Object Oriented Programming I

- **Brief Introduction to Programming**
- **First C++ Programs**
- **Programming Terminology, Basic C++ Syntax**
- **Variables and Types**

Fabio Cannizzo - NUS

Brief Introduction to Programming

Fabio Cannizzo - NUS

What is a Computer?

- A device which performs computations of some kind, with the purpose of generating the solution of a **well defined** problem in correspondence of some **given inputs**

Algorithm

- The detailed set of instruction which performs a task
- Recipe for cooking pasta:
  - Put the water in the pot
  - Add salt
  - Wait until the water boils
  - Put the pasta in
  - Wait 5 minutes
  - Pasta is ready
- How detailed was that? Did we remember to light up the fire? A computer is not smart!

Algorithm

- Algorithm
  - A well-ordered collection of unambiguous and effectively computable operations that, when executed, produces a result and halts in a finite amount of time
- Unambiguous operation
  - An operation that can be understood and carried out directly by the computing agent without needing to be further simplified or explained
- Computing agent
  - The machine, robot, person, or thing carrying out the steps of the algorithm
  - Does not need to understand the concepts or ideas underlying the solution

Languages

- Natural languages are the languages that people speak. Examples of natural languages are English, Italian and German.
  - Natural languages evolved naturally, with usage by people. They are not designed and fixed by someone, even if often people try to impose some order on them.
- Formal languages are languages that are designed by people for specific goals. Examples are:
  - Math language: a formal language used for denoting relationships among numbers and symbols y = sin(x)/2
  - Programming language: a formal language designed to express computations

Languages

- The main difference between formal and natural languages is about ambiguity.
- Natural languages are full of ambiguity.
- Formal languages are designed to be unambiguous: any statement has exactly one meaning.
- Formal languages tend to be much more concise.

Programming Languages

- There are hundreds programming languages, which can be categorized in many different ways based on their characteristics, for example:
  - General Purpose vs Domain Specific
  - Interpreted / Compiled / Partially Compiled
  - Low Level vs High Level
  - Imperative Programming vs Functional Programming
  - Object Oriented
  - Strongly Typed vs Loosely Typed
  - …

What is the Best Language?

- No language is better than another.
- Some languages are better suited to certain task than others.
- Once the concepts are understood, it is not difficult to learn a new language

General Purpose vs Domain Specific

- Languages can be designed for general programming or for a specific task
- The intended use defines the primitives and the syntax available
- If I were to design a language specifically designed to control vocally a phone, probably this would only include the primitives
  - dial *name*, hang up
- C++ is a general purpose language, i.e. can be used to programming “anything”

High Level vs Low Level

- Think about taking a taxi: what’s your style?

1. Bring me to the airport
2. Bring me to the airport via East Coast Road
3. Turn left here, then right, then left, ….then we arrive at the airport

High Level vs Low Level

- Low level: the closer a language it is to machine language
  - Very verbose to describe even the simplest of the operations.
  - Programmer <u>can</u> and <u>must</u> exercise a very detailed control on the sequence of operations to execute
  - Long source code, difficult to read/maintain
  - High degree of optimization possible
  - Very slow development
  - Require explicit control of resource management (e.g. memory allocation)
- High level: closer to natural language
  - Remove control on details of how macro operations will be executed
  - Hide burden of resource management
  - Allow to express complex operations in easy and quick way
  - High productivity
  - Compact source code, easy to maintain
  - Less control on optimization

Interpreted vs Compiled

- Compiled: program is translated into binary code by another program, called compiler. Translation often happens in stages (C++ ➔ ASM ➔ Binary).
- Interpreted: programs are not translated into binary code. The source code is interpreted on the fly (at runtime) by another program, called “interpreter”
- Pseudo-interpreted: it is something in between the two. The program is compiled into pseudo-code, which is not directly executable on a machine, but requires another program (a virtual machine) to be interpreted and executed. This is used typically for portability.

“Classics”: Fortran and C

- Fast and memory sparing
- Very good for scientific purposes
- Many scientific libraries available
- Problem: Few tools for well structured programming (no classes or templates)

Others…

- assembler, pascal, visual basic, java, haskell, dotnet, python, …
- There are hundreds!

C++

- Created in 1979 by Bjarne Stroustrup
- An evolution of C
- Has all the advantages of C and Fortran
- Allows well structured programming
- Is the standard for high performance professional programming
- Scientific C++ libraries exist in abundance

Traits

- Imperative
- Compiled
- Very low level and verbose, but extensive set of libraries available to use it as a more high level language (e.g. boost and STL)
- Strongly and statically typed
- Allow advanced function manipulation both at compile time (e.g. bind) and runtime (e.g. function pointers)

Generating a Program in C++

The Toolchain

- A toolchain is a set of distinct software development tools that are linked (or chained) together by specific stages (e.g. pre-compilation, compilation, linkage, libraries).
- Optionally, a toolchain may contain other tools such as a Debugger.

Compiler of Choice

- We use the Visual Studio Compiler 2015 (Windows OS)
- It is a full GUI based IDE:
  - Editor
  - Pre-compiler
  - Compiler
  - Libraries
  - Linker
  - Debugger
  - Profiler
  - Project Manager

How to Create The Program

- Open Visual Studio
- VS 2015: File -> New -> Project -> Project Types -> Visual C++ -> General
  Console Application
- Select “Empty Project”
- Enter a name for the project
- Specify under “Location” where the project is stored
- Click “OK”

How to Run The Program (continued)

- Click on “Source Files” on the left pane
- Project -> Add New Item
- Choose “C++ File (.cpp)
- Enter a name for the file, click “OK”
- Type some code in the editor window
- Debug -> Start Without Debugging
- If no error, this starts the program

Textfiles

- We use the text editor contained in Visual Studio to write C++ programs
- Other text editors are Notepad, Wordpad, Word, emacs, nano, vim, …
- We can save text as a “text document”
- File name extension often .txt
- Such files are called textfiles
- C++ programs are also saved in textfiles, but typically with file name extension .cpp or .h

