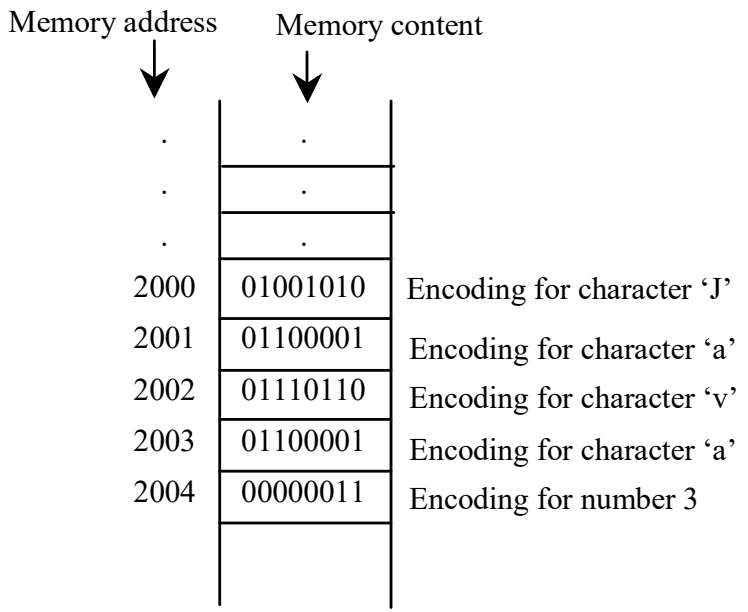
Chapter 1 Introduction to Computers, Programs, and C++

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 know programming style and documentation.
- To explain the differences between syntax errors, runtime errors, and logic errors.

How Data is Stored?

Data of various kinds, such as numbers, characters, and strings, are encoded as a series of bits (zeros and ones). Computers use zeros and ones because digital devices have two stable states, which are referred to as *zero* and *one* by convention. The programmers need not to be concerned about the encoding and decoding of data, which is performed automatically by the system based on the encoding scheme. The encoding scheme varies. For example, character 'J' is represented by 01001010 in one byte. A small number such as three can be stored in a single byte. If computer needs to store a large number that cannot fit into a single byte, it uses a number of adjacent bytes. No two data can share or split a same byte. A byte is the minimum storage unit.



Programs

Computer *programs*, known as *software*, are instructions to the computer.

You tell a computer what to do through programs. Without programs, a computer is an empty machine. Computers do not understand human languages, so you need to use computer languages to communicate with them.

Programs are written using programming languages.

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:

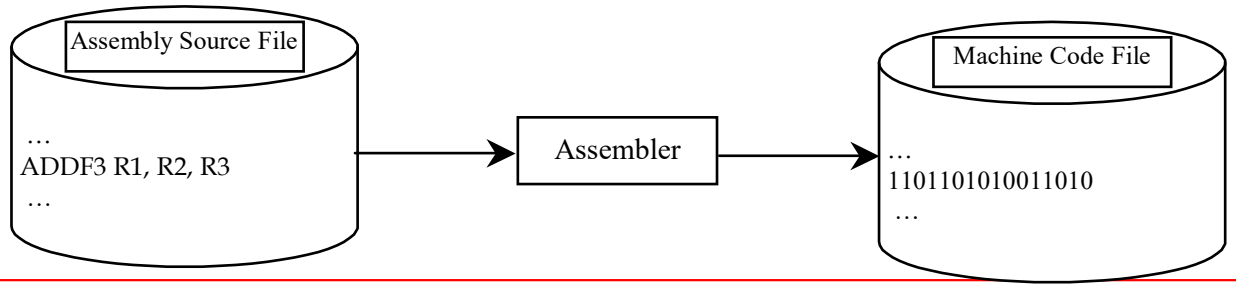
```
1101101010011010
```

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:

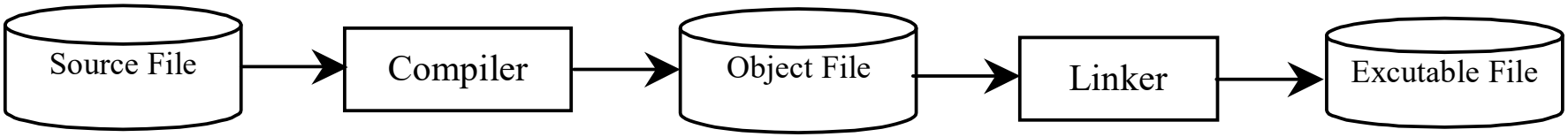
```
area = 5 * 5 * 3.1415;
```

Popular High-Level Languages

- COBOL (COmmon Business Oriented Language)
- FORTRAN (FORmula TRANslation)
- BASIC (Beginner All-purpose Symbolic Instructional Code)
- Pascal (named for Blaise Pascal)
- Ada (named for Ada Lovelace)
- C (whose developer designed B first)
- Visual Basic (Basic-like visual language developed by Microsoft)
- Delphi (Pascal-like visual language developed by Borland)
- **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)

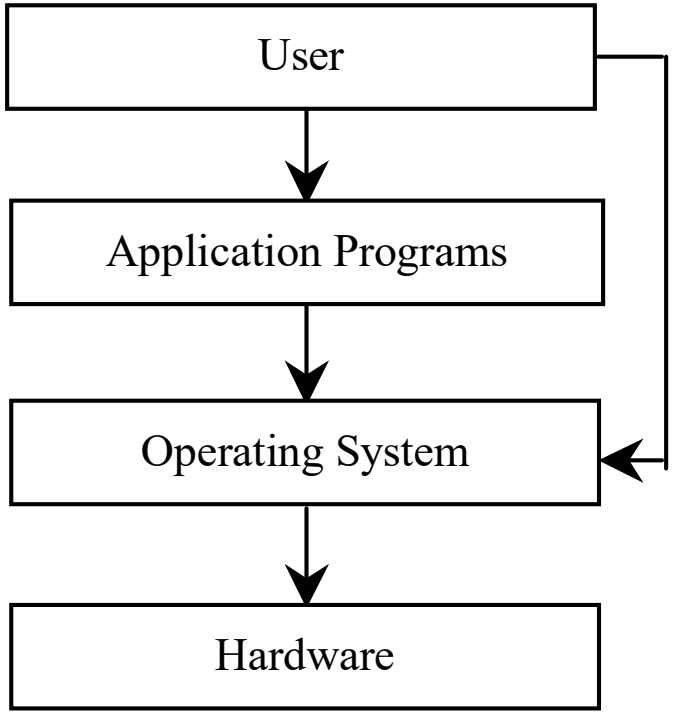
Compiling Source Code

A program written in a high-level language is called a s*ource 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.



Operating Systems

The *operating system* (OS) is a program that manages and controls a computer's activities.. Windows is currently the most popular PC operating system. Application programs such as an Internet browser and a word processor cannot run without an operating system.



History of C++

C, C++, Java, and C# are very similar. C++ evolved from C. Java was modeled after C++. C# is a subset of C++ with some features similar to Java. If you know one of these languages, it is easy to learn the others.

C evolved from the B language and the B language evolved from the BCPL language. BCPL was developed by Martin Richards in the mid-1960s for writing operating systems and compilers.

C++ is an extension of C, developed by Bjarne Stroustrup at Bell Labs during 1983-1985. C++ added a number of features that improved the C language. Most importantly, it added the object-oriented programming.

An international standard for C++ was created by American National Standards Institute (ANSI) in 1998 (C++98). C++11 and later also C++14 was adopted just recently.

A Simple C++ Program

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

```
#include <iostream>
using namespace std;
int main()
{
  // Display Welcome to C++ to the console
  cout << "Welcome to C++!" << endl;
  return 0;
}
```

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.

Extending the Simple C++ Program

Once you understand the program, it is easy to extend it to display more messages. For example, you can rewrite the program to display three messages, as shown in example below.

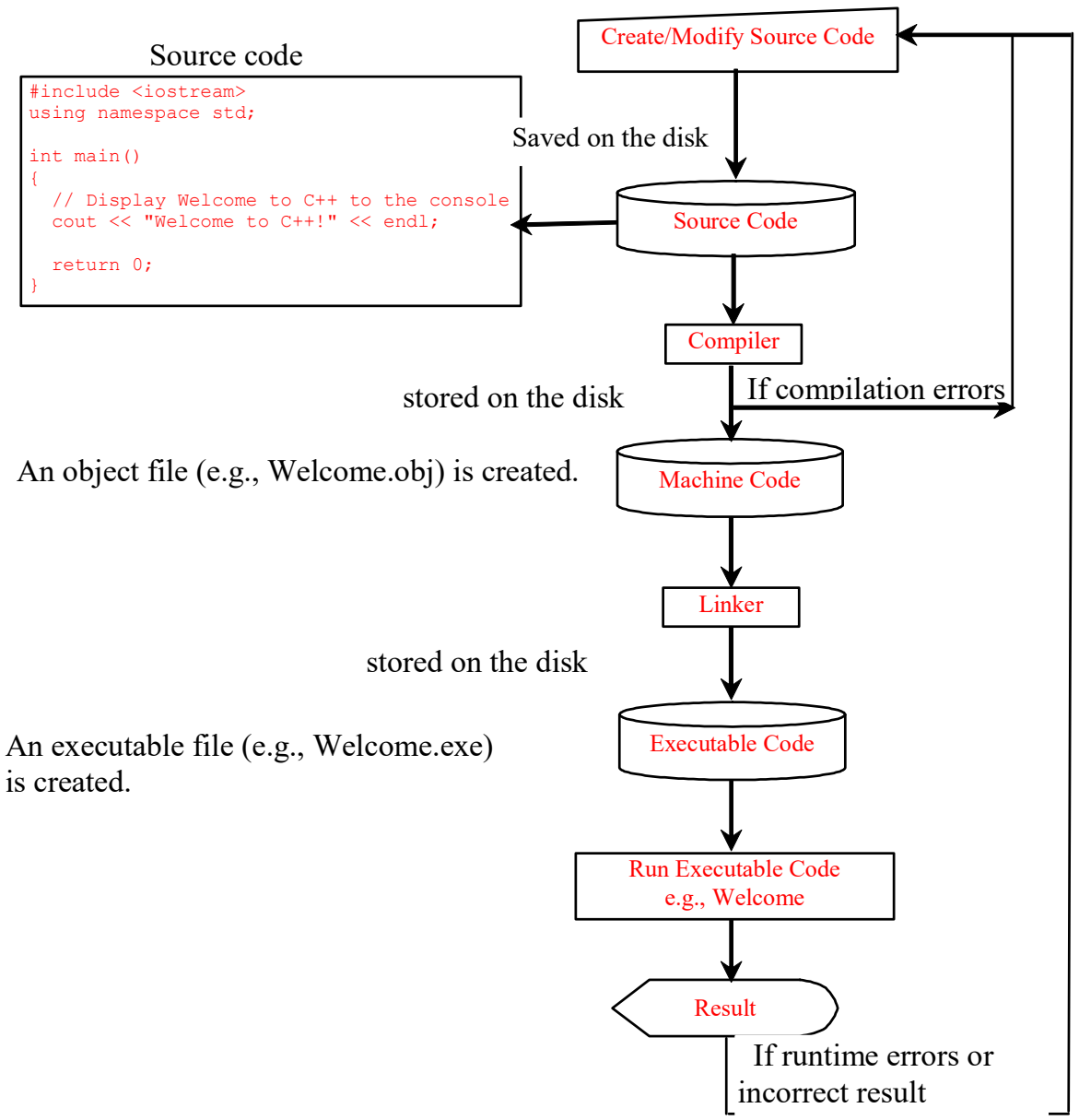
```
#include <iostream>
using namespace std;
int main()
{
  cout << "Programming is fun!" << endl;
  cout << "Fundamentals First" << endl;
  cout << "Problem Driven" << endl;
  return 0;
}
```

Computing with Numbers

Further, you can perform mathematical computations and displays the result to the console. The code below gives such an example.

