Introduction to C programming

A Brief History to C

C language is one of the most famous programming languages and best-known middle-level languages.

Originally, it was designed for the development of the Unix operating system and implemented in assembly language by Dennis Ritchie and Ken Thompson in the early stage of 1970s at AT&T. By design, it employs a static type system, procedural paradigm and is to be compiled to provide low-level access to memory as well as translated into machine instructions with minimal runtime support.

The versions of C were implemented for various machines during the 1970s and 1980s as its popularity grew significantly after its birth. In 1983, a committee formed by American National Standards Institute (ANSI) established a standard specification of C (often referred to ANSI C or C89 nowadays) and was ratified a few years afterwards in 1989.

C has achieved great successes since its birth as it can be applicable to all contexts in which low-level access is frequently or runtime efficiency is given top priority. Its applications range from operating systems, embedded systems to website programming as its most famous applications are Unix and Windows kernel.

Hello World: A Toy Example

We will start by the following example code which simply outputs “Hello, World!” to know how a C program works. In the given example, we can see a function named **main** is defined, a function named ***printf*** is called.

In addition, we can see a **include** directive was placed in the beginning of the source. This is to tell compilers that we will use functions implemented in ***stdio.h***. This behavior is similar to Java import mechanism.

Please note using an external function without including its implementation results in failures of compilation. The last statement “***return 0;***” is the "exit status" of the program when it finishes.

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

Basically, rules in C are:

- All C programs should start from the main function at the execution time.
- External functions can be used if its implementation file (normally known as header files) are explicitly imported using include directive.
- Curly brackets to define scopes such as a function.

Data Types and Syntax

Basic Data types

Data types are expressed in the language syntax in the form of declarations for memory locations or variables. Data types also determine which types of operations or methods to be used for processing of data elements. Please read the following example code to get familiar with data types.

C provides users with basic arithmetic types such as integer, real numbers, characters, and booleans. The syntax to define an array is straightforward as shown in the following example code. In C, all data elements in one array must be the same with the type of the array and the length of an array is fixed once it is declared.

```c
int a = 10;        // an integer variable
float pie = 3.14; //  a float one
double pie = 3.14; //  a double one
bool bo = true; // a boolean one
char c = 'a';   // a character variable
int a[10];      //define an array of integer type, length is 10
```

struct type

struct is a compound data type that allows users to define multiple variables under one type name in a continuous block memory. You can access different members of the struct type variable via the struct declared name or a single pointer associated with it.

```
/*
The general way to define a struct type is:
struct tag_name {
   type member1;
   type member2;
};
Declare the struct with float members x, y,z
*/
struct point {
   float   x;
   float   y;
   float   z;
};

struct point p1; // define a variable with struct point type
P1.x = 0.0; // assign values to its members
P1.y = 0.2;
P1.z = 0.4;
```

for/while loop

Looping in C is similar to its counterparts in other languages. Please refer to the following syntax to understand for/while loops. You will see more real examples in the attached C source examples.

```
while (expression)
{
    /* ... */
    cont: ;
}

do
{
    /* ... */
    cont: ;
} while (expression);
```

```
for (expr1; expr2; expr3) {
     /* ... */
     cont: ;
}
```

Function Definition and Calls

To define a function:

```
return_type function_name( parameter list ) {
   body of the function
}
```

Pointers

C is featured by pointers as they are sometimes the only way to express a computation and brings more efficiency compact than it with other ways. A pointer is a variable that stores the memory address of a variable. Some key principles of pointers are:

- Every pointer has an associated type, indicating which type of variable the pointer points to.
- Every pointer has a value which is an address of a variable of the designated type. NULL (0) value indicates that the pointer does not point anywhere.
- The & operator can be placed ahead of a variable to return its address.
- Pointers are dereferenced with the * operator.
- Pointers and arrays are closely related by design.

Pointers and Addresses

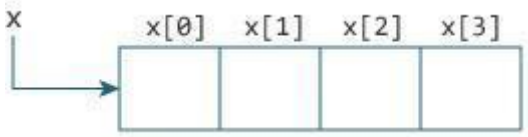
To understand pointers, let's look at the following example code.

```
int x = 1, y = 2;// define variables and an array
int z[10];       // define an integer array
int *ip;         // ip is a pointer to int */
ip = &x;         //let ip point to x */
y = *ip;         // y is now 1 */
*ip = 0;         // x is now 0 */
ip = &z[0];     //ip now points to z[0]
```

We first define a pointer, ***ip*,** for storing addresses of integer variables. Then we use an unary operator **&** to give the address of ***x*** to **ip**. Next, the value of ***x*** or we can say the value at the location of ip is retrieved and assigned to ***y***. Finally, we let ip point to array ***z***. The unary operator * is the dereferencing operator to access the variable to which the ***ip*** points. Please note that the variable can be practically replaced with ****ip*** in any context x could be used.

Pointers and Arrays

In C, pointers are closely linked to arrays. Array subscripting can be achieved with pointers alternatively. Indeed, using pointers to achieve array subscripting can be generally faster though such usages can be difficult to understand if pointers are not initiated. Using uninitiated pointers are deemed as bad practices normally.