C++ Code

- C++ instructions are saved as text files which are called source files and header files.
- The content of these text files is called code (or source or source code)
- Source files usually have file name extension ".cpp", for example program1.cpp
- Header files usually have file name extension ".h", for example header.h
- cpp files are actually compiled, h files are simply imported in cpp files
- The same h file can be imported by many cpp files

Main.cpp

```
// My first C++ program

#include<iostream>             // import library containing cout
using namespace std;           // import namespace

int main()                     // main: start of program
{
    cout << “hello” << endl;   // print hello to screen
    return 0;
}
```

Main.cpp

```
// My first C++ program
```

- Everything which follows // is treated as a comment, i.e. it is ignored by the compiler
- Multiline comments can be specified with

  / this is a*

  *multiline comment */*

Main.cpp

```
#include<iostream>           // import library containing cout
using namespace std;         // import full namespace
```

- Import some libraries
- Libraries identifiers are often qualified with a namespace. We have 3 choices
  - Import full namespace
  - Import only selected symbols from namespace

    using std::cout;
  - Qualify identifiers

    std::cout;

Main.cpp

```
int main()
{
   // body of the program
   return 0;
}
```

- **main** is the first function executed
- every program must have a main
- usually it returns a integer: by convention 0 tells the OS that everything went well
- In this case it has no input arguments, but it could have

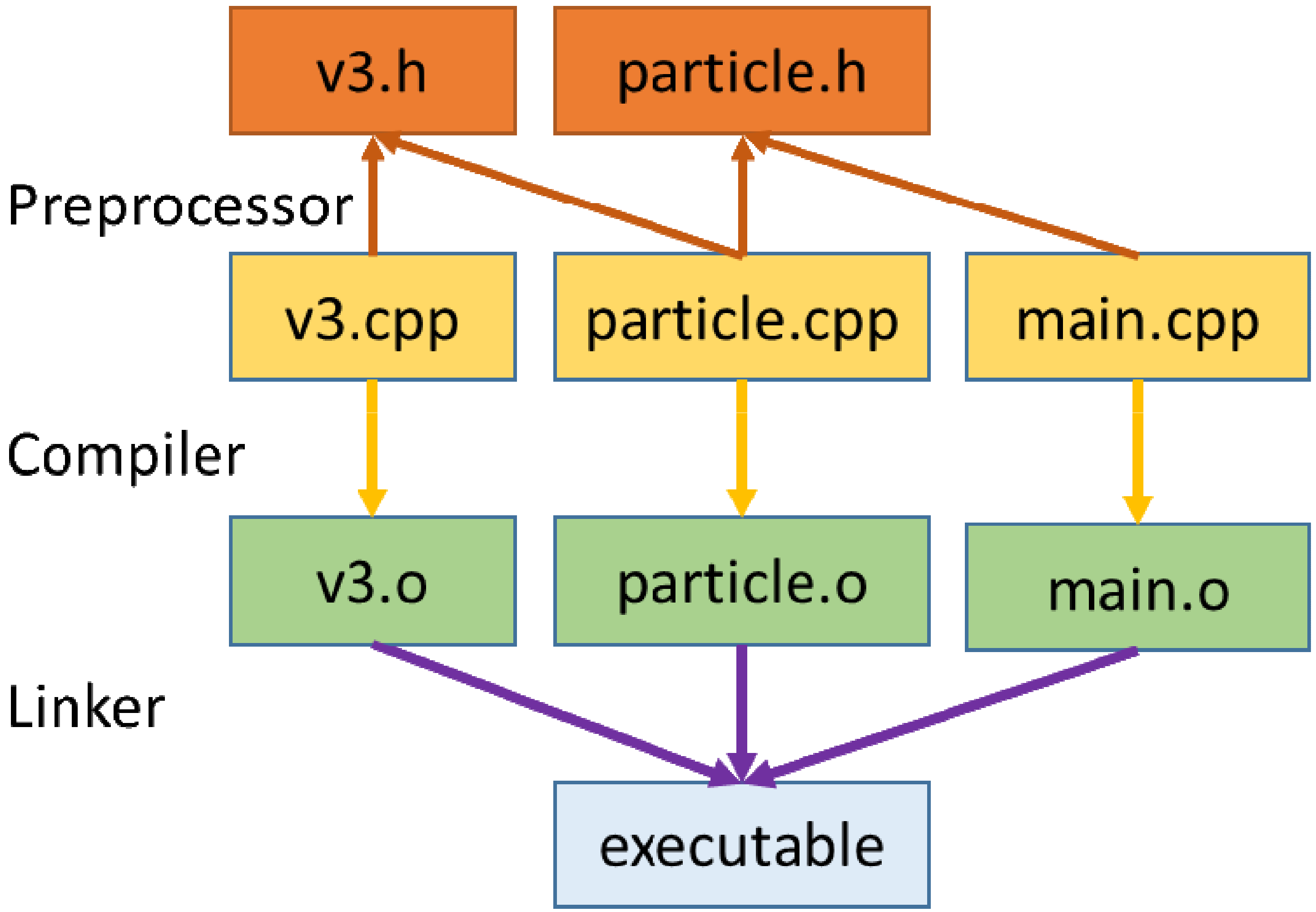
Compiling

- The translation of C++ instruction into machine readable files is called compilation. This is done by a program called compiler.
- The binary files produced by the compiler are called object files.
- Syntax errors in C++ code are detected by the compiler. This is called a compiler error.
- Compiler errors are easy to fix.
- The compiler messages are usually helpful.

