C Programming Primer

C is a general-purpose programming language strongly associated with UNIX, as it was developed to write the UNIX operating system. C does not support classes and objects so C++ was developed as an extension of C which does support classes.

Here is a first code in C:

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

Libraries to be used are requested in the beginning. Here <stdio.h> is a library which allows the use of input/output functions. A "main" function indicates where the program should start executing. A return statement terminates a function. Every C statement ends with a semicolon ";". C is immune to white spaces, new lines, etc. Use // for single line comments (from // to the end of the line) or /* ... */ for multiline comments.

All C **variables** must be identified **with** unique names which follow the rules:

- Names can contain letters, digits and underscores
- Names must begin with a letter or an underscore (_)
- Names are case sensitive (xVar and Xvar and xvar are different variables)
- Names cannot contain whitespaces or special characters like !, #, %, etc.
- Reserved words (such as ***int***) cannot be used as names

Basic Data Types

Type	**Description**
int	Stores integer numbers
float	Stores fractional numbers, containing one or more decimals. Can store 6-7 decimal digits
double	Stores fractional numbers, containing one or more decimals. Can store 15 decimal digits
char	Stores a single character/letter/number, or ASCII values
bool	Boolean

There is also a keyword **const** which is not a data type but can be added to make a variable read-only:
```
const int thisNumber = 5;
```

For print statements (command: **printf**) we specify the list of variables to be printed as a comma separated list. A formatting string may be used to specify text in which the variables can be embedded.

For example: printf("Variable x is %d and y is %f, z is %c and w is %s", x, y, z, w); assuming that x is integer, y is floating, z is a character variable, and w is a string variable. The %d, %f, %c, and %s here are "format specifiers" and they are placeholders to be substituted with the variables following in the list in that same order of appearance.

The operators are the typical ones +,-,*,/,++,-- and +=, -= etc. Check equality with '==', non-equality with '!=' and the usual <,>,>=,<= comparison operators. The logical operators are && (and), || (or), ! (not).

Arrays

To create an array, specify the data type (like int) and the name of the array followed by square brackets **[]**. Typically you must specify the size of the array explicitly (int myarray[5]) or implicitly (see next). Optionally, you can pre-populate an array by providing a comma separated list of values for it:
int myarray[] = {2, 3, 20, 80};
Multidimensional arrays also are ok: int matrix[2][3] = { {1, 4, 2}, {3, 6, 8} }; in which case matrix[1][0] holds "3".

C does not have a **String type**, so you must create a char type of array:
char message[] = "Hello World";   Special characters which must be part of strings must be escaped:
char txt[] = "It\'s alright, he said \"ok\" so don\'t worry.\n"; (\n produces a new line instead of the character n).

Structures

You can create a structure by using the **struct** keyword and declare each of its members inside curly braces:

```
struct MyStructure {   // Structure declaration
  int myNum;           // Member (int variable)
  char myLetter;       // Member (char variable)
}; // End the structure with a semicolon
```

The structure in a way declares a data type on the fly but to use it you must create a variable of it:

```
struct myStructure {
  int myNum;
  char myLetter;
};

int main() {
  struct myStructure s1;
  return 0;
}
```

Operating statements

Every programming language must be able to do variable assignments, conditional tasks, and loops. Programming languages mainly differ in the syntax of doing these and in how many ways (for convenience) they do these.

The if Statement

```
if (condition) {
  // block of code to be executed if the condition is true
}
```

or

```
if (condition) {
  // block of code to be executed if the condition is true
} else {
  // block of code to be executed if the condition is false
}
```

or

```
if (condition1) {
  // block of code to be executed if condition1 is true
} else if (condition2) {
  // block of code to be executed if the condition1 is false and condition2 is true
} else {
  // block of code to be executed if the condition1 is false and condition2 is false
}
```

or the short hand if:

```
variable = (condition) ? expressionIfTrue : expressionIfFalse;
```

The Switch Statement

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```

The While Loop

```
while (condition) {
  // code block to be executed
}
```

The For Loop

```
for (statement 1; statement 2; statement 3) {
  // code block to be executed
}
```

where ***Statement 1*** is executed (one time) before the execution of the code block; usually performs initialization for the control of the loop. ***Statement 2*** defines the condition for executing the code block; the loop exits if this statement does not evaluate to true. ***Statement 3*** is executed (every time) after the code block has been executed; usually used to auto-increment a loop control variable.

Break and **Continue**

The **break** statement is used to jump out of a segment of code such as a loop or a switch.

The **continue** statement is used to flush the remainder of a segment of code such as a loop (but continue with subsequent iterations of the same loop).

Functions:

Typically a function is defined with the following syntax:

```
void myFunction() {
  // code to be executed
}
```

void means that the function does not have a return value. But if it should then the appropriate type declaration should replace the void in the above syntax, for example int *myFunction*() {…}
In general the syntax is:

```
returnType functionName(parameter1, parameter2, parameter3) {
  // code to be executed
}
```

You may pass an array as parameter. Note that when passing it as an argument, you only need to use the name of the array, e.g. ***myFunction(myNumbers)***. But you must use the full declaration of the array when specifying it as a function parameter, e.g. ***void myFunction(int myNumbers[5])***. Example:

```
void myFunction(int myNumbers[5]) {  ←==
  for (int i = 0; i < 5; i++) {
    printf("%d\n", myNumbers[i]);
  }
}

int main() {
  int myNumbers[5] = {10, 20, 30, 40, 50};
  myFunction(myNumbers);  ←==
  return 0;
}
```

File Handling

In C, you create, open, read, and write to files by declaring a pointer of type **FILE**, and use the **fopen()** function:

```
FILE *fptr
fptr = fopen(filename, mode);
```

where mode is r/w/a (read/write/append)

Write into a file using **fpritf(fileHandler, "Stuff to write");** Always close a file that you do not need using **fclose(fileHandler);** This will destroy the handler but it can be reused as a new handler.

The fine stuff

The function **fork()** generates an operating system call which spawns a new process. A process (referred to as parent process) that executes a fork() causes the generation of a new child process. The new (child) process is a copy (replica) of the (calling) parent process. This child process will have its own memory space, including its own stack. The child process will **not** share the same stack as the parent process. At the time of its creation, the child process has a working space which is copied from the parent's space but it is from there on separate from that of the parent. The child process will have its own set of registers and its own program counter, but it will have the same code as the parent process and its initial data is what is copied from the parent at the time of the child's creation.

The fork() function does not take any parameters and it returns an integer value both to the parent and to the child which it creates (but they are different for each).

- If fork() fails to create a child process then it returns a negative value to the parent.
- The fork() returns a value of zero 0 to the child process.

- The fork() returns a positive integer value to the parent process and that is the system process ID of the child process.

The function **getpid()** produces the (system) process ID of the process which executes the getpid(). If a process p1 spawns a child process p2, and then p2 executes the function **getppid()**, that produces the (system) process ID of the parent process p1.

A process can be paused by executing **sleep(n)** (n is number of seconds).

By using the **wait()** function we signal a process to wait for all its children to finish before terminating itself.