

Object Oriented Programming - Lecture 1

Diana Borza - diana.borza@ubbcluj.ro

February 21, 2022

- C/C++ language
- Compilation process. Debugging
- Data types
- C/C++ lexical elements
- Statements
- Pointers
- Summary

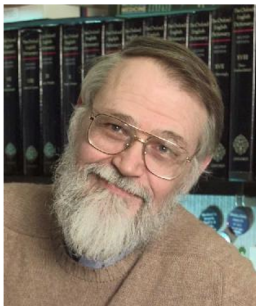
C/C++ programming language

Hall of fame

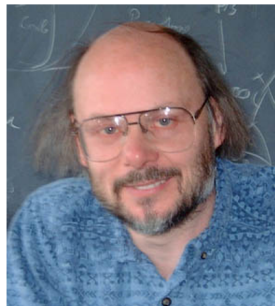
"No beard, no belly, no guru "



Dennis Ritchie



Brian Kernighan



Bjarne Stroustrup

Why use C/C++?

- **widely used**, both in industry and in education
- **hybrid, multi-paradigm language**: implements all the concepts required for object oriented programming
- **high-level programming** language; compiled language
- many programming languages are based on C/C++ (Java, C#). Knowing C++ makes learning other programming languages.
- C ranked second in the TOBIE index as of January 2022, and C++ is ranked fourth <https://www.tiobe.com/tiobe-index/>

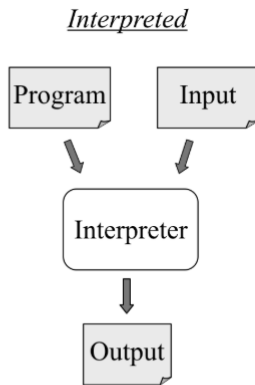
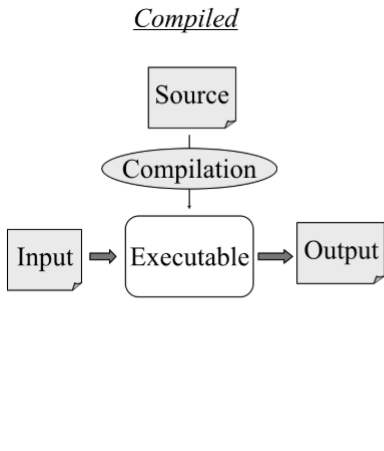
Why use C/C++?

- **rich library support** - many functionalities are already available, which helps in quickly writing code.
- **Speed:** C++ is the preferred choice when latency is a critical metric.
- C++ is an evolving, **highly standardized language**

Usage of the C/C++ programming language

- **Operating Systems** - C/C++ is the backbone of all the well-known operating systems
- **Libraries** - high-level libraries use C++ as the core programming language; e.g. *tensorflow* machine learning library
- **Graphics and Web Browsers** - require fast rendering and C++ helps in reducing the latency
- **Embedded Systems** - medical machines, smartwatches, etc. use C++ as it is closer to the hardware level as compared to other high-level programming languages.
- **Banking Applications:** process millions of transactions on a daily basis and require high concurrency and low latency support.

Compiled vs interpreted languages I



Compiled vs interpreted languages II

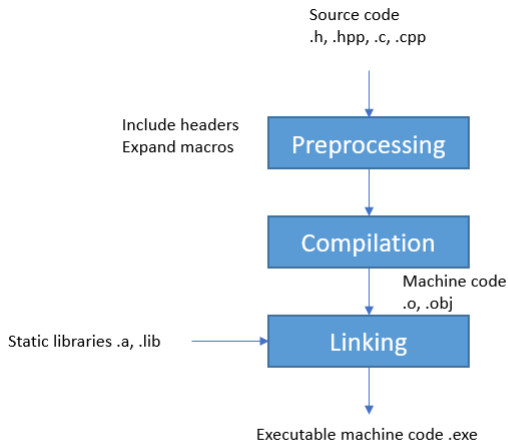
Compiled

- **input:** takes entire program as input
- **speed:** executes faster
- **workload** doesn't need to compile every time, just once
- generates intermediate object code
- **error checking** at compilation the entire program is checked
- **Examples:** C/C++, BASIC, C# (to bitcode)

Interpreted

- **input:** takes a single instruction as input
- **speed:** executes slower
- **workload** has to convert high level languages to low level machine code at execution
- does not generate any intermediate data
- **error checking** displays errors when each instruction is run
- **Examples:** python, Matlab, javascript, Ruby