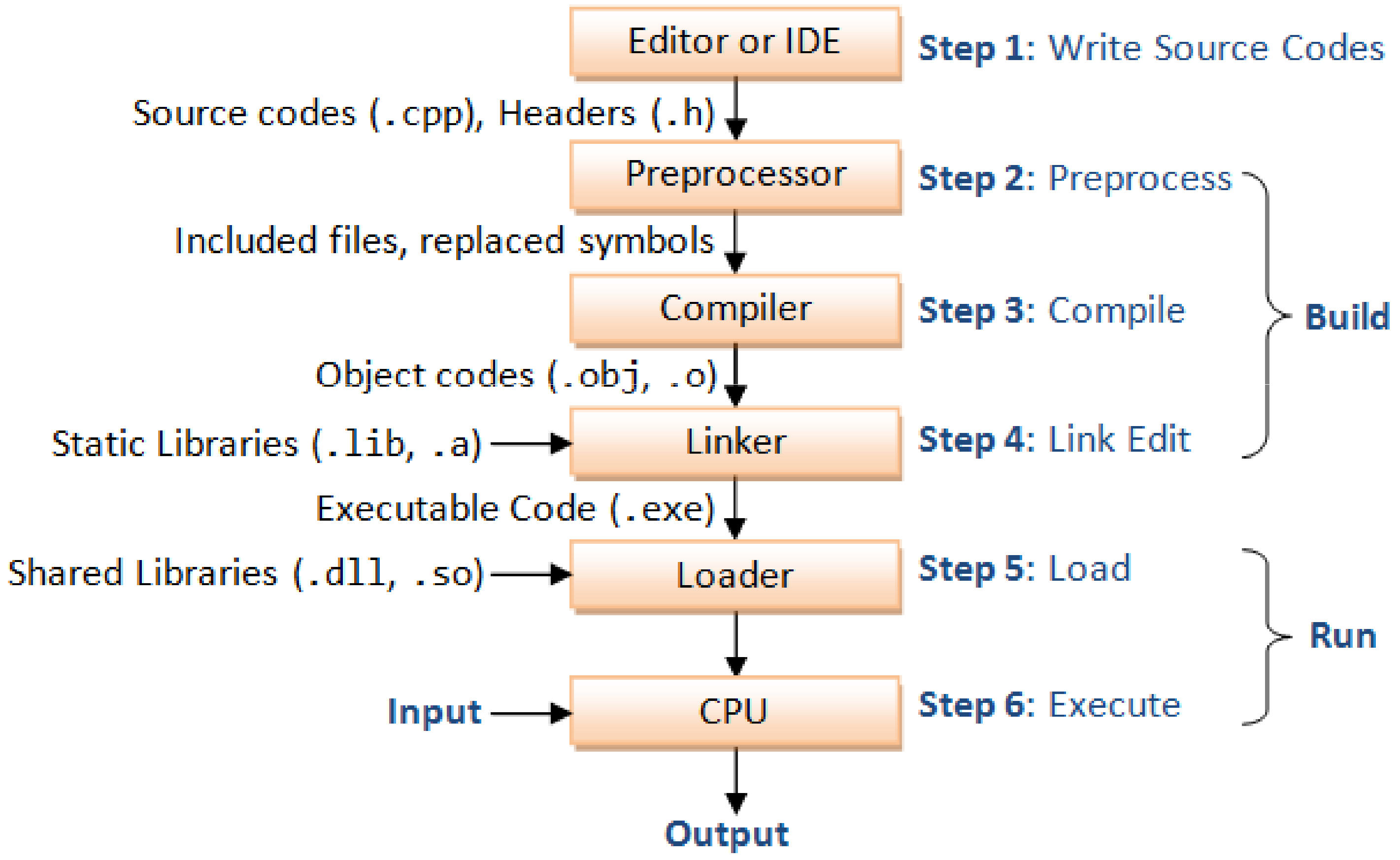
Executable Files / Linker

- An executable file is a complete program, i.e., a computer file that can be run (by double clicking or by calling it from a console)
- Examples: winword, wordpad, excel are executable files under Windows
- C++ source and header files are not executable files
- C++ executable files are produced by a program called linker