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

Creating, Compiling, and Running Programs



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 ./test

- All together:

  g++ -o test test.cpp

Programming Style and Documentation

- Appropriate Comments
- Proper Indentation and Spacing Lines
- Block Styles

Programming Errors

- Syntax Errors
- Runtime Errors
- Logic Errors

Syntax Errors

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

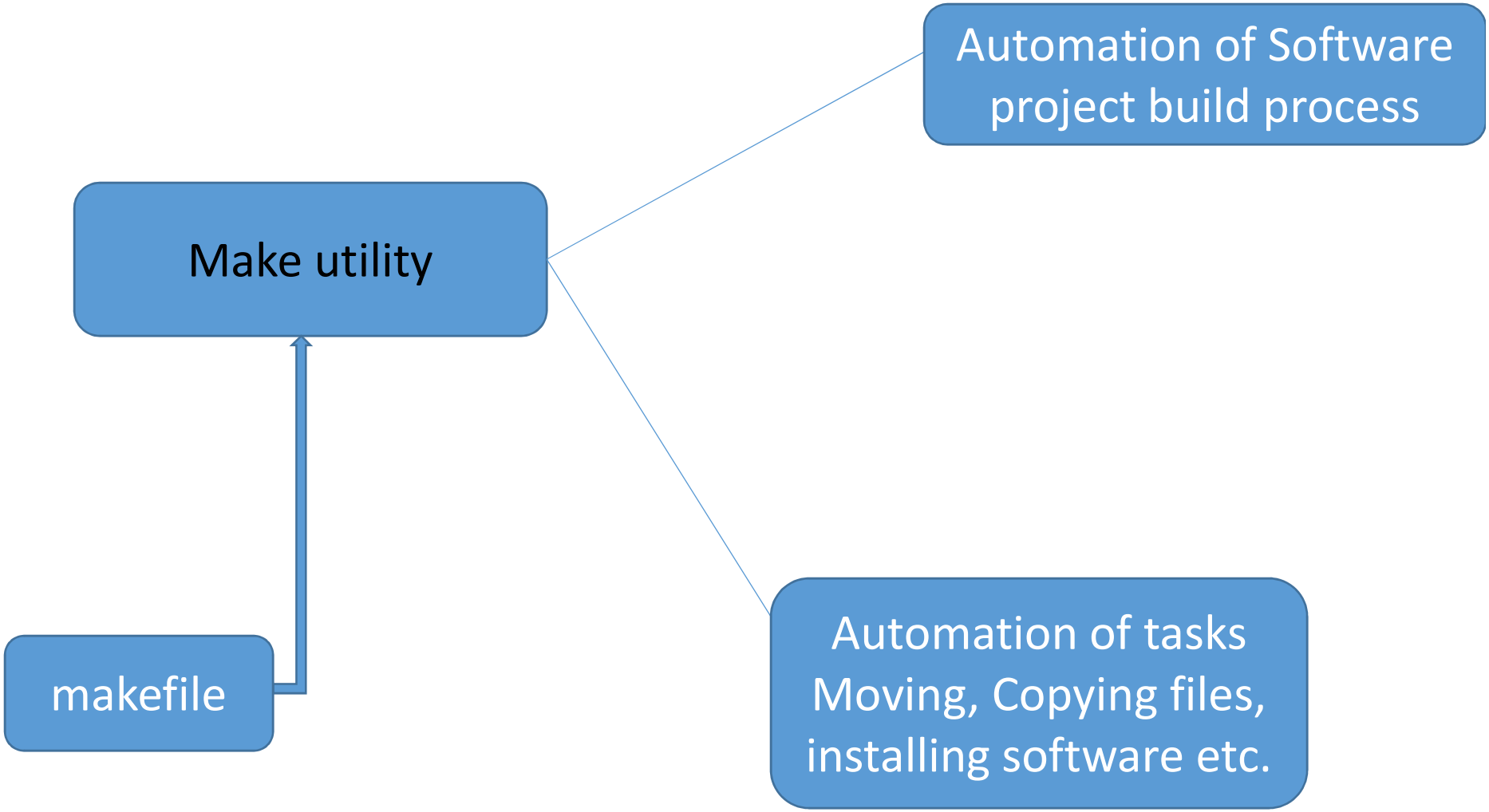
Logic Errors

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

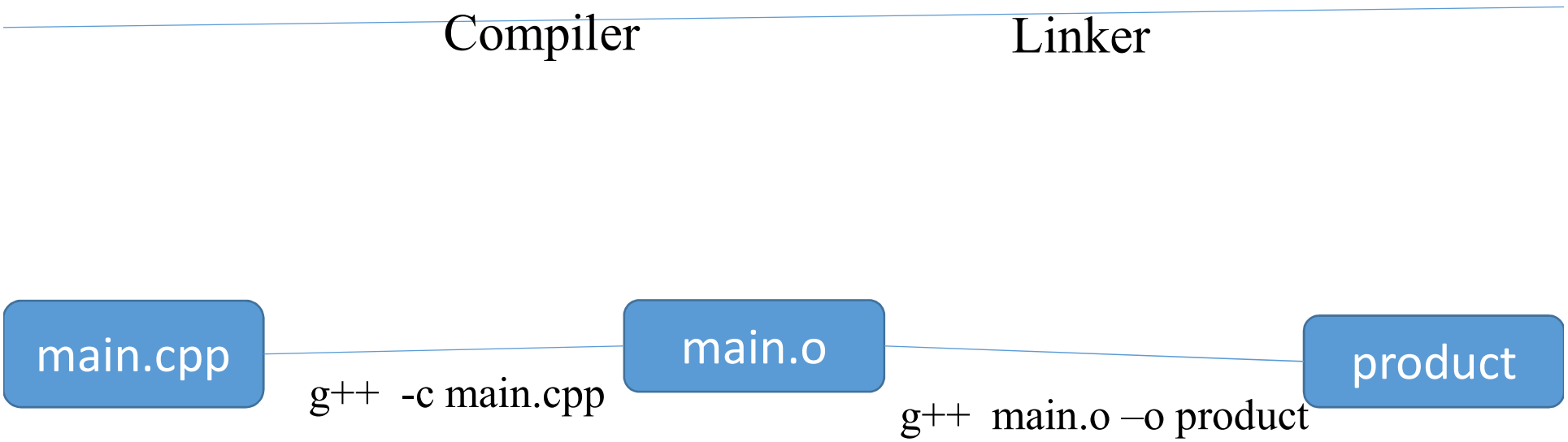
Runtime Errors

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

make utility and makefile

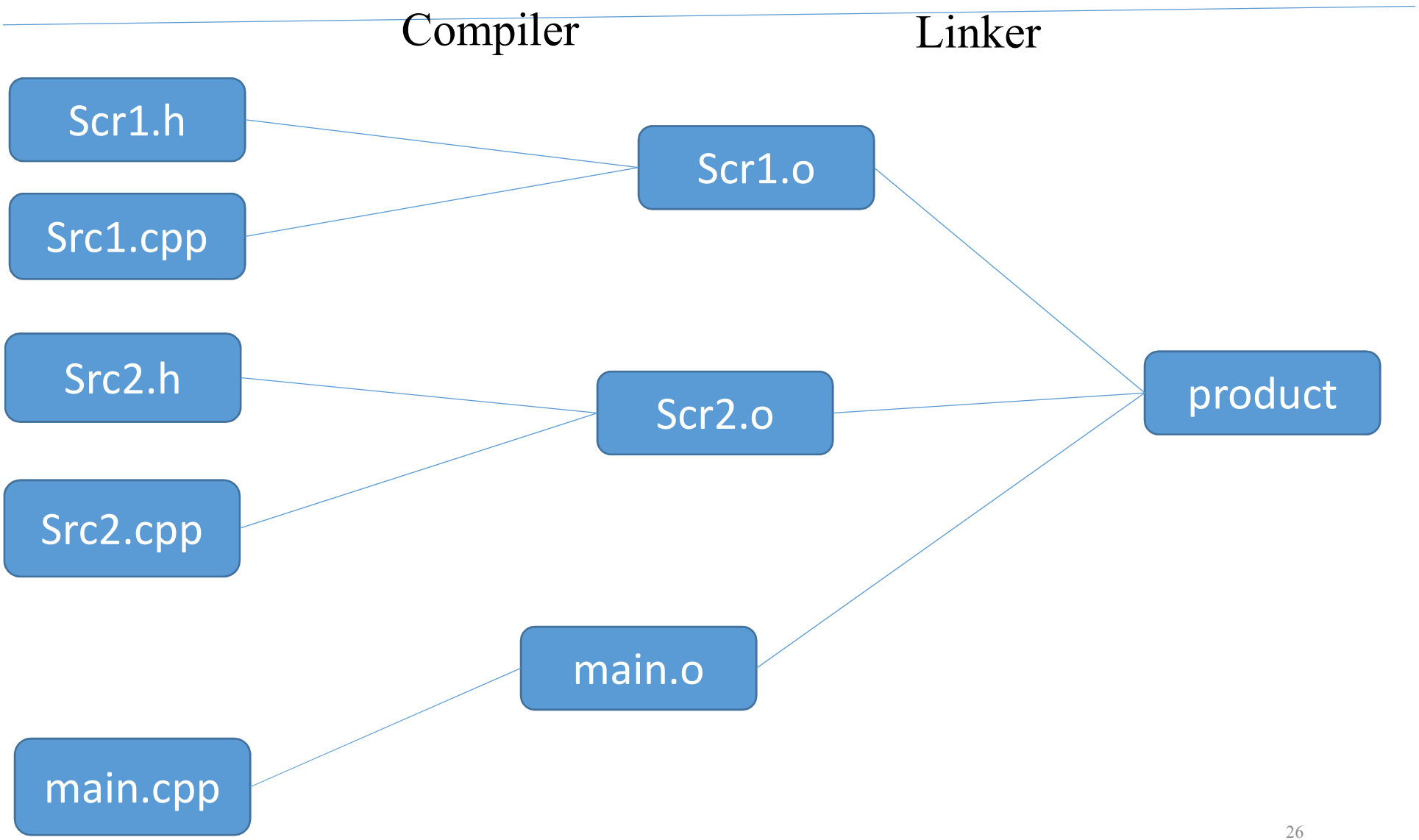


make utility and makefile :Build Process



All together: g++ main.cpp –o product

make utility and makefile :Build Process



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

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

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

_____________________________
# -o to specify executable file name
# -c to compile only (no linking)
```

Variables

The old way (no variables)

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

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

A 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
```

Defining variables on the command line:

Take precedence over variables defined in the makefile.

`make C=cc`

Automatic variables

Automatic variables are used to refer to specific part of rule components.

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

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

`$@` - The name of the target of the rule (`test.o`).

`$<` - The name of the first dependency (`test.cpp`).

`$^` - The names of all the dependencies (`test.cpp and test.h`).

`$?` - The names of all dependencies that are newer than the target

Defining implicit rules

```
%.o : %.cpp   # % is wildcard,meaning for every .o
            #use .cpp  file
 $(C) -c –Wall $<

C = g++
OBJS = test.o

test : test.o
 $(C) -o test $(OBJS)
```

Avoiding implicit rules - empty commands

```
target: ;
```
#Implicit rules will not apply for this target.

`make` options

`make` options:

`-f` *filename* – when the makefile name is not standard

`-t` – (touch) mark the targets as up to date

`-q` – (question) are the targets up to date, exits with 0 if true

`-n` – print the commands to execute but do not execute them

/ `-t`, `-q`, and `-n`, cannot be used together /

`-s` – silent mode

`-k` – keep going – compile all the prerequisites even if not able to link them **!!**

Phony targets

Phony targets:

Targets that have no dependencies. Used only as names for commands that you want to execute.

```
clean :
    rm $(OBJS)
```

or

```
.PHONY : clean
clean:
        rm $(OBJS)
```

To invoke it: `make clean`

Typical phony targets:

`all` – make all the top level targets

```
.PHONY : all
all: test1 test2
```

`clean` – delete all files that are normally created by `make`

`print` – print listing of the source files that have changed

Thank you all