Compiled languages

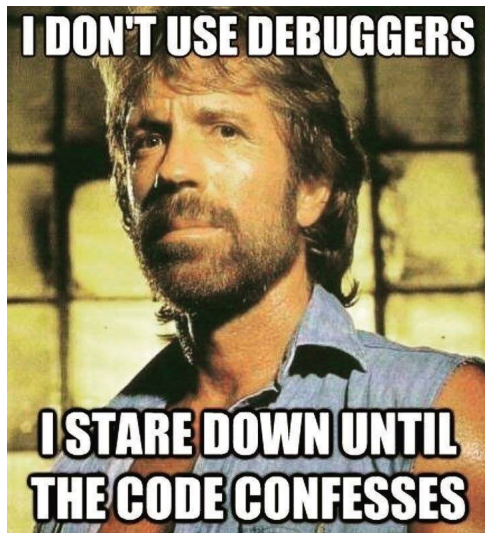


Compiling...



“20% of programming is writing code.
The other 80% is debugging”
Anonymous

- allows you to step through each line of code, as the program is running
- inspect the current state of the program: variables, call stack, expressions
- breakpoint - stop the program in certain points



Structure of a simple C/C++ program

- include directives: for using other modules, libraries

```
#include <iostream>
```

- main() function - the entry point of the program, called by the operating system to run it

```
int main(){  
    std::cout<<"Hello world"<<std::endl;  
    return 0;  
}
```

C/C++ is case sensitive.

- **Identifier**: Sequence of letters and digits, start with a letter or (underline). They are names used to identify program elements (which are not built into the language): functions, variables, constants, etc. E.g.: `max_value`, `_parent_obj`, `my_function`, etc.
- **Keywords** (reserved words): Identifier with a special purpose; Words with special meaning to the compiler; E.g.: `int`, `while`, `class`, `struct`.
- **Operators**: symbols that tell the compiler to perform specific mathematical or logical manipulations. E.g.: `+`, `-`, `!`, `*`, etc.

Literals: Basic constant values whose value is specified directly in the source code; E.g.: "Hello", 72, 4.6, 'c'.

Separators: Punctuation defining the structure of a program: e.g. ";", " ", "(", ")".

Comments: ignored by the compiler, but are used to explain the code, and to make it more readable

```
// This is a single line comment
/* This is a
   multi line
   comment */
```


The **type** of a variable determines the **domain of values** and a **set of operations** defined on these values. C/C++ are strongly typed languages.

- casting

Demo

C/C++ datatypes (datatypes.cpp)

C++ datatypes

Data type	Bytes	Range of values
char	1byte	-127 to 127 or by default
unsigned char	1byte	0 to 255
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
short int	2bytes	-32768 to 32767
unsigned short	2bytes	0 to 65,535
bool	1byte	false or true
long	4bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4bytes	0 to 4,294,967,295
long long	8bytes	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
unsigned long long	8bytes	0 to 18,446,744,073,709,551,615
enum	varies	-
float	4bytes	3.4E +/- 38 (7 digits)
double	8bytes	1.7E +/- 308 (15 digits)



Arrays

- *An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier.*
- If T is an arbitrary basic C/C++ type: **$T \text{ arr}[N]$** : arr is an array of length N with elements of type T ;
- indexes are from 0 to $n-1$;
- indexing operator: `[]`;
- multidimensional array: `arr[L][C]`

C string

- one-dimensional array of characters terminated by a null character `'\0'`
- standard library functions for string manipulation
 - `strcpy(s1, s2)` - Copies string `s2` into string `s1`.
 - `strcat(s1, s2)` - Concatenates string `s2` onto the end of string `s1`.
 - `strlen(s1)` - Returns the length of string `s1`.
 - `strcmp(s1, s2)` - Returns 0 if `s1` and `s2` are the same; less than 0 if `s1 < s2`; greater than 0 if `s1 > s2`.
 - `strchr(s1, ch)` - Returns a pointer to the first occurrence of character `ch` in string `s1`.
 - `strstr(s1, s2)` - Returns a pointer to the first occurrence of string `s2` in string `s1`.

! None of these string routines allocate memory or check that the passed memory is the right size.

Demo

C string (cstring.cpp)

- Record/structure - a linear, direct-access data structure with heterogeneous components
 - declared using `struct`
 - dot `.` operator

Demo

Structures (structures.cpp)

Pointers

- Every variable is a named memory location;
- **A pointer** is a variable whose value is a memory location (the address of another variable).
- **Declaration** same as declaring a normal variable, except an asterisk (*) must be added in front of the variable's identifier.

```
int *x ;  
char *str ;
```

- Operators
 - *address of* operator & - take the address of a variable;
 - *dereferencing* operator * - get the value at the memory address pointed to.