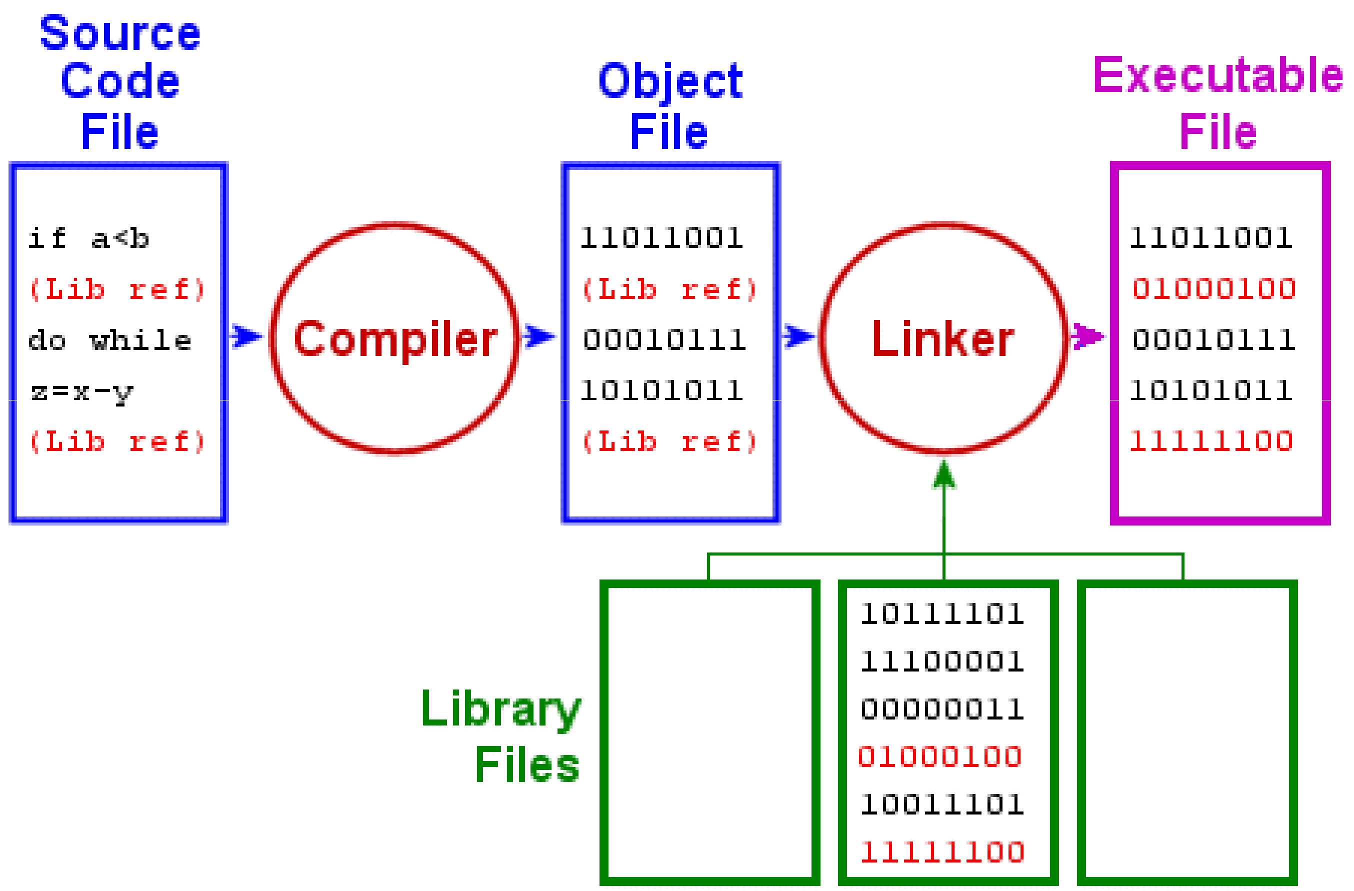
Compilation Process



Compilation Process

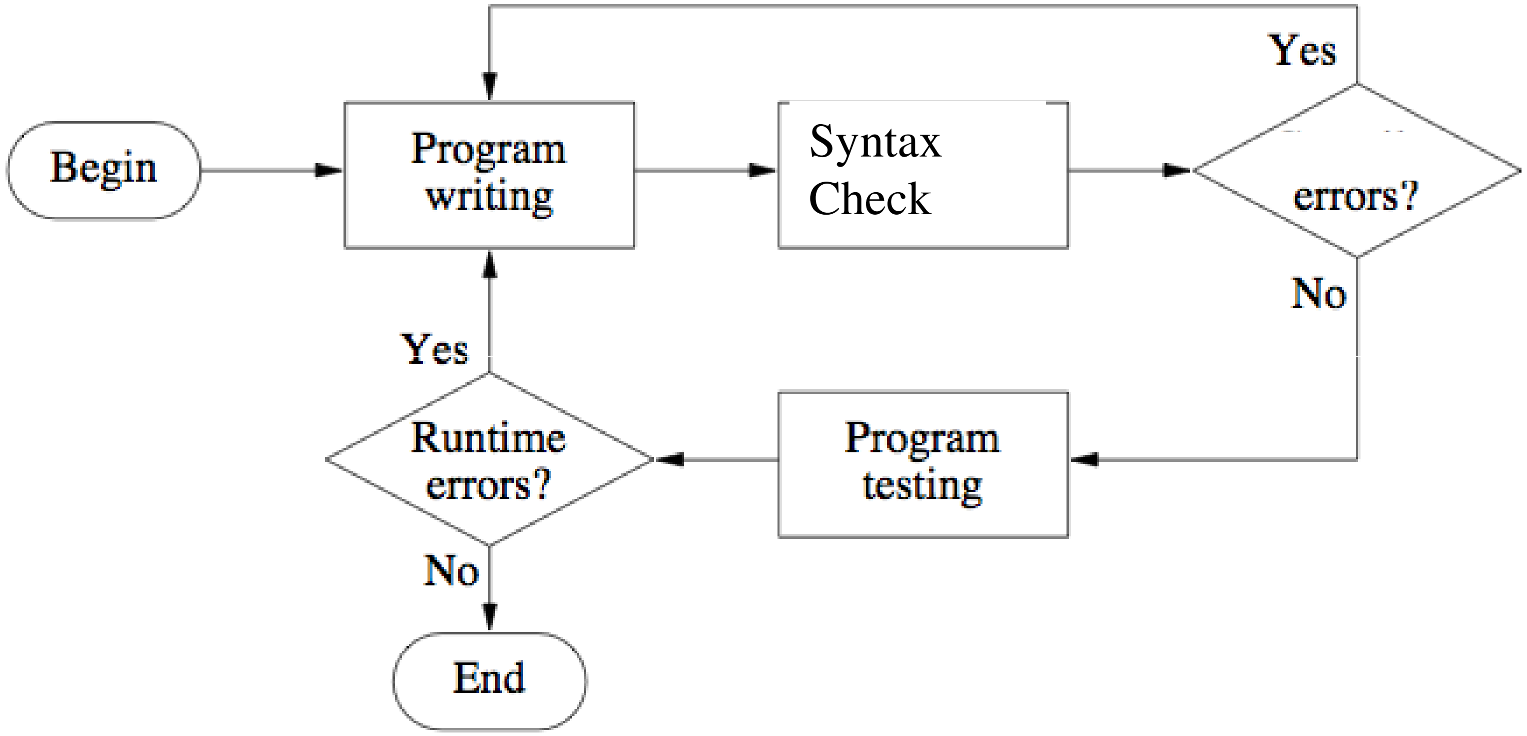


Compilation Process



Errors

Program-Writing Lifecycle



Error Types

- Design Error
  - The solution is conceptually incorrect, i.e. do not address the problem (e.g. Problem: sort ascending an array of numbers, where all numbers except the first two are already sorted. Solution: if x1>x2 the swap x1 and x2. Does this work?)
- Implementation Errors
  - Mistakes in the implementation of the solution (e.g. want sqrt(b^2-4ac), but type sqrt(b*2-4*a*c))
- Runtime Errors
  - At runtime some inadmissible behavior occurs (e.g. forget to check if b2-4ac>0 before taking sqrt())
- Syntax errors
  - Syntactical constraints of the language are violated (e.g. forget to close a parenthesis)

Finding Errors (Debug)

- Syntax error are usually found by the compiler, which will complain. They may be not trivial to fix (e.g. with C++ templates).
- All other error types requires brainstorming (how can I possibly get that figure?) or a proper debug session (execute the program step by step and inspect results at every step)
- Some languages offer a debugger, which simplify the task. If not we can always print out the info we need at every step, to find the place where things go wrong.

