Object-Oriented Programming

Iuliana Bocicor
iuliana@cs.ubbcluj.ro

Babes-Bolyai University

2017

Overview

C/C++ programming language I

Why C/C++?:

- widely used, both in industry and in education;
- is a high level programming language;
- C++ is a hybrid (multi-paradigm) programming language, implements all the concepts needed for object oriented programming;
- many programming languages are based on C/C++ (Java, C#). Knowing C++ makes learning other programming languages easier.

C/C++ programming language II

Why C/C++:

- C++ is an evolving language;
- C++ is highly standardized;
- C++ gets compiled into processor instructions (no interpretation engine needed).

Integrated Development Environment for C/C++

Microsoft Visual Studio 2015/2013 Community/Professional/Express/Premium

- download from https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx or from Microsoft DreamSpark;
- offers both an IDE and a compiler.

Eclipse CDT

- you need an external C/C++ compiler: MinGW or Cygwin;
- you can find a list of compilers at: https://isocpp.org/get-started
- to install on Windows, see tutorials mentioned in Lab1.

Online IDEs

- you can find a list of online IDEs at: https://isocpp.org/get-started

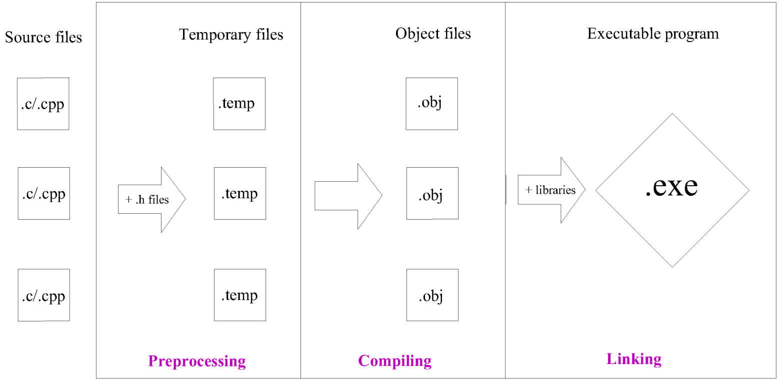
Hello World Demo

DEMO

Hello World! (*HelloWorldC.c, HelloWorld.cpp*).

The compilation process

The compiler translates source code into machine code.



All these steps are performed before you start running a program. This is one of the reasons C/C++ code runs far faster than code in many more recent languages.

Structure of a simple C/C++ program

- Preprocessor directives - e.g. for using libraries;

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

- *main* - special function that is called by the OS to run the program;

```
int main()
{
    //...
    return 0;
}
```

- Every statement must end with a semicolon.

Debugging

- Allows us to step through the code, as it is running;
- Execution can be paused at certain points;
- The effects of individual statements can be seen;
- Allows inspecting the current state of the program (values of variables, call stack);
- Breakpoints - stop the program when reaching the breakpoint.

Lexical elements I

C/C++ is case sensitive.

Identifier:

- Sequence of letters and digits, start with a letter or _ (underline);
- Names of things that are not built into the language;
- E.g.: *i*, *myFunction*, *res*, *_nameOfVariable*.

Keywords (reserved words):

- Identifier with a special purpose;
- Words with special meaning to the compiler;
- E.g.: *int*, *for*, *typedef*, *struct*.

Lexical elements II

Literals:

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

Operators:

- Mathematical: e.g. +, -, *;
- Logical: e.g. !, &&.

Separators:

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

Lexical elements III

Whitespace: Spaces of various sorts, ignored by the compiler: space, tab, new line.

Comments: ignored by the compiler.

```
// This is a single line comment.

/*
This is
a multiline
comment.
*/
```

Data types

A **type** is a domain of values and a set of operations defined on these values.
C/C++ are strongly typed languages.

Fundamental data types in C:

- char (1 byte)
- int (4 bytes)
- unsigned int (4 bytes)
- long int/long (4 bytes)
- float (4 bytes)
- double (8 bytes)
- long double (8 bytes)
- bool (1 byte)

Data types in C++: `https://msdn.microsoft.com/en-us/library/s3f49ktz.aspx`.

Casting

- implicit casting;
- static_cast.

DEMO

Type casting (*Casting.cpp*).

Arrays

If T is an arbitrary basic type:

- T[n] - is an array of length n with elements of type t;
- indexes are from 0 to n-1;
- indexing operator: [ ];
- compare 2 arrays by comparing the elements;
- multidimensional arrays: t[n][m].

DEMO

Arrays (*Arrays.c*).

