EXTMOD_01 Software Development and Parallel Computing - with C++
CLASS 1: BRIEF INTRO TO C

Gustavo Ramírez Hidalgo

C is a procedural and imperative computer programming language. Was developed in 1972 by Dennis Ritchie. Essentially all UNIX application programs have been written in C.

1 This course is mostly about C++. Then, why C first?

First of all, why C and/or C++ at all? Why not Fortran? Or Java? Or Python? (kidding about Python)

Fortran is used widely in HPC for science. One of the main reasons for not using Fortran is that not many people outside the HPC community know the language. But why? Partly because people learn to code in Compute Science (CS) classes/schools, where almost no one knows Fortran.

C/C++ are widely known. Period. This implies a good documentation and many (really, many!) blogs on issues and questions regarding C/C++, as well as a large number of online tutorials.

This is sort of like the question: should I use Django for back-end development? Or Ruby on Rails? Django is very well documented, supported and widely used. I mean, just the library. But Python as well. Python is used in web development, Machine Learning, Data Science, etc. But rails can represent less work to get lightweight web apps up and running. Also, RoR has a very large community. There are also technical issues to the question: what's the model used for the database?

Some people argue that Fortran is naturally suitable for HPC (on this regard: https://queue.acm.org/detail.cfm?id=1820518).

We'll use C/C++ here. This is just a (somewhat personal) preference.

The reason for introducing C before C++ is that (the former) contains/represents the basics for other programming languages (C++, ...). Also, most courses on C++ assume C knowledge. Finally, depending on what you're doing, you could use only C and not go into C++.

There are multiple reasons for C being so widely used:

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

2 How to learn it (C)

Many possible learning strategies. Three possible pilars:

- do things on YOUR OWN! Before checking the answer to a problem, or looking it up online, try to solve the problem yourself. Even if you sense you'll fail, always stick to solving any problem by yourself (or with a group of people, but in a non-passive way, always participating in the discussion)
- there's plenty of online material (blogs, http://stackoverflow.com/, etc.)
- a very good source to get started is: www.tutorialspoint.com

3 Basic structure of a program

A basic C program including most of the core syntax is the following:

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

int main()
{
  //print msg on screen
  printf("Hola,  World!\n");
  return 0;          /* 0 represents a successful
                     execution and return of
                     the program */
}
```

where /* TEXT */ represents a block of comments and // is the start of a line of comments. Lines starting with # are preprocessor directives: #include brings libraries resources (e.g. printf(...) is brought from the stdio library). Finally, return 0 represents a successful end of the program.

The line #include <stdio.h> tells the compiler to include the stdio.h before going to actual compilation. This is interpreted by the preprocessor (for a deep study of the C preprocessor: https://gcc.gnu.org/onlinedocs/cpp/ and https://www.tutorialspoint.com/cprogramming/c_preprocessors.htm).

The function main() is where the program execution begins.

Comments are ignored by the compiler. You cannot have comments within comments and they do not occur within a string or character literals.

4 Tokens (and others)

Either a keyword, an identifier, a constant, a string literal, or a symbol. Example:

```
printf("Hello, World! \n"); /* this is called a statement,
                            the whole instruction in
                            this line. Could be split
                            in multiple lines */
```

which is, keyword-wise, equal to:

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

Reserved words (keywords) in C:

auto, else, long, switch, break, enum, register, typedef, case, extern, return, union, char, float, short, unsigned, const, for, signed, void, continue, goto, sizeof, volatile, default, if, static, while, do, int, struct, _Packed, double.

Do not use these keywords as constants or variables or any other identifier names.

Whitespaces: blanks, tabs, newline characters and comments. All ignored by the compiler!

5 Variables and types

A C identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z, a to z, or an underscore followed by zero or more letters, underscores, and digits (0 to 9). C is case-sensitive.

The declaration of a variable goes like this:

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

and there must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them.

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.

Integer types: char, unsigned char, signed char, int, unsigned int, short (2 bytes), unsigned short, long, unsigned long.

Floating-point types: float, double, long double.

Getting info from data types:

