struct Node, which is defined as

```c
struct Node {
    int val; // The value in the Node
    struct Node *next; // Pointer to the next element
};
```

Suppose we create such a linked list inside a function foo(), but we want to use the linked list inside the main() function which called the function foo(). Note that the memory for any instances of struct Node declared inside foo() would have been considered *deallocated*. So, foo cannot just return a pointer to the first element of the linked list as it would be invalid. A solution for this problem is to allocate memory dynamically using `malloc`. The programmer can explicitly control the scope of memory which is allocated using malloc : the lifetime is from the time the memory is malloc-ed to the time we choose to "free" the memory.

Malloc allows us to request a specific amount of memory from the system, and it returns a pointer to the allocated memory block. Note that the memory returned by malloc does not have a "type". Therefore, we must **cast** it into the type of memory we want. Here's an example:

```
int *numbers = (int*) malloc(sizeof(int)); // Cast the pointer to type int*
// Always check the output of malloc for NULL
if (numbers == NULL) {
    // Error handling if malloc failed
} else {
    // Use the allocated memory
}
```

The memory allocated using `malloc` is done dynamically at *runtime*, i.e. when the program is actually running. Due to this reason, `malloc` may fail due to insufficient or corrupted memory. So, we should always check the result of malloc for `NULL`.

We can also use this method for allocating space for arrays whose size we do not know when writing the program. The following code would give an error:

```
int n;
scanf("%d",&n); // Read the value of n
int numbers[n]; // This line would give a compilation error
```

However, if we use malloc, the code will work.

```
int n;
int *numbers;
scanf("%d", &n);
numbers = (int*) malloc(n * sizeof(int));
```

In the above code, `numbers` is now a pointer to the first element of an array of size n. We can treat and index it like a regular array.

3.3 Pass by Value and Pass by Reference

What happens when we pass arguments to functions in C? Typically, C uses *pass by value* semantics. What this means is that when a function call is made, before the function body executes, a *copy* of the arguments which are passed is made (i.e. only their value is important, not the memory location they refer to). This means that any changes made to the parameters inside the functions will *not* be reflected outside. For example,

```
void change(int a) {
    a = 3;
}
int main() {
    int a = 2;
    change(a);
    printf("%d",a); // Prints 2
}
```

If we want to make changes to the arguments, we can pass them by *reference*. We need to add the `&` symbol before the name of the argument in the function definition. We can pass the arguments like normal. For example,

```
void change(int &a) {
    a = 3;
}
int main() {
    int a = 2;
    change(a);
    printf("%d",a); // Prints 3
}
```

4 What's New in C++

4.1 Classes

A class is a user-defined data type that encapsulates data and functions into a single unit. An *instance* of a class is called an **object** - a class is essentially a blueprint for creating objects, similar to structs. Typically, the data members of a class are called **attributes** and the functions contained within them are called **methods**. Here is an example of a class in C++:

```
class Rectangle {
private:
    // Private Attributes
    double length;
    double width;

public:
    // Public Attributes
    bool is_square;

    // Constructor
    Rectangle(double l, double w) {
        length = l;
        width = w;
        is_square = (l==w);
    }
    // Method
    double calculateArea() {
        return length * width;
    }
};
```

In the above example, we define a class called `Rectangle`. Let us walk through the various components of the class we see above:

1. Inside the class definition, before specifying attributes and methods, we must specify whether they are **public**, i.e. can be accessed from outside the class, or **private**, which means they can only be accessed inside the methods of the class. These are called **access specifiers**.

2. The **constructor** is a function which is called whenever we instantiate an object, it is used to initialise the attributes/fields on an object, if required. If we do not specify a constructor, there exists a default constructor which takes no arguments and does nothing. Two ways of creating an object:

   ```
   int main() {
       Rectangle r1(3,5); // Creates a rectangle r1 of dimensions 3x5
       Rectangle* r2 = new Rectangle(3,5); // Creates and returns a pointer to a rectangle r2
   ```

```
    // ...
}
```

Note that the lifetime of the rectangle `r2` is till we delete it.