Runtime and Logical Errors

- Compiling and linking was successful and we have obtained an executable file. Does this mean that we have a correct program?
- No. There still can be runtime errors or logical errors
- Runtime errors usually are memory problems arising from incorrect use of arrays
- Logical errors occur if wrong methods are used to try to solve a problem

Example of Runtime Error

```
int main()
{
  char *p = 0;
  *p = 34;
  return 0;
}
```

Usually produces runtime error and a crash (depending on compiler settings)

*p is a pointer (will study pointers later)

Finding Errors (Debug)

- Debugging is an art: some bugs can be really time consuming to find and require really skillful strategy to pin them down (e.g. memory leaks in C)
- There are tools to help in the task (e.g. debuggers)
- Easier when coding good practices are followed, i.e. you need to write code well organized, documented, indented and consistent, i.e. you need “style”

Introduction to C++ Syntax

Fabio Cannizzo - NUS

C++ Syntax

Meaning of “syntax”:

The C++ syntax is the set of all rules for writing correct C++ programs, i.e., programs which do not produce compiler or linker errors.

Tokens

- Tokens are the minimals chunk of program that have meaning to the compiler –the smallest meaningful symbols in the language. Our code displays all 6 kinds of tokens

Tokens

Type	Description	Example
Keywords	Words with special meaning to the compiler	int, double, for, return
Identifiers	Names of things that are not built into the language	cout, std, x, myFunction
Literals	Basic constant values whose value is specified directly in the source code	"Hello, world!", 24.3, 0, 'c'
Punctuation / Separators	Punctuation defining the structure of a program	{ } ( ) , ;
Operators	Mathematical or logical operations	+, -, &&, %, <<
Whitespaces	Spaces of various sorts; ignored by the compiler. With the exception of comments, used to delimit tokens	Spaces, tabs, newlines, comments

Keywords

- Special words which are reserved for internal use in the language
  - type names (e.g. ‘int’, ‘double’, ‘bool’, …)
  - qualifiers (e.g. ‘const’, ‘static’, ‘volatile’, …)
  - control flow instructions (e.g. ‘for’, ‘while’, ‘return’, …)
- Keywords cannot be use for variable names or function names

C++ Comments

- A comment is a part of the C++ code that is ignored by the compiler
- After // (double slash) the compiler ignores everything to the end of that line
- The compiler ignores everything between /* and */ (possibly more than 1 line)

  Examples

```
// this is a comment
/*
  this is
  a comment
*/
```

Statements and Statement Blocks

- A **command** is a unit of code that instructs the program ``to do something’’
- Every command must be ended by a semicolon
- A **statement block** is a group of commands enclosed by curly braces
- A **statement** is a single command or a statement block
- An **expression** is a particular type of statement which has a value (e.g. x+3)
- Not all statements are expressions

Parentheses, Square Brackets, Curly Brackets, Angle Brackets

- ( ): parentheses
  - determine order of evaluation of expressions
  - functions arguments
- [ ]: square brackets. For arrays and vectors
- { }: curly brackets/braces. For code blocks and namespaces
- < >: angle brackets. For template parameters

Examples

Command:

```
cout << “hello” << endl;
```

Statement block:

```
{
    cout << “abc” << endl;
     cout << 2*3 << endl;
}
```

Both of these are “statements”

Statements Can be Nested

Example: One statement containing two smaller statements

```
{
     {

          cout << “abc” << endl;
          cout << 2*3 << endl;

     }

     {

          cout << “xxx” << endl;
          cout << 2*5 << endl;

     }
}
```

Variables

C++ Variables: Why?

- C++ programs obtain information from user input, from computer files, or by computation
- Usually the information has to be used later by the same program
- Thus the information needs to be stored
- This is the purpose of variables

Variables

- A variable is a mnemonic label associated with a particular piece of data inside a program and is characterized by the following properties:
  - **Name**
    - Necessary in order to identify the variable.
    - Names must respect some rules
    - Within the rules, it is totally free and up to you to choose a certain name for a variable.
  - **Value**
    - the data stored at the associated memory address at a certain point during the execution of the program
  - **Type**
    - specifies the type of the data stored at the associated memory address. Determines how this information is interpreted (e.g. as an integer, floating point number, string)

Scope

- Variables are only valid, i.e. they only exist, inside the statement block in which they have been created (**scope**)
- We will talk more about scope later

What to do with Variables

- Every variable needs to be created (=declared) first before it can be used
- Moreover, a value has to assigned to a variable (initialization) before it can be used
- The value of a variable can be changed (assignment)
- The value of a variable can be accessed for computations or output

Example

```
{  // begin of a statement
   int x;  // create a variable of type x local to this code block
   x=10; // initialize the value 10 to the variable x
   cout << x << “\n”;  // print the value of x
   x=100; // destructive update: the previous values of x is discarded
   cout << x << “\n”;  // print the value of x
}  // end of a statement
// here the variable x does not exits anymore, i.e. it is out of scope
```

Case Sensitivity and Typos

- C++ is case sensitive: lower and upper case matters
- Example: **test** and **Test** are considered different in C++
- C++ is not only case sensitive, but also extremely “typo sensitive”
- Any typo is a programming mistake!
- Fortunately, the compiler usually finds typos
- Example: `int test=1; cout << Test;` produces compiler error.

Problem

Test what happens if you run the program below.

```
#include<iostream>
using namespace std;

int main()
{
    int x=5;
    cout << X << endl;
}
```

Problem: “Out of Scope”

Test what happens if you run the program below.

```
#include<iostream>
using namespace std;

int main()
{
    {
        int x=5;
    }
    cout << x << endl;
}
```

Compiler error.

Rule: Variables are only valid inside the statement block in which they have been created

Syntax of Variable Declaration

typeName variableName;

Examples

```
 int x;              // type int, name x
double y_34_34;     // type double, name y_34_34
 int 4xy;           // invalid name: start with
 number
 int x$5&;          // invalid name:
 contains &
```

Rule: A variable name can be any sequence of uppercase and lowercase letters, digits, underscores and $, which does not begin with a digit. Usually $ is not used.

Syntax of Variable Assignment

variableName = value;

- This is the way to assign a value to a variable
- The value must fit to the type of the variable
- The first assignment of a value to a variable is called initialization.

> The meaning of “=“ is <u>completely different</u> from that in math: in C++ it means “left side gets a new value which is given on the right side”

Variable Assignment: Examples

```
int x, y ;      // multiple declarations (note we
use the comma)
x=10*10;       // assignment(initialization)
y=x*x;         // assignment(initialization)
y=10*x;        // re-assignment (not initialization)
```

Problem

Write and test a C++ program that does the following:

- Create a variable x of type integer
- Assign the value 10 to x
- Print x to the screen
- Execute the command x = 2*x;
- Print x to the screen
- What is happening?

Combined Declaration and Assignment

typeName variableName = value;

Examples

```
int x =1;

double Pi = 3.1415926;

int z = 2*x, h = 3*z;  // multiple declarations
and initializations (left to right)
```

the const qualifier

const typeName variableName = value;

- The const qualifier can be used to mark a variable as constant, thus avoiding assignment mistakes
- Their value cannot be changed
- const variable must be initialized when declared
- Example:

```
const int x=1;
const int x; x=3;    // compilation error
const int x=1; x=3;  // compilation error
```

cin and cout

Screen Output with cout

- Use of `cout` requires `#include<iostream>`
- The value of a variable `x` is printed to the screen by
  `cout << x;`
- Output can be concatenated, for instance,
  `cout << x << y << endl;`
- A **`string`** is any sequence of symbols enclosed in quotes, e.g
  `“j25j28*(_2423”`
- A ***`string`*** is printed by `cout <<` “***`string`***”`;`

Keyboard input with cin

- Use of `cin` requires `#include<iostream>,` which imports symbols (variables and functions) defined in the iostream library
- The value of a variable `x` is read from the keyboard by `cin >> x;` User input must be ended by pressing `Enter`
- Input can be concatenated, for instance, `cin >> x >> y;`
- Spaces or tabs are only read as separators. 10 spaces have to same effect as 1 space
- `cin` is error-prone since user input can be incorrect.

Problem

Write and test a C++ program that does the following:

- Create variables x, y of type integer
- Read the values of x and y from the keyboard with cin
- Print x+y to the screen

Problem

Write a program that asks the user to type the width and the length of a rectangle and then outputs to the screen the area and the perimeter of that rectangle.

Hints:
Multiplication in C++: “*”
Addition : “+”

Good Practice

- Choose meaningful variable names
- They make the code more readable and reduce mistakes
- A very short variable name (e.g. int i), is ok only if used in a small piece of code.
- Adopt consistently a naming convention, for instance, I use a variation of the camelcase notation, e.g. *thisIsMyVariable*

Fundamental Data Types

Fundamental Data Types

- Now we know how to use variables in principle
- But which *types* of variables are available in C++?
- The fundamental data types are the built-in types like **int**, **double**, **char**
- The specifications of these types may be compiler dependent
- The following specifications are valid for the Visual C++ compiler and several others, including gcc/g++, Dev-C++

Most Commonly Used Types

Type	Range	Comment
`char`	$[-128, 127]$	Characters are stored according to their ASCII-Code, For instance, `char x='A';` is equivalent to `char x=65;`
`int`	$[-2^{31}, 2^{31} - 1]$	integer
`double`	$\pm[2.2e{-308}, 1.79e308]$	Floating point type with a precision of 15 decimal digits. Usually it is advisable to use `double` for all floating point computations. Floating point numbers like 324.343 are automatically interpreted as of type `double`.
`bool`	`true, false`	`true` is identical with 1, `false` with 0. Attention: *All* values $\neq 0$ are converted into `true` by the compiler when interpreted as `bool`.
`void`	-	Return type for function without a return value (must not be ommitted!).

Problem

Write a program that asks the user to type the coordinates of two points, A and B (in a plane), and then outputs the distance between A and B.

Hints:

Use variables of type double.

If x is of type double, its square root can be obtained by sqrt(x).

For this, put **#include<cmath>** at the beginning of the program.

Overflow

- If the values of int or double variables exceed the allowed range, we speak of “integer overflow” or “double overflow”
- In this case, C++ reduces the exceeding values to different values in the allowed range, or, for double, to a special value 'inf'. So the values become incorrect.
- Overflows must be avoided
- Integer overflow is what happened in Problem 4

Problem

Write and test a C++ program that does the following:

- Create a variable x of type integer
- Assign the value 1024*1024*1024 to x
- Print x to the screen (with cout)
- Multiply x by 2 (with x*=2;)
- Print x to the screen
- Any idea what is happening?

Input and Output of bool Variables

- If we “cout” bool variable, we get 0 (for false) or 1 (for true)
- If we “cin” a bool variable, we must use 0 or 1 when we input the value on the keyboard
- If values different from 0,1 are used, the cin command simply has no effect at all

Problem

Write and test a C++ program that does the following:

• Create a variable x of type bool and a variable y of type char

• Read the values of x and y from the keyboard with cin

• Print x and y to the screen with cout

Experiment with different input values for x and y

Number Representation

- Computer represents numbers with a finite number of digits in some base.
- The number of digits available depends the storage space available allocate to a certain number.
- There are some standard types, which have standard size, e.g int32 is an integer stored in a 4 byte word.
- The **base** is the number of 1 digit-numbers available in the numerical system (e.g., in the decimal system there are 10 1-digit numbers 0,1,2,3,4,5,6,7,8,9)
- Most common base representations are binary (2), decimal (10), octale (8) and hexadecimal (16)

Number Representation

- Any integer number can be represented exactly in any base finding the appropriate set of coefficients a_i so that:

$$value = \sum_{i=0}^{n} a_i base^i$$

$$\underbrace{2 \cdot 10^1 + 4 \cdot 10^0}_{decimal: 24} = \underbrace{1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 +}_{binary: 11000} = \underbrace{1 \cdot 16^1 + 8 \cdot 16^0}_{hexdecimal: 18}$$

- From a computer arithmetic point of view, integer operations are exact (i.e. there is no rounding)
- There can be however overflow, if the number is too large to be represented (e.g. *n* is too big)

Example

- Using only 3 bits, we can map to 2^3 integers
- In unsigned format the mapping is from 0 to 7
- In signed format, the mapping is from -4 to 3
- Mappings are chosen to make additions easy

binary	signed	unsigned
000	0	0
001	1	1
010	2	2
011	3	3
100	-4	4
101	-3	5
110	-2	6
111	-1	7

```
2+(-4) = -2        (-2)+(-2) = -4

010 +              110 +
100 =              110 =
--------           --------
110                100
```

⇦ Note the overflow, in this case benign

Computer Arithmetic

- Computer typically represents decimal number in terms of mantissa, base and exponent. Both mantissa and exponent have a sign. The exponent is integer. The base is fixed.

  x = (-1)^sign * mantissa * base ^ exponent

- Suppose we have three digits for the mantissa and two digits for the exponent, and the chosen base is 10, every number must be represented in the format

  x.xx * 10 ^ yy

  Example: 31.89 gets rounded to 3.19 * 10^1

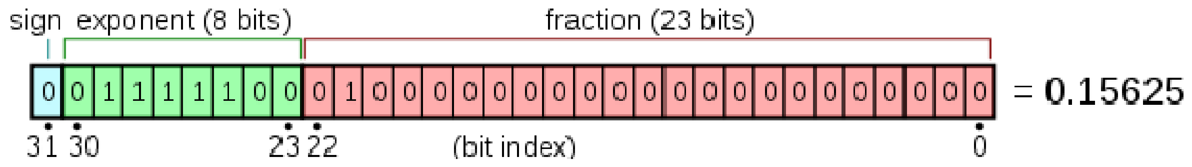
IEEE-754

- A technical standard for floating-point computation established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE).
- Many hardware floating point units use the IEEE 754 standard.
- Includes the definition of the commonly used format binary32 (float) and binary64 (double)

IEEE-754 Binary32 (float)

- Number are represented with 32 bits (4 bytes)
  - Sign bit: 1 bit
  - Exponent width: 8 bits (integer ranging from 0 to 255)
  - Exponent Offset: 127
  - Significand precision: 24 (23 explicitly stored)
  - Some special combinations of bits are used to represent infinity and Not-A-Number
- In most languages this format is referred to as “*float*” or “*single*”

IEEE-754 Binary32 (float)



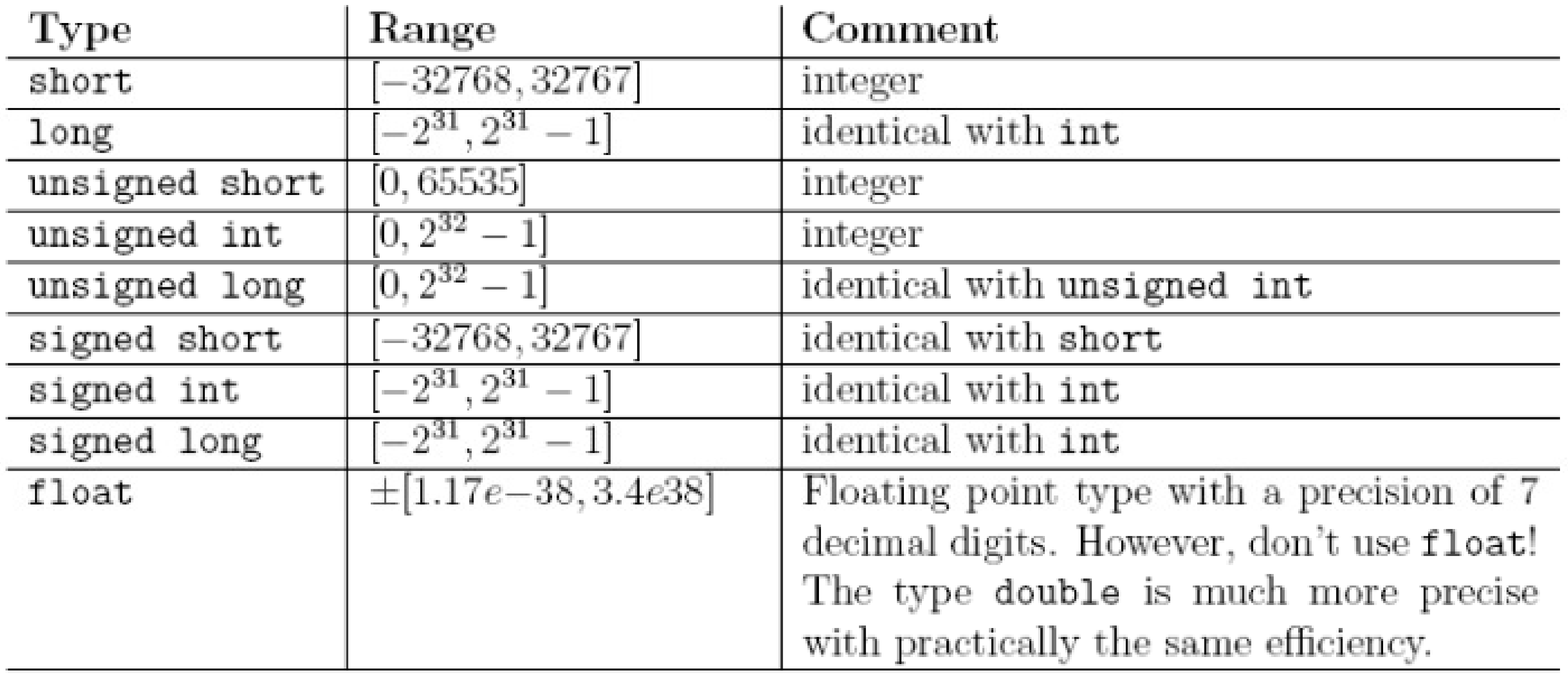
$$value = (-1)^{a_{31}} \left(1 + \sum_{i=1}^{23} a_{23-i} 2^{-i} \right) 2^{\left(\sum_{i=23}^{30} 2^{a_{i-23}} - 127 \right)}$$