Source: https://www.programiz.com/c-programming/c-pointers-arrays

Please read the following example. Can you answer the question in the comment?

```
int x[4];        // define an array for 4 elements only
int *px = &x[0]; // let the pointer px to point to x[0]
*px = 0;         // this is equivalent to x[0] = 0;
*(px+1) = 1;     // this is equivalent to x[1] = 1;
*(px+2) = 2;     // this is equivalent to x[2] = 2;
*(px+10) = 10;   // what would happen if this statement is
executed?
```

Dynamic memory allocation

As we showed in the array example, the length of an array is fixed once it is declared. However, it is very possible that we need to use arrays with varying length in runtime. In this case, we can use a pointer and ***malloc*** function to construct a dynamic array as shown in the following examples. You will see the full code of this example in ***dynamic_memory_allocation.c***,

```c
// Allocate space for an array with n elements
//of integer type where n is a variable */
const int n = 5;
int *ptr = (int *) malloc(n * sizeof(int));
if (ptr == NULL) {
    /* Memory could not be allocated */
} else {
    // Allocation is done
    free(ptr);  /* free the associated pointer. */
    ptr = NULL; /* The set NULL value to the pointer. */
}
```

Pointers and Function Arguments

In C language, passing arguments to functions is always by value. Therefore, there is no direct method to alter the value of a variable by passing it to a function.

```c
#include<stdio.h>
void swap_by_value(int x, int y) {
      int temp;
      temp = x;
      x = y;
      y = temp;
 }
int main(){
swap_by_value(a, b);  //Values of a and b remain the same
}
```

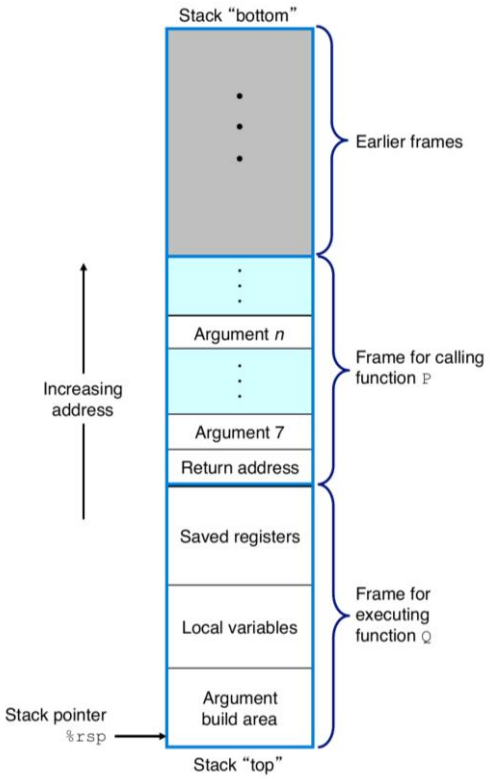
Given the above example, calling the function ***swap_by_value*** will not affect the value of the arguments a and b as the function simply copies the value of a and b at execution time. The desired results can be obtained by passing pointers as arguments as shown in the below code snippet. You will see the full code of this example ***call_by_pointers.c***.

```
void swap_by_pointer(int px, int py) {
      int temp;
      temp = *px; // access the variable at address stored in px.
      *px = *py;   // same as above.
      *py = temp;
 }
int main(){
swap_by_pointer(&a, &b); // values of a and b can be swapped
}
```

A Brief Introduction to Call Stack

When a function is called at runtime, certain steps have to be done. First data including parameter values and return value should be passed. Second, the control from one part of a program should be transferred to another part (lately called function).

In addition, space for the local variables of the function on entry will be allocated and deallocated on exit. All of these steps are manipulated by the program stack which uses a region of memory and is composed of stack frames. A general structure of a stack frame is depicted in the following figure.



Source: Computer Systems: A Programmer's Perspective page 220

As we can see from the figure, a frame pointer and a stack pointer appear in each of stack frames. The function being executed sits at the top of the stack and the stack grows from high addresses to low ones. The stack pointer always points to the top element of the stack.

Let us consider the case that a function P (caller) calls a function Q (callee). When P calls Q, it will push the return address onto the stack, indicating the location where the program should resume when execution of Q is over. In this case, the return address is seen within the stack frame for P.

The stack will be extended when code for Q requires space for its local variables or setup arguments for calling other functions within Q.

Labs

Example codes have been prepared for you. Please read through example codes and try to compile and run them in the lab environment.

You may use the following command to compile and run your C code with Linux system. Feel free to use your preferred IDE if you do not prefer to use command line tools.

```
gcc hello_world.c -o hello_world
./hello_world
```

Further Reading

Kernighan, B. W., & Ritchie, D. M. (2006). The C programming language.

Bryant, R., & O'Hallaron, D. (2003). Computer systems (pp. 153-229). Upper Saddle River, NJ: Prentice Hall.

What C/C++ programmers need to understand about the call stack https://www.cs.bham.ac.uk/~hxt/2015/c-plus-plus/stack.pdf

GCC online documentation https://gcc.gnu.org/onlinedocs/