Software Development and Parallel Computing with C++ – Class 1 (part 1)

Gustavo Ramírez

Brief intro to C

What is C?

- procedural, imperative computer programming language
- developed in 1972 by Dennis M. Ritchie
- essentially all UNIX application programs have been written in C
- formalized in 1988 by ANSI (American National Standards Institute)

Why C first?

- contains/represents the basics for other programming languages (C++, ...)
- most courses in C++ assume C knowledge
- depending on what you're doing, you could use only C and not go into C++ or any other OOP language

How to learn it (C)?

- plenty of online material
- good source: www.tutorialspoint.com
- do things **on your own**!

Why is it so widely used?

- easy to learn
- structured
- efficient (programs)
- can handle low-level activities
- its compilation is cross-platform

What we'll cover (today)

- basic structure of a program
- compilation and execution
- variables and types
- functions (within 'main')
- *#include* guards and *#pragma once*
- arrays
- pointers
- structs
- control statements

Basic structure of a C program

```
/* Lines beginning with # are directives read and
   interpreted by what is known as the preprocessor */
#include <stdio.h>

int main()
{
  //print msg on screen
  int i=15;
  printf("Hola, World!\n");
  printf("Value of i: %d\n", i);
  return 0;  //0 represents a successful execution and
             //return of the program
}
```

Compilation and execution

- save the code in a file named program_basic.c (or whatever name you want)
- compilation:

```
gcc program_basic.c -o program_basic
```

- execution:

```
./program_basic.c
```

Variables and types

Declaration of a variable:

```
TYPE IDENTIFIER1, IDENTIFIER2, ...;
```

Types of types:

1	Basic types	They are arithmetic types and are further classified into: (a) integer types and (b) floating-point types.
2	Enumerated types	They are again arithmetic types and they are used to define variables that can only assign certain discrete integer values throughout the program.
3	The type void	The type specifier void indicates that no value is available.
4	Derived types	They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

Typedef

Syntax:

```
typedef OLD_TYPE NEW_NAME;
```

Example:

```
typedef unsigned int BARE_INT;
```

Getting info from data types

```
#include <stdio.h>
#include <limits.h> // for int
#include <float.h>  // for float
#include <math.h>   // for pow(...)

int main() {
  printf("INFO - unsigned int\n");
  printf("Storage size for int: %d bytes \n", sizeof(unsigned int));
  printf("Range: %d - %d", 0, pow(2, sizeof(unsigned int))-1);

  printf("\nINFO - float\n");
  printf("Storage size for float : %d \n", sizeof(float));
  printf("Minimum float positive value: %E\n", FLT_MIN );
  printf("Maximum float positive value: %E\n", FLT_MAX );
  printf("Precision value: %d\n", FLT_DIG );

  return 0;
}
```

Functions

General form:

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

Specific example:

```
#include <stdio.h>

//Function declaration
void print_map_elem(char, int);
//Alternative declaration
//void print_map_elem(char key, int val);

int main()
{
  print_map_elem('k', 5);
  return 0;
}

//Full implementation of the function
void print_map_elem(char key, int val) //key and val are called 'formal para
{
  printf("Pair: %c, %d\n", key, val);
}
```

Static variables

```c
#include <stdio.h>

/* The x variable will be redefined to 123 each time
   that f() is called */
void f(){
  int x=123;
  x++;
  printf("f(): x=%d\n",x);
}

/* The use of 'static' makes x accessible globally, and the line
   'static int x=123;' is called only once i.e. the first time that
   g() is called */
void g(){
  static int x=123;
  x++;
  printf("g(): x=%d\n",x);
}

int main(){
  //non-incremental calls
  f(); f(); f();
  //incremental calls
  g(); g(); g();
}
```

#include guards and #pragma once

Check files:

```
program_pragma_once.c
program_pragma_once_level1.h
program_pragma_once_level2.h

program_include_guards.c
program_include_guards_level1.h
program_include_guards_level2.h
```

@ ../../examples_miscellaneous/

Arrays

Declaration:

```
type arrayName [ arraySize ];
```

Example:

```
...
#define LENGTH 5
double array_doubles[LENGTH] = \
    {10.0, 2.3, 5.4, 8.9, 3.3}; /* The number of values between
                                   braces { } cannot be larger
                                   than the number of elements that
                                   we declare for the array
                                   between square brackets [ ] */
//Alternatively
double array_doubles[] = {10.0, 2.3, 5.4, 8.9, 3.3};
...
```

Access:

```
double third_elem = array_doubles[2];
```

Passing arrays to functions

```
void myF(int params[], int size)
{
  ...
}
```

Return array from function

```
int *myF(...)
{
  ...
  /* If the array is created within the function,
     use static */
  static int array_o[17];
  ...
  return array_o;
}
```

Pointers

Let's print the address of a memory location:

```
#include <stdio.h>

int main()
{
  int test_arr[] = {2,3,7};

  printf("Value of second element: %d\n", test_arr[1]);
  printf("Address of second element: %p\n", &(test_arr[1]));
  printf("Address of second element: %p\n", test_arr+1);

  return 0;
}
```

Declaration of a pointer: TYPE *IDENTIFIER = ADDRESS;

Pointers

Alternatively:

```
#include <stdio.h>
#include <stdlib.h> //to enable malloc(...)

int main()
{
  int *p_arr;

  p_arr = (int*) malloc(3*sizeof(int));

  p_arr[0] = 2;
  *(p_arr+1) = 3;
  p_arr[2] = 7;

  printf("Value of second element: %d\n", p_arr[1]);
  printf("Address of second element: %p\n", &(p_arr[1]));
  printf("Address of second element: %p\n", p_arr+1);

  free(p_arr);

  return 0;
}
```

Pointers (to pointers!)

In the previous example, one could do:

```
...
int **p_p_arr = &p_arr; /* Pointer pointing to
                           a pointer */
...
```

Pointers

Check:

```
program_pointers.c
```

@ ../../examples_miscellaneous/

Pointers

Check:

`program_pointers_functions.c`

@ ../../examples_miscellaneous/

Control Statements: if

Syntax:

```
if(CONDITION1)
{

}
else if(CONDITION2)
{
  /* STATEMENTS */
}
...
/* MORE else if STATEMENTS HERE */
...
else
{
  /* STATEMENTS */
}
```

Control statements: if

Besides if: while, do while, the ternary ? operator, switch, for. We'll only cover if and for here.

The C programming language assumes any non-zero and non-null values as true, and if it is either zero or null, then it is assumed as false value.

Command Line Arguments

Compilation and execution again:

```
gcc program.c -o program
./program inp1 inp2 ...
```

Command Line Arguments

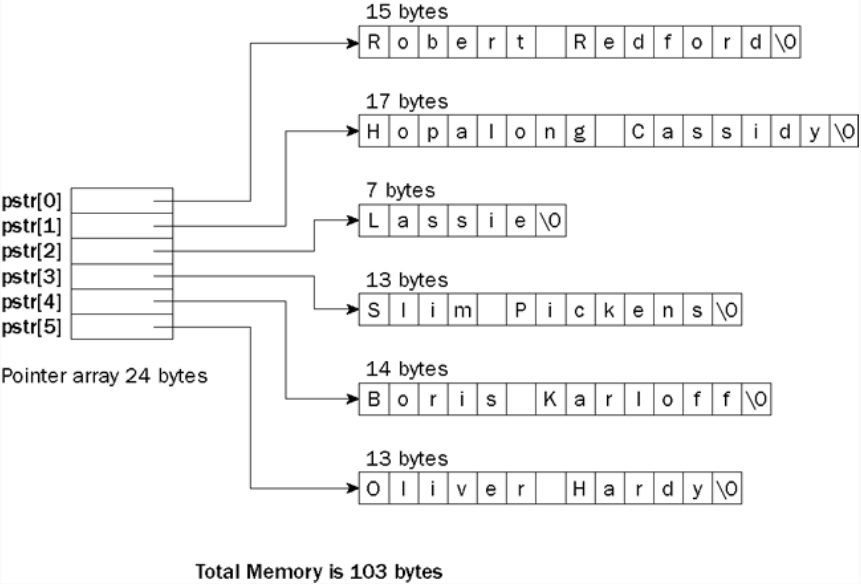
Example:

```
#include <stdio.h>

int main(int argc, char *argv[]){   //The second argument can be
                                     //written as: char **argv
  if(argc == 2) {
    printf("The argument supplied is %s\n", argv[1]);
  }
  else if(argc > 2)
  {
    printf("Too many arguments supplied.\n");
  }
  else{
    printf("One argument expected.\n");
  }

  return 0;
}
```

Command Line Arguments: memory



Control Statements: for

Syntax:

```
TYPE VAR;
for(INIT; CONDITION; EXEC)
{

}
```

where:

- INIT: initialisation of VAR (or any other necessary initialisations)
- CONDITION: when this condition is met, the loop stops and exits
- EXEC: something that is executed on each iteration

Control Statements: for

Example:

```
...
int i;
for(i=0; i<10; i++)
{
  printf("Iteration: %d\n", i);
}
...
```

Commands per iteration in the previous example:

```
Iter    Commands

1       initialize i=0, check i<10, call printf(...),
        increment: i++
2       check i<10, call printf(...), increment: i++
3       check i<10, call printf(...), increment: i++
...
11      check i<10, break
```

Extension of Command Line Args using for

```
#include <stdio.h>

int main(int argc, char *argv[]){  //The second argument can be
                                   //written as: char **argv
  if(argc == 2) {
    printf("The argument supplied is %s\n", argv[1]);
  }
  else if(argc > 2)
  {
    printf("The arguments supplied are: ");
    int i;
    for(i=1; i<argc-1; i++)
    {
      printf("%s, ", argv[i]);
    }
    printf("%s\n", argv[argc-1]);
  }
  else{
    printf("One argument expected.\n");
  }

  return 0;
}
```

Structs

Syntax:

```
struct [structure tag] {

   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

Structs

Let's say we want to avoid passing an array like this:

```
#include <stdio.h>

void print_arr(int params[], int size)
{
  int i;

  printf("Array: ");
  for(i=0; i<size-1; i++)
  {
    printf("%d, ", params[i]);
  }
  printf("%d\n", params[size-1]);
}

int main()
{
  int arr_i[] = {2, 3, 5, 89};
  const int size = 4;

  print_arr(arr_i, size);
}
```

and we want to pass it as a single structure. Use a struct!

Structs

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

struct arr_ints{
  int size; int *data;
};

//Redefine the name of the struct
typedef struct arr_ints ARR_INTS_P;

void print_arr(ARR_INTS_P arr_pack_x)
{
  int i;
  printf("Array: ");
  for(i=0; i<arr_pack_x.size -1; i++)
  {
    printf("%d, ", arr_pack_x.data[i]);
  }
  printf("%d\n", arr_pack_x.data[arr_pack_x.size -1]);
}

int main()
{
  ARR_INTS_P arr_pack;

  arr_pack.size = 4;
  arr_pack.data = (int*) malloc(arr_pack.size * sizeof(int));

  (arr_pack.data)[0] = 2; (arr_pack.data)[1] = 3; (arr_pack.data)[2] = 5; (arr_pack.data)[3] = 89;

  print_arr(arr_pack);
  free(arr_pack.data);
  return 0;
}
```

Final comments

There are many things we didn't cover today. Go and check ../notes.pdf for a more thorough intro to C!

End

End.