```
#include <stdio.h>
#include <limits.h> // for size variables such as FLT_MIN
#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;
}
```

Integer overflows occur when the result of an arithmetic operation is a value, that is too large to fit in the available storage space. To understand further about overflow in C, read:

- https://splone.com/blog/2015/3/11/integer-overflow-prevention-in-c/
- https://en.wikipedia.org/wiki/Two%27s_complement
- https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html

And to understand more on how floating point numbers are represented in the computer: http://steve.hollasch.net/cgindex/coding/ieeefloat.html.

There are two kinds of expressions in C:

- lvalue Expressions that refer to a memory location are called "lvalue" expressions. An lvalue may appear as either the left-hand or right-hand side of an assignment.
- rvalue The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right-hand side but not on the left-hand side of an assignment.

Variables are lvalues and so they may appear on the left-hand side of an assignment. Numeric literals are rvalues and so they may not be assigned and cannot appear on the left-hand side.

Finally, types can be given new names. Syntax:

```
typedef old_type NEW_TYPE
```

and by convention, uppercase letters are used for the new name.

6 Functions

Group of statements that together perform a task. Example: main().

Parts of a function:

- declaration: name, return type and parameters
- definition: body of the function

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 parameters' */
{
   printf("Pair: %c, %d\n", key, val);
}
```

As seen above, if the function wants to be used within main(), then it has to be at least declared before usage.

There's a fundamental concept related to functions, that of scope: https://www.tutorialspoint.com/cprogramming/c_scope_rules.htm

Global variables: those variables defined outside functions are initialised before the code runs, and are visible to all functions. They can be hidden from external functions using static. If the variable is local, creating a variable inside a function with the same name makes the variable outside the function invisible, and only the variable inside is accessible.

Static serves the purpose of ”giving memory” to functions, and for encapsulation (https://alastairs-place.net/blog/2013/06/03/encapsulation-in-c/). More on static: https://stackoverflow.com/questions/572547/what-does-static-mean-in-c.

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables are undefined.

When calling functions, C uses a call by value: This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.

Alternatively, instead of functions one can use macros (sections of code to be inserted into the body of the program by the preprocessor): https://gcc.gnu.org/onlinedocs/cpp/Macros.html. A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro. By convention, they're written in uppercase.

7 *#include* guards and *#pragma once*

When one or more functions have to be used but their declarations are spread over different files, then those files have to be included (sometimes one file has to be included from multiple files). If one file is included multiple times from another file, an error might occur due to the multiplicity in the declaration of certain functions and/or variables. To prevent this, #include guards or #pragma once can be used for safety.

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

all @ ./examples_miscellaneous/

8 Arrays

Fixed-size sequential collection of elements of the same type. Contiguous memory locations!

Declaration: type arrayName [arraySize];

where arraySize has to be an integer greater than zero. 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];
```

It's important to note the following. If for example, one has the following code:

```
#include <stdio.h>

void myF(int params[])
{
  printf("Address within function: \t%p\n", params);
}

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

  printf("Address within main: \t\t%p\n", arr_i);

  myF(arr_i);
}
```

then the output of it is:

```
Address within main:            0x7ffde9cceea0
Address within function:        0x7ffde9cceea0
```

which means that both arrays are the same! There is no unnecessary double-copy of the entire array. This notion of avoiding extra copies of data (which can lead to unnecessary massive duplications of data) leads to the concept of pointers.

9 Pointers

For dynamic memory allocation (to be covered in the next section), pointers are necessary in C.

A pointer is a variable whose value is the address of another variable. Also, the address of a variable can be obtained using the & operator. In terms of ararys, addresses work as in the following example:

```
#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;
}
```

The declaration of a pointer is as follows:

```
TYPE *IDENTIFIER = ADDRESS;
```

The prints in the previous example can be performed now with pointers:

```
#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;
}
```

As a pointer points to the address of a variable, and because a pointer is a variable itself, a pointer can point to another pointer. In the previous example:

```
...
  int **p_p_arr = &p_arr;
...
```

A full example using pointers to pointers:

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

int main()
{
  int *p_arr = NULL; // always a good practice

  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]);
```

```c
  printf("Address of second element: %p\n", &(p_arr[1]));
  printf("Address of second element: %p\n", p_arr+1);
  printf("Size of pointer to int: %lu\n", sizeof(p_arr));

  int **p_p_arr = &p_arr;
  printf("\nValue of second element: %d\n", *(p_p_arr[0]+1));
  printf("Value of second element: %d\n", (*p_p_arr)[1]);
  printf("Address of second element: %p\n", &((*p_p_arr)[1]));
  printf("Size of pointer to pointer: %lu\n", sizeof(p_p_arr));

  //try the following two to see the error - in compilation!
  //printf("Value of second element: %d\n", *((*p_p_arr)[1]) );
  //printf("Value of second element: %d\n", &((*p_p_arr)+1) );

  /* More on sizes */
  short s1 = 4;
  short *p_s1 = &s1;
  printf("\nSize of short: %li\n", sizeof(s1));
  printf("Size of pointer to short: %lu\n", sizeof(p_s1));

  //always release allocated memory!
  free(p_arr);

  //try the following to see the error - in execution!
  //free(p_p_arr);

  return 0;
}
```

It's fundamental to know how to pass/return pointers to/from functions:

```c
#include <stdio.h>

void incr_nr(int *x)
{
  (*x)++;
}

int* decr_nr(int *y)
{
  (y[0])--;
  return y;
}

int main()
{
  int w = 10;
  int *x = &w;
  int *z = NULL;

  incr_nr(x);
  printf("After increment: %d\n", *x);

  z = decr_nr(x);
  printf("After decrement: %d\n", *z);

  //Try the following: compile once and execute multiple
  //                   times to the the different outputs
  //z++;
  //printf("Value of second (non-existent) entry: %d\n", *z);

  return 0;
}
```

which works exactly as in the case of arrays, i.e. there's no duplication of memory.

Finally, next to malloc() and free() there are more functions for memory management: https://www.tutorialspoint.com/cprogramming/c_memory_management.htm. Furthermore, check an extension of realloc called reallocarray (which checks for integer overflow): https://www.freebsd.org/cgi/man.cgi?query=reallocarray&sektion=3.

10 Control statements: if

We'll discuss here two control statements: if and for. But there are others (while, do while, the ternary ? operator, switch).

The if statement is simply a decision making operation:

```
if(CONDITION1)
{

}
else if(CONDITION2)
```

```
{
  /* STATEMENTS */
}
...
/* MORE else if STATEMENTS HERE */
...
else
{
  /* STATEMENTS */
}
```

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.

See the next section for an example of the use of the if statement combined with command line arguments.

11 Command Line Arguments

Until now, the main() function has had empty arguments (void). This function can take arguments as well, and those are precisely the user inputs. For example, after the program is compiled with:

```
gcc program.c -o program
```

the execution can have the form:

```
./program inp1 inp2 ...
```

where the number of inputs and the values of those strings, will be both encoded in argc and argv, respectively.

Example of the use of command line arguments:

```
#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;
}
```

12 Control statements: for

The for loop is simply that, a loop. 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

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

The for loop can be used to extend the extraction of command line arguments:

```c
#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("At least one argument expected.\n");
  }

  return 0;
}
```

13 Structs

Let's say we want to pass the information of an array as a whole, instead of array and size, both in separated variables. This can be done with the use of structs.

Originally, if we want to print the array we do the following:

```c
#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);

  return 0;
}
```

The use of structs allows us to pack both the array info and the size of the array into a single information package.