3. The method `calculateArea` is used to access the private attributes and return the value of the area. We can access the public methods and attributes of a class as follows:

```
Rectangle rect(4,2);
double area = rect.calculateArea();
bool sq = rect.is_square;
```

In case we have a pointer to an object instead of an object, we can access its public members using the `->` operator instead of dot.

Classes provide a powerful mechanism for organizing and modeling data and behavior in C++ and are the backbone of **object oriented programming** or OOPs. We define some basic OOPs terminology below:

1. **Abstraction:** Exposing an interface to the end user so that they only need to see the end output rather than the inner workings. For example, sending a message on WhatsApp involves clicking a button, which would call a function `send_message(...)` which would internally use some complicated logic to actually send the message.

2. **Encapsulation:** In classes, we are able to bundle up a lot of different types of data & objects and various methods to operate on them inside a single class. We can also restrict who can access this data (public/private) This is called encapsulation (making a "capsule").

3. **Instantiation:** An object is called an instance of a class - since a class is like a blueprint of an object. Therefore instantiation means creation of an object of a class.

4. **Inheritance:** Inheritance allows for code reusability and the creation of a hierarchy of classes. It allows a class to 'inherit' properties and behaviors from another class.

   - Base Class: The class which is being inherited from
   - Child Class: The class which is inheriting from the base class. It is also called subclass

   We can add more functionalities (extend) the base class as well as override some portions of the base class. For example,

```
// Base class: Animal
class Animal {
public:
    void makeSound() {
        std::cout << "Animal makes a sound" << std::endl;
    }
};

// Derived class: Dog
class Dog : public Animal {
public:
    void makeSound() {
        std::cout << "Dog barks" << std::endl;
    }
};
```

You can read about more complex OOPs concepts here.

4.2 New and Delete

In C++, dynamic memory allocation is typically done using the new and delete operators. They are similar to malloc and free in C but are specifically designed for working with objects. Here's an example:

```
int *number;
number = new int;
if (number == nullptr) {
    // Error handling
} else {
    // Use the allocated memory
}
// Free the allocated memory
delete number;
```

In the above example, we use new to dynamically allocate memory for an array of 5 integers. When we are done, we can free the allocated memory using the delete[] operator.

4.3 Iostream

In C++, input and output operations are performed using the `iostream` library, which provides the `cin` and `cout` objects for input and output, respectively. Consider the example:

```
#include <iostream>
using namespace std;

int main() {
int number;
cout << "Enter a number: ";
cin >> number;
cout << "You entered: " << number << endl;
return 0;
}
```

In the above example, we use the `using namespace std` directive for convenience. Otherwise, we must specify `std::` before `cin, cout` and `endl`.

4.4 Strings

In C++, the Standard Library provides a string class called `std::string` (or simply `string` if we use the `using namespace std` directive) that offers a more flexible way to work with strings compared to C-style strings. To use these, we need to include the `<string>` header. An example:

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string greeting = "Hello, World!";
    cout << greeting << endl;
    return 0;
}
```

The `std::string` class provides a wide range of member functions to manipulate and access strings. Some commonly used member functions include:

- `size_t length()`
  Returns the length of the string. This function is equivalent to `size()`.

- `void append(const string &str)`
  Appends the string `str` to the end of the current string (in-place).
- `string substr(size_t pos, size_t len) const`
  Returns a new string which is a substring of the current string, starting from the position `pos` and taking `len` characters.
- `size_t find(const string &str, size_t pos = 0) const`
  Searches for the first occurrence of the string `str` within the current string, starting from the position `pos`. It returns the position of the found substring if successful, or `std::string::npos` if the substring is not found.

The following snippet has some examples of the usage of these functions.

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    string str = "Hello";
    // Length of the string
    cout << "Length: " << str.length() << endl;
    // Appending a string
    str.append(" World!");
    cout << "Appended string: " << str << endl;
    // Extracting a substring
    string substr = str.substr(1, 5);
    cout << "Substring: " << substr << endl;
    // Finding a substring
    size_t position = str.find("World");
    cout << "Substring position: " << position << endl;
    return 0;
}
```

4.5 STL

The Standard Template Library (STL) is a powerful set of C++ template classes and functions that provide essential data structures and algorithms. You can start reading about STL here.