C String

- Represented as char arrays, the last character is '\0' (marks the end of the string);
- Handled as any ordinary array;
- Standard library for string manipulation in C (*string.h*);
  - strlen - Returns the number of chars in a C string.
  - strcpy - Copies the characters from the source string to the destination string.
    Obs. The assignment operator will not copy the string (or any array).

C String

- Standard library for string manipulation in C (*string.h*);
  - strcmp - Compares two strings and returns: *zero*, if a = b; *negative*, if $a < b$; *positive*, if $a > b$.
    Obs. Using ==, <, >operators on C strings (or any array) compares memory addresses.
  - strcat - Appends the characters from the source string to the end of destination string.

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

DEMO

CStrings (*CStrings.c*).

Record - composite type

- is a collection of items of different types;
- group various data types into a structure.
- declared using struct.

typedef - Introduce a shorthand name for a type.

DEMO

Records (*StructExample.c*).

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 for it and what kinds of operations may be performed on it;
- The value of the variable is undefined until the variable is initialized;
- It is recommended to initialise the variables (with meaningful values) at declaration;
- Use meaningful names for variables.

Constants

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

    ```
    #define LENGTH 10
    const int LENGTH = 10;
    ```
  - *floating*

    ```
    #define PI 3.14
    ```
  - *string literal*

    ```
    const char* pc = "Hello";
    ```
  - *enumeration constant*

    ```
    enum colors {RED, YELLOW, GREEN, BLUE};
    ```

Pointers

- Every variable is a named memory location;
- A pointer is a variable whose value is a memory location (can be 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;
```

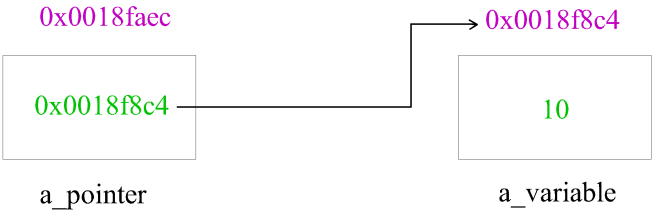
Operators

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

DEMO

Pointers (*Pointers.c*).

Pointers



a_pointer = 0x0018f8c4

&a_pointer = 0x0018faec

*a_pointer = 10

a_variable = 10

&a_variable = 0x0018f8c4

Statements

- A statement is a unit of code that does something - a basic building block of a program.
- Except for the compound statement, in C and C++ every statement is ended by ";".

Statements in C and C++:

- Empty statement;
- Compound statement;
- Conditional statement: *if, if-else, else if, switch-case*;
- Loops: *while, do-while, for*.

DEMO

Statements (*Statements.c*).

Read/Write from/to console

- scanf - read from the command line
  - http://www.cplusplus.com/reference/cstdio/scanf/
- printf - print to the console (standard output)
  - http://www.cplusplus.com/reference/cstdio/printf/

DEMO

Read and Write (*ReadWrite.c*).

Functions

- A function is a group of related instructions (statements) which together perform a particular task. The name of the function is how we refer to these statements.
- The *main* function is the starting point for every C/C++ program.

Declaration (Function prototype)

<result type> name (<parameter list>);

```
/*
Computes the greatest common divisor of two
positive integers.
Input: a, b integers, a, b > 0
Output: returns the the greatest common
divisor of a and b.
*/
int gcd(int a, int b);
```

Functions

Definition

```
< result type> name (< parameter list >)
{
    // statements - the body of the function
}
```

- return <exp> - the result of the function will be the expression value and the function is unconditionally exited.
- A function that returns a result (not void) must include at least one return statement.
- The declaration needs to match the function definition.

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).

Function invocation

<name>(<parameter list>);

- All argument expressions are evaluated before the call is attempted.
- The list of actual parameters need to match the list of formal parameters (types).
- Function declaration needs to occur before invocation.

Variable 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).

Variable scope and lifetime II

Global variables

- Variables defined outside of any function are global variables. 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).

Function parameters I

Pass by value

E.g.

```
void byValue(int a)
```

- 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 change made inside the function.

Function parameters II

Pass by reference

In C:

- there is no *pass by reference*;
- it is simulated with pointers;
- pointers are passed by value.

E.g.

C

```
void byRefC(int* a)
```

C++

```
void byRef(int& a)
```

Function parameters III

- The memory address of the parameter is passed to the function.
- Changes made to the parameter will be reflected in the invoker.
- Arrays are passed "by reference".

DEMO

Functions (*Functions.cpp*).

Test functions

Assert

```
#include <assert.h>
void assert(int expression);
```

- if *expression* is evaluated to 0, a message is written to the standard error device and the execution will stop.
- the message includes: the expression whose assertion failed, the name of the source file, and the line number where it happened.

Function design guidelines

- Single responsability 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 efects (if possible).