Chapter 1:

INTRODUCTION TO FUNDAMENTALS OF PROGRAMMING

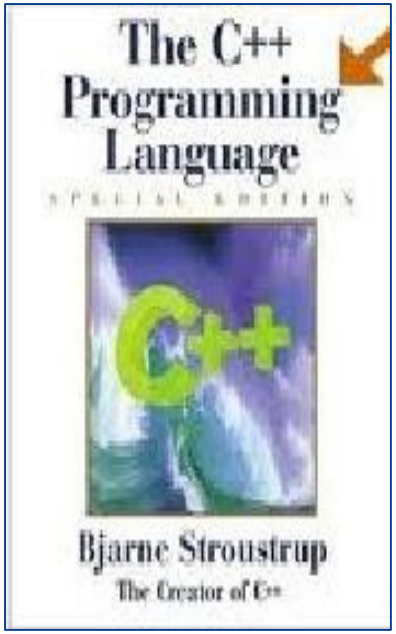
At the end of this sub-chapter, students should be able to:

Define the C++ program basic structure

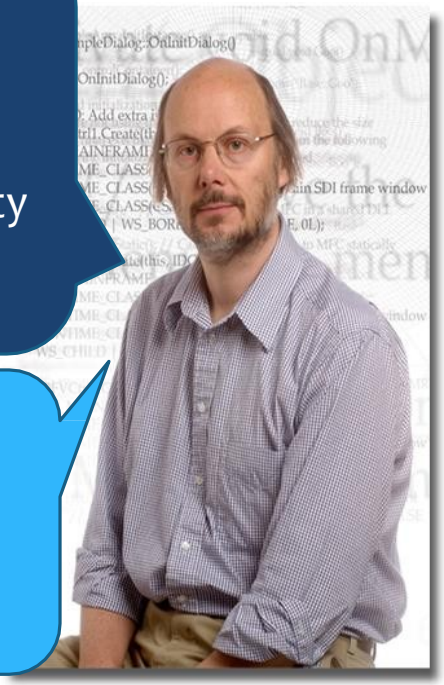
- Describe the item in C++ program structure
- Describe two types of comments that supported by C++ program
- Explain coding standards best practices

The Introduction of C++

C (Early 1970) → C with Class (Before 1983) → C++ (1983)



- C++ was designed for the UNIX system environment
- C++ expanded and enhance version of C that embodies the philosophy of OOP.
- C++ enables programmers to improve the quality of code produced, thus making reusable code easier to write.



- