Lecture #1

Introduction to Programs and C++

chapter 1.1 – 1.3

Objectives

- To understand programs, and operating systems.
- To describe the history of C++.
- To write a simple C++ program for console output.
- To understand the C++ program-development cycle..
- To explain the differences between syntax errors, runtime errors, and logic errors.

Programming Languages

Machine Language   Assembly Language   High-Level Language

Machine language is a set of primitive instructions built into every computer. The instructions are in the form of binary code, so you have to enter binary codes for various instructions. Program with native machine language is a tedious process. Moreover the programs are highly difficult to read and modify. For example, to add two numbers, you might write an instruction in binary like this:

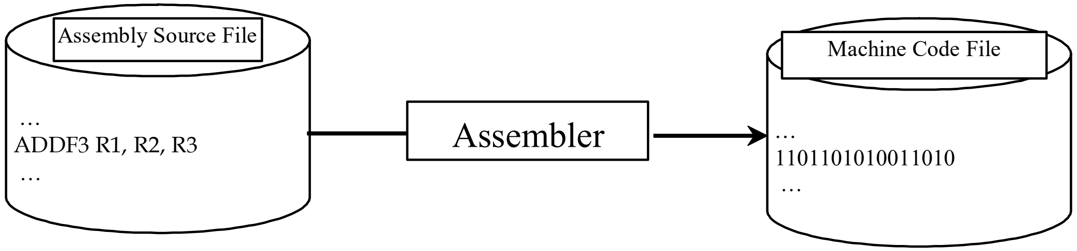
```
1101101010011010
```

Programming Languages

Machine Language Assembly Language High-Level Language

Assembly languages were developed to make programming easy. Since the computer cannot understand assembly language, however, a program called assembler is used to convert assembly language programs into machine code. For example, to add two numbers, you might write an instruction in assembly code like this:

ADDF3 R1, R2, R3



Programming Languages

Machine Language   Assembly Language   High-Level Language

The high-level languages are English-like and easy to learn and program. For example, the following is a high-level language statement that computes the area of a circle with radius 5:

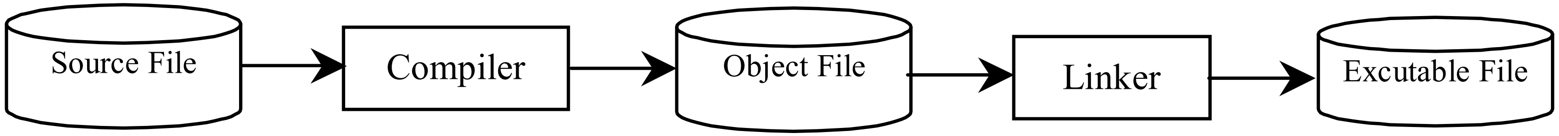
```
area = 5 * 5 * 3.1415;
```

Some High-Level Languages

- BASIC (Beginner All-purpose Symbolic Instructional Code)
- Pascal (named for Blaise Pascal)
- C (whose developer designed B first)
- Visual Basic (Basic-like visual language developed by Microsoft)
- **C++ (an (Hybrid) object-oriented language, based on C)**
- Java (a popular object-oriented language, similar to C++)
- C# (a Java-like developed my Microsoft)

Compiling Source Code

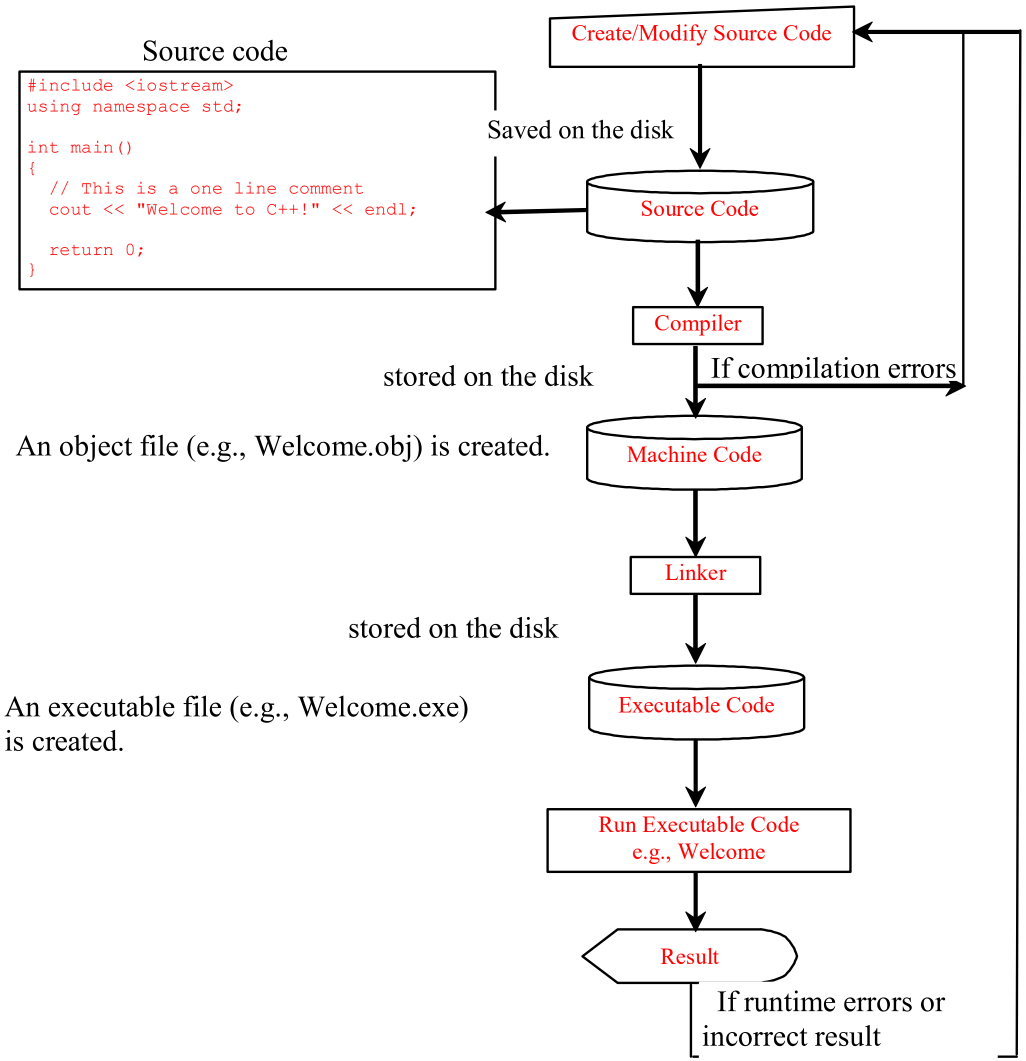
A program written in a high-level language is called a *source program*. Since a computer cannot understand a source program. Program called a *compiler* is used to translate the source program into a machine language program called an *object program*. The object program is often then linked with other supporting library code before the object can be executed on the machine.



C++ IDE Tutorial

You can develop a C++ program from a command window or from an IDE. An IDE is software that provides an *integrated development environment* (*IDE*) for rapidly developing C++ programs. Editing, compiling, building, debugging, and online help are integrated in one graphical user interface. Just enter source code or open an existing file in a window, then click a button, menu item, or function key to compile and run the program. Examples of popular IDEs are Microsoft Visual C++, Dev-C++, Eclipse, and NetBeans, **Visual Code**, etc. All these IDEs can be downloaded free.

Creating, Compiling, and Running Programs



Source code

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

int main()
{
  // This is a one line comment
  cout << "Welcome to C++!" << endl;

  return 0;
}
```

Create/Modify Source Code
Saved on the disk
Source Code
Compiler
If compilation errors
stored on the disk
An object file (e.g., Welcome.obj) is created.
Machine Code
Linker
stored on the disk
An executable file (e.g., Welcome.exe) is created.
Executable Code
Run Executable Code e.g., Welcome
Result
If runtime errors or incorrect result

A Simple C++ Program

Let us begin with a simple C++ program that displays the message “Welcome to C++!” on the console.

Alt. 1:

```
#include <iostream>
using namespace std;
int main()
{
  // This is a one line comment
  cout << "Welcome to C++!" << endl;
  return 0;
}
```

Alt. 2: better

```
#include <iostream>

int main()
{
  // This is a one line comment
  std::cout << "Welcome to C++!" << std::endl;
  return 0;
}
```

Compiling, linking and execution of a program with g++ compiler from Terminal

- Compilation:

```
g++  -c test.cpp    //creates objectfile ”test.o”
```

- Linking and creating executable program named ”test” :

```
g++  test.o  -o test
```

- Exectuting the program: `./test`

  or just: `test`

  iff you have current directory in your path.

Put: **export PATH=“.:$PATH”** in your **.profile** located in your home-directory

Programming Errors

- Syntax Errors
- Runtime Errors
- Logic Errors

Syntax Errors

```
#include <iostream>
int main()
{
  std::cout << (1 + 2 + 3) / 3 << std::endl;
  return 0
}
```

Logic Errors

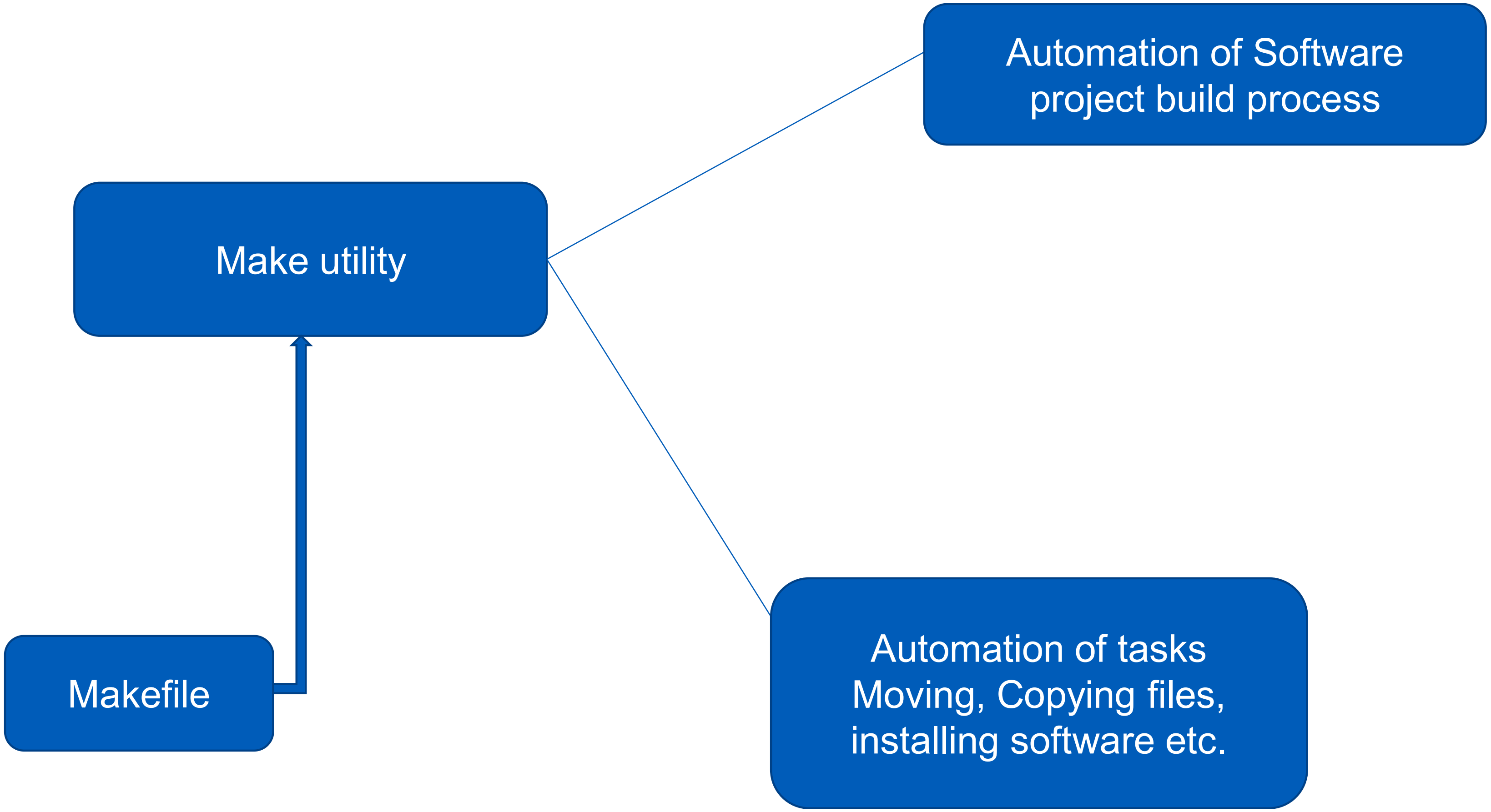
```
#include <iostream>
int main()
{
  std:: cout <<"Adding 3 numbers" <<std::endl;
  std::cout << (1 + 2 - 3) / 3 << std::endl;
  return 0;
}
```

Runtime Errors

```
#include <iostream>

int main()
{
    int b=0;
 std::cout <<"Adding 3 numbers" << std::endl;
 std::cout << (1 + 2 + 3) / b << std::endl;
    return 0;
}
```

make utility and Makefile

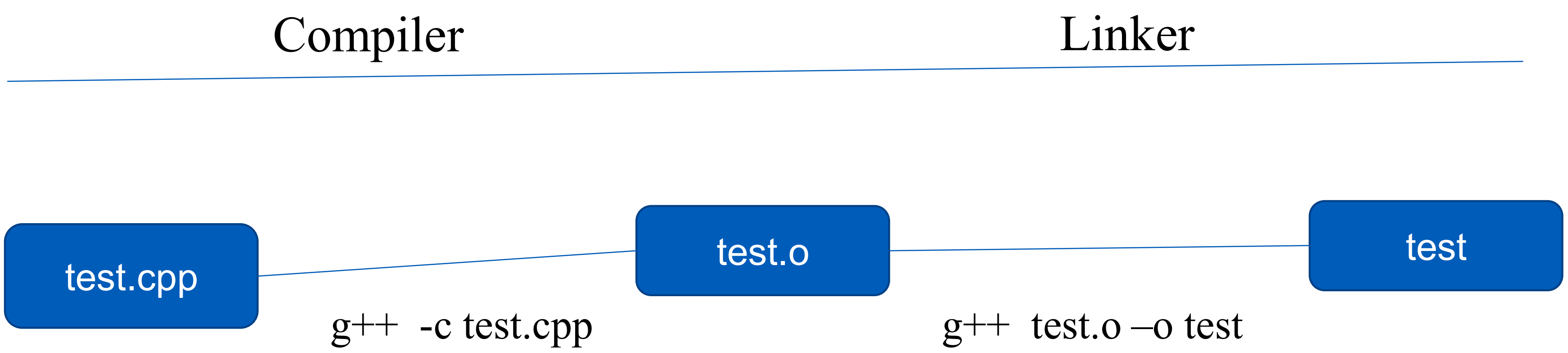

Automation of Software project build process

Make utility

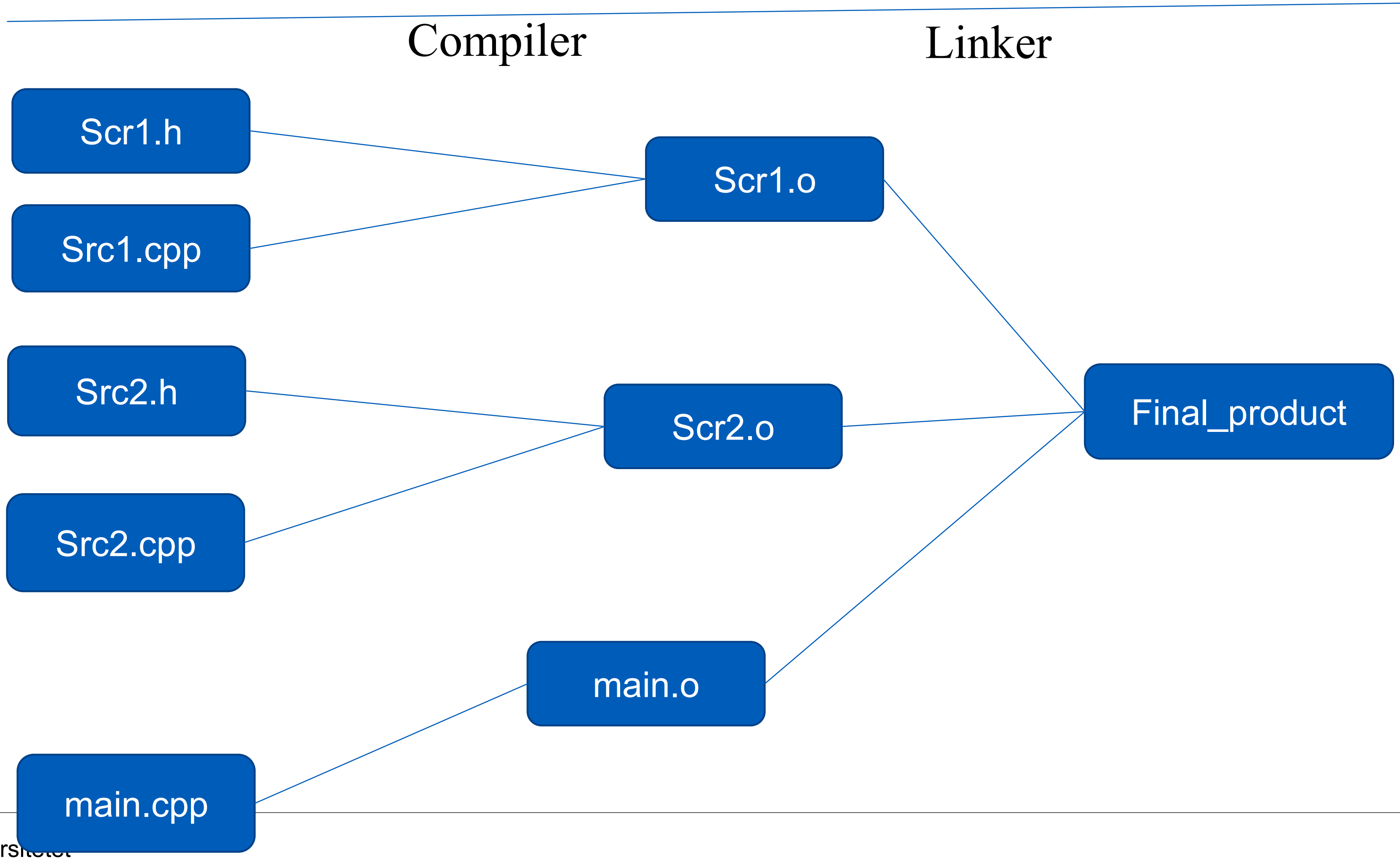
Makefile

Automation of tasks Moving, Copying files, installing software etc.

make utility and Makefile : Build Process



make utility and makefile :Build Process



Makefiles content

Makefiles content

- rules : implicit, explicit
- variables (macros)
- directives (conditionals)
- # sign – comments everything till the end of the line
- \ sign - to separate one command line on two rows

Sample Makefile

- Makefiles main element is called a *rule*:

```
target : dependencies
TAB   commands                    #shell commands
```

Example:

```
test : test.o
  g++ -o test test.o          # -o to specify executable file name
test.o : test.cpp
  g++ -c test.cpp          # -c to compile only (no linking)
```

Using Makefiles

Naming:

- *makefile* or *Makefile* are standard
- other name can be also used

Running `make`

- `make`
- `make -f` *`filename`* – if the name of your file is not "makefile" or "Makefile"
- `make` *`target_name`* – if you want to make a target that is not the first one

Using variables in Makefile

Old way (no variables)

```
test : test.o
       g++ -o test test.o


test.o : test.cpp
       g++ -c test.cpp
```

New way (using variables)

```
C = g++
OBJS = test.o

test : test.o
       $(C) -o test $(OBJS)
test.o : test.cpp
       $(C) -c -Wall test.cpp
```

-Wall turns on some warning flags.

Defining variables on the command line:

Take precedence over variables defined in the makefile. `make C=cc`

Thank you