The syntax is the following:

```
struct [structure tag] {

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

The square brackets mean optional.

To access any member of a structure, we use the member access operator (.).

The example above (printing an array) can be re-written as follows, using structs to pack the data:

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

```c
#include <stdlib.h>

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

//Redefined 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;
}
```

Finally, when there's a pointer to a struct, one can access the fields of the struct with the use of an arrow operator. This means that the following:

```
...
printf("Size: %d\n", arr_pack.size);
...
```

prints the same as:

```
...
ARR_INTS_P *p_arr_pack = &arr_pack;
printf("Size: %d\n", p_arr_pack->size);
...
```

14 EXTRA

14.1 Constants and literals

constants refer to fixed values that the program may not alter during its execution. These fixed values are also called literals. More on constants and literals: https://www.tutorialspoint.com/cprogramming/c_constants.htm.

In particular, it's important to know two ways of defining constants:

- the #define preprocessor directive

```
#include <stdio.h>

#define LENGTH 10

int main() {
  printf("%d", LENGTH);

  return 0;
}
```

- the const keyword

```
#include <stdio.h>

int main() {
  const int LENGTH = 10;
```

```

    printf("%d", LENGTH);

    return 0;
}
```

14.2 Others

Links to some relevants topics that weren't covered here but should be studied for a full understanding of C:

Conditional compilation:
https://www.cs.auckland.ac.nz/references/unix/digital/AQTLTBTE/DOCU_078.HTM
https://stackoverflow.com/questions/4925300/program-compiled-on-linux-not-getting-code-in-ifdef-section
Storage classes:
https://www.tutorialspoint.com/cprogramming/c_storage_classes.htm
Operators in C (addition, etc.):
https://www.tutorialspoint.com/cprogramming/c_operators.htm
Strings:
https://www.tutorialspoint.com/cprogramming/c_strings.htm
Unions:
https://www.tutorialspoint.com/cprogramming/c_unions.htm
Bit fields (in structs):
https://www.tutorialspoint.com/cprogramming/c_bit_fields.htm
Program I/O:
https://www.tutorialspoint.com/cprogramming/c_input_output.htm
File I/O:
https://www.tutorialspoint.com/cprogramming/c_file_io.htm
Header files (includes):
https://www.tutorialspoint.com/cprogramming/c_header_files.htm
Type casting:
https://www.tutorialspoint.com/cprogramming/c_type_casting.htm
Error handling:
https://www.tutorialspoint.com/cprogramming/c_error_handling.htm
Variable (input) arguments in functions:
https://www.tutorialspoint.com/cprogramming/c_variable_arguments.htm
Function to measure time:

https://www.tutorialspoint.com/c_standard_library/c_function_time.htm

Using typedef with structs:

https://www.tutorialspoint.com/cprogramming/c_typedef.htm

Libraries from the C standard library:

https://www.tutorialspoint.com/c_standard_library/index.htm

System calls:

https://www.tutorialspoint.com/unix_system_calls/index.htm

Garbage collection:

http://www.linuxjournal.com/article/6679

https://dev.to/thecodeboss/programming-concepts-garbage-collection

http://wiki.c2.com/?LanguagesWithoutGarbageCollection

Recursion:

https://www.tutorialspoint.com/cprogramming/c_recursion.htm

About the meaning and use of size_t:

https://stackoverflow.com/questions/2550774/what-is-size-t-in-c

Developers best practices:

https://www.tutorialspoint.com/developers_best_practices/index.htm

On return(...) from main:

https://msdn.microsoft.com/en-us/library/sta56yeb.aspx

https://www.gnu.org/software/libc/manual/html_node/Exit-Status.html

http://man7.org/linux/man-pages/man3/errno.3.html

getopt(...) to get passed arguments:

https://www.gnu.org/software/libc/manual/html_node/Example-of-Getopt.html

gettimeofday() to measure time:

https://stackoverflow.com/questions/2150291/how-do-i-measure-a-time-interval-in-c

http://man7.org/linux/man-pages/man2/gettimeofday.2.html

Bitwise operations in C:

https://www.programiz.com/c-programming/bitwise-operators

https://www.cprogramming.com/tutorial/bitwise_operators.html