Demo

Pointers (pointers.cpp)

Variables

- A variable is a named location in memory;
- Memory is allocated according to the type of the variable;
- The types tell the compiler how much memory to reserve;
- The value of the variable is undefined until the variable is initialized for it and what kinds of operations may be performed on it;
- It is recommended to initialise the variables (with meaningful values) at declaration;
- Use suggestive names for variables.

- Fixed values that the program may not alter during its execution;
- Can be defined using the `#define` *preprocessor* directive, or the `const` keyword;
- Examples

```
#define PI 3.1415  
const float PI = 3.1415;
```

? What are the differences of using `const` vs. `#define` for constants?

C++ operators

- 1 Assignment Operator: `=`
- 2 Mathematical Operators: `+`, `-`, `*`, `+=`, `-=`, etc.
- 3 Relational Operators: `<`, `<=`, `==`, `>=`, `>`
- 4 Logical Operators: `&&`, `||`
- 5 Bitwise Operators: `&`, `|`, `~`
- 6 Shift Operators: `<<`, `>>`

C++ operators

8 Unary Operators: -, ++, !

9 Ternary Operator:

```
cond ? condition_true : condition_false
```

10 Comma Operator: to separate variable names/ expressions. (the value of last expression is produced and used)

Demo

Operators example (operators.cpp)

Statement:

- each of the individual instructions of a program
- all statements, except the compound statement, always end with a semicolon (;)
- are executed in the same order in which they appear in a program.

Type of C++ statements:

- Empty statement
- Compound statement
- Conditional statement: if, switch
- Loops: while, do-while, for

Demo

Statements (statements.cpp)



- **Function** - group of statements that is given a name, and which can be called from some point of the program.
 - **Declaration**
 - **Definition**
 - Actual and formal parameters
 - **Invocation**

Demo

Functions (function.cpp)

Functions - specification

- meaningful name for the function;
- short description of the function (the problem solved by the function);
- meaning of each input parameter;
- conditions imposed over the input parameters (precondition);
- meaning of each output parameter;
- conditions imposed over the output parameters (post condition).

/*

Computes the maximum value of vector v.

Inputs: v - 1D array, $n > 0$ size of the array

Output: returns the value of the maximum element from v */

float max_val(float v[], int n);

Functions - design guidelines

- Single responsibility principle.
- Use meaningful names (function name, parameters, variables).
- Use naming conventions (`add_rational`, `addRational`, `CONSTANT`), be consistent.
- Specify and test functions.
- Use test driven development.
- Include comments in the source code.
- Avoid functions with side effects (if possible).



Pass by value vs pass by pointer

• Pass by value

- Default parameter passing mechanism in C/C++.
- On function call C/C++ makes a copy of the actual parameter.
- The original variable is not affected by the changes made inside the function.

`void` byValue(`int` a);

• Pass by reference

- Changes made to the parameter will be reflected in the invoker.
- In C: there is no pass by reference; it is simulated with pointers;
- Pointers are passed by value;
- Arrays are passed "by reference".

`void` byPointer(`int` * a);

Variables scope and lifetime I

- **Scope:** the place where a variable was declared determines where it can be accessed from.
- **Local variables**
 - Functions have their own scopes: variables defined inside the function will be visible only in the function, and destroyed after the function call.
 - Loops and if/else statements also have their own scopes.
 - Cannot access variables that are out of scope (compiler will signal an error).
 - A variable lifetime begins when it is declared and ends when it goes out of scope (destroyed).

• Global variables

- Variables defined outside of any function. Can be accessed from any function.
- The scope is the entire application.
- Do not use global variables unless you have a very good reason to do so (usually you can find better alternatives).

Q: What's the best prefix for a global variable?

A: //

Input/Output functions

- read from the command line `scanf` , `cin`:
<http://www.cplusplus.com/reference/cstdio/scanf/>
- write to the console (standard output) `printf` , `cout`
<http://www.cplusplus.com/reference/cstdio/printf/>

Demo

Input-output operations (input_output.cpp)

Summary

- Both C and C++ are compiled languages.
- All C/C++ programs must contain a main function.
- All variables must have types and are recommended to be initialised.
- A variable lifetime begins when it is declared and ends when it goes out of scope (destroyed).
- C/C++ allow the use of pointers.
- Function parameters can be passed by value or by reference (C++).