Data Structures and Algorithms

Lecture 2

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."

Anonymous

C Language Review - Contents

- Variables and data types
- Loops
- Branch control – conditional constructs
- Functions
- Preprocessor directives
- Input/ Output
- Usual Libraries
- Structures
- Pointers
- Debugging

C Language Review - Variables

- Variables are entities that can store different values during the life time of a program
- Variables are **always defined** as:

```
data_type variable_name;
```

- The **data type** represents the set of values that a variable can take; For example, a **char** type variable can take any integer value between -128 and 127.
- The **variable name** is an identifier that can be used to refer the variable in the scope that it is defined in; A variable can exist and be used only in a specific scope (area) in your program;
- An identifier, in C, must obey the following rules:
  - It can contain any of the following characters: lowercase character (a-z), uppercase character (A-Z), figure (0-9) and the underscore (_) character;
  - It must not start with a figure.

C Language Review – Variables (cont'd)

- Variables can be stored in memory in three different locations, depending on where and how they are defined:
  1. Global Variables – are stored in the program area of the code (area allocated when the executable is loaded in the memory); they are visible in all the functions defined in the same file and that memory is only freed at the end of the program.
  2. Local variables – are allocated on the execution stack, when the function they are defined in is called; they are only visible inside that function and the memory is freed when the function returns (except static local variables which behave that global variables).
  3. Dynamically allocated variables – are allocated in the HEAP memory, they exist until they are explicitly freed and they are referred to by pointers;

HINT: Avoid using global variables, it's a bad practice.

C Language Review – Variables (cont'd)

```
int globalIntVariable;

void doSomething(){
        int _localIntVariable;
        // Do something
}

int main(){
        doSomething();
        int *localPointer;
        localPointer = (int*)malloc(sizeof(int));
        // Lifetime of localPointer memory
        free(localPointer);
        return 0;
}
```

C Language Review – Variables (cont'd)

Home review:

What is the execution stack? How does it work?

C Language Review – Data Types

- A data type can be:
  - Primitive
  - Structure
  - Typedef
- A primitive data type is defined as:

```
sign_specifier size_modifier type
```

- Any of the three keywords can be missing, but at least one needs to be specified.
- The sign specifiers are: signed and unsigned
- The size modifiers are short, long and long long
- The types can be: char, int, float, double
- Use the sizeof operator to find out the size of a data type (or variable)

char	signed char	unsigned char	short
short int	signed short	signed short int	unsigned short
unsigned short int	int	signed int	unsigned
unsigned int	long	long int	signed long
signed long int	unsigned long	unsigned long int	long long
long long int	signed long long	signed long long int	unsigned long long
unsigned long long int	float	double	long double

Possible data types in C

```
"error: 'long long long' is too long for GCC"
```

GCC compilation error

C Language Review – Data Types (cont'd)

Type	Size	Signed	Range
char	8 bits	yes	[-127; 128]
unsigned char	8 bits	no	[0; 255]
short	16 bits	yes	[–32768; 32767]
unsigned short	16 bits	no	[0; 65535]
int	32 bits	yes	[–2147483648; 2147483647]
unsigned int	32 bits	no	[0; 4294967295]
long	64 bits	yes	[–9223372036854775808; 9223372036854775807]
unsigned long	64 bits	no	[0; 18446744073709551615]
long long	64 bits	yes	[–9223372036854775808; 9223372036854775807]
unsigned long long	64 bits	no	[0; 18446744073709551615]
float	32 bits	yes	Floating point low precision
double	64 bits	yes	Floating point medium precision
long double	128 bits	yes	Floating point high precision

C Language Review – Data Types (cont'd)

Home review:

1. How are negative numbers stored in memory?
2. How are floating point numbers stored in memory?
3. What is casting, and how is it used?

C Language Review - Loops

C provides three loop blocks:

1. while loops;
2. do – while loops
3. for loops

	while	do-while	for
Description	Verifies a condition at the start of every loop, and jumps after the while block if the condition doesn't hold.	Same as "while", only the condition is checked at the end of the loop	Complex loop statement containing an initialization expression, executed once, before the first loop, a condition that is checked at the start of each loop, similar to "while", and a end-of-loop section which is executed at the end of every loop.

Generic Block	`while(<condition>){`
	`  <loop block>`
	`}`
		`do{`
		`  <loop block>`
		`}while(<condition>);`
			`for(<init_exp>; <condition>; <end_loop_section>){`
			`  <loop block>`
			`}`

C Language Review - Branch control – conditional constructs

Whenever a DECISION needs to be made, a conditional statement has to be used:

1. if/ if-else statement
2. switch statement
3. ternary operator

	if	switch	ternary
Description	Verifies a condition and executes a section of code if the condition holds. Optionally, an "else" block can be specified, to be executed if the condition does not hold.	A switch statement is initiated with a variable expression and then, specifying a set of possible values for that expression with associated behavior. Optionally, it can have a "default" statement, which is executed if none of the other values match.	Simple operator that evaluates as different expressions depending on a condition.