- In the example
  - sign = 0
  - exponent: $(2^2+2^3+2^4+2^5+2^6)$ -127=124 -127 = -3
  - mantissa: 1 + 2^{-2}=1.25
  - value = 1.25 2^{-3} = 0.15625

IEEE-754 Binary64 (double)

- Number are represented with 64 bits (8 bytes)
  - Sign bit: 1 bit
  - Exponent width: 11 bits (integer ranging from 0 to 2047)
  - Exponent offset: 1023
  - Significand precision: 53 (52 explicitly stored)
  - Some special combinations of bits are used to represent infinity and Not-A-Number
- In most languages this format is referred to as “*double*”

Other Types and Qualifiers



Type	Range	Comment
`short`	$[-32768, 32767]$	integer
`long`	$[-2^{31}, 2^{31}-1]$	identical with `int`
`unsigned short`	$[0, 65535]$	integer
`unsigned int`	$[0, 2^{32}-1]$	integer
`unsigned long`	$[0, 2^{32}-1]$	identical with `unsigned int`
`signed short`	$[-32768, 32767]$	identical with `short`
`signed int`	$[-2^{31}, 2^{31}-1]$	identical with `int`
`signed long`	$[-2^{31}, 2^{31}-1]$	identical with `int`
`float`	$\pm[1.17e{-38}, 3.4e38]$	Floating point type with a precision of 7 decimal digits. However, don't use `float`! The type `double` is much more precise with practically the same efficiency.

- Keywords like signed unsigned are called *modifiers*

Data Models

Same types are data model dependent:

http://en.cppreference.com/w/cpp/language/types

sizeof

- To know the size in bytes of a variable in memory, we can use the “sizeof” keyword
- It can be used with a type name or with a variable name
- Results depend on the data model
- Size of some types is platform dependent!
- Examples:

```
cout << "size of short int: " << sizeof(short int) <<
endl;
double x;
cout << "size of x: " << sizeof(x) << endl;
```

Sub types

- Note that the domain of some types is strictly contained in other types
- For instance a **short** is contained in an **int**

Sub-Types Up-lifting

- when computing the addition x is automatically converted from short to int and the result is of the expression (y+x) is of type int

```
short x =2;
int y = 3;
int z = y + x;
```

- Example hierarchy:

  bool ➔ char ➔ short ➔ long

enum

- A type of the integer family, convenient to define set of constants with human-friendly names. The syntax is:

  ```
  enum typeName {typeValues = code, . . .}
  ```

- An enum is a subtype of int. It is internally represented as an int.
- For every enum value the compiler assigns an ewuivalent integer
- Optionally, we can make this assignment explicit.

Examples - enum

- Examples of enum definitions:

```
enum Seasons { Spring, Summer, Fall, Winter };
enum WeekDay { Sun=0, Mon=1, Tue=2, Wed=3, Thu=4, Fri=5, Sat=6 };
enum Flags { FLAG1=1, FLAG2=2, FLAG3=4 };
enum { Alpha=1, Beta=2 };  // un-named
```

- Let's try:

```
int main()
{
  enum Seasons { Spring, Summer, Fall, Winter };  // type definition
  Seasons x = Fall;   // variable declaration and initialization
  cout << x << endl;  // print out variable value
  cout << sizeof(Summer) << endl;  // print out variable size
}
```

The **auto** keyword

- A powerful introduction of C++ 11.
- The type of a variable is auto-inferred by the type of the expression assigned to it
- Can only be used with combined declaration and initialization

```
int x = 5;
auto y = x;  // y is of type int
```

typedef

- User can define its own alias to types

typedef typeName typeNameAlias;

Example:

```
// we define uint as an alias for unsigned int
typedef unsigned int uint;
uint x = 3; // x is of type unsigned int
```

typedef

- Pros
  - Can use short names instead of long names (e.g. typing *uint* is shorter than *unsigned int*)
  - Can make refactoring easier: if I define all variable of type *uint*, and then I want to change *uint* from *unsigned int* to *unsigned long*, I just need to change the *typedef*
- Cons
  - Make the program less readable
  - Make refactoring harder if not used consistently

typedef – Example 1

```
typedef double real;
int main()
{
   real x = 5.1;
   real y = 6.2;
  real z = x+y;

  std::cout << z << std::endl;
}
```

- If I change my mind and decide to use **float** instead of **double**, I can change only the ***typedef***

typedef – Example 2

```
typedef double mytype;

int main()

{
    mytype x = 5.1;

   double y = 6.2;

    mytype z = x+y;  // remind me how was mytype defined?


   std::cout << z << std::endl;

}
```

- The chosen name *mytype* is not meaningful. Difficult to remember its declaration!
- I am mixing explicit declaration with *mytype*. If I want to change to *float* I will have to change both.

typedef

- I use it sparingly, to replace very long type names and local to small block statements

```
{
  // this definition is local to this statement
  typedef someVeryLongTypeName myType;
  // ...
}
```

Good Practice

- Always use the most appropriate type for your variable
- E.g. if your variable can only assume positive values in the range 0-40000, then an *unsigned short* is generally the correct type to use.
  - It only takes two bytes, so it reduces memory usage and disk storage space
  - Its domain covers all the necessary range
  - It disallow negative values
  - If memory usage is not a concern, an *unsigned int* would also be an acceptable choice