Generic Block	`if(<condition){`
	`  <code_for_true>`
	`}else{`
	`  <code_for_false>`
	`}`
		`switch(<variable_exp>){`
		`  case <possible_value_1>:`
		`  <behavior_1>;`
		`  break;`
		`  case <possible_value_2>:`
		`  <behavior_2>;`
		`  break;`
		`  //…`
		`  default: <alternative_behavior>`
		`}`
			`<condition> ?`
			`<expression_for_true> :`
			`<expression_for_false>`

C Language Review - Functions

- Functions are sections of code which can be reused by being called more than one time;
- Similar to math functions, C functions have:
  - A name, which is a C identifier (see variables, same rules apply)
  - A return value (which in C is specified as a data type, or void, if the function does not return anything)
  - One or more parameters, or arguments, which in C is specified as a comma separated list of variable definitions

Math	C

$f: Z^2 \rightarrow Z$
$f(x, y) = x + y$
	`int f(int x, int y){`
	`  return x + y;`
	`}`

$a \in Z, a = f(1,3)$	`int a = f(1, 3);`

C Language Review - Preprocessor directives

- Preprocessor directives are executed before the compilation takes place;
- All preprocessor directive start with the hash (#) sign:
  - #include – used to include a header file, usually with function and constants definitions in the program (stdio.h, stdlib.h, etc.)
  - #define – used to define a macro which is a simple replacing of an identifier with another string;
  - #ifdef/ #ifndef/ #endif – are used to check if an identifier is defined for the preprocessor, and include a section of code for compilation or not (usually for debug purposes, or for different compilation environments)

C Language Review - Preprocessor directives (cont'd)

```
#include<stdio.h>
#define MY_INT 10
#define DEBUG
int main(){
    int a = 15;
    if(a == MY_INT){
        printf("If a is 10, then this message gets written!\n");
    }
#ifdef DEBUG
        printf("If DEBUG is defined, this message gets printed too!\n");
#endif
}
```

C Language Review - Input/ Output

Functions used to read from the standard input stream and write to the standard output stream:

- printf – used to print items on the standard output stream;
- scanf – used to read values from the standard input stream;
- gets_s – used to read a string containing white spaces, from the standard input stream
- getchar – used to read a single char from the standard input stream, no enter key is required, like in the case of scanf

All these functions are defined in the header file stdio.h

See http://en.cppreference.com/w/c/io/fprintf, http://en.cppreference.com/w/c/io/getchar, http://en.cppreference.com/w/c/io/gets

C Language Review – Usual Libraries

The following libraries are usually needed for writing C programs:

- stdio.h – Standard Input/ Output library, used for functions related to I/O from/ to keyboard, screen and files
- math.h – Mathematical related functions: cos, sin, sqrt, etc.
- stdlib.h – Standard Library containing conversion functions, pseudo-random number generators, memory allocation functions, etc. (see http://www.cplusplus.com/reference/cstdlib/)
- string.h – functions needed for processing character strings (see http://www.cplusplus.com/reference/cstring/)

C Language Review -Structures

- Structures are assemblies of variables that have meaning together.
- They are "composed" data types:

```
struct Person{
    char * firstName;
    char * lastName;
    unsigned char age;
    char * address;
};
```

- "struct Person" now acts like a data type so we can use it to create variables:

```
struct Person me;
```

- We can access "members" of a structure type variable by using the "." operator:

```
me.age = 30;
```

C Language Review – P  inters

- Pointers are addresses in memory to certain types of data;
- All pointers are stores on 32 bits, on 32-bit CPUs and on 64 bits on 64-bit CPUs.
- "`int * pA;`" means that pA doesn't actually hold an integer, it holds an address value and if we go and read the contents of the memory from that address, we will get an integer;
- We can use casting to change the type of the data that we expect to read from an address: this is a very serious security issue!
- Pointer arithmetic can be used to move from one address to the next.
- Operator to get an address for a variable is "&"
- Operator to get a value from an address is "*"

C Language Review - Debugging

- Debugging a C program is a very important skill you must acquire
- Learn to read and interpret the error message
- Figure out if what you're seeing is:
  a) A preprocessor error
  b) A compilation error
  c) A linking error
- Use GDB (GNU DeBugger) and any GUI front-end to analyze your code and run it step-by-step

Thank you!

*For next class, please review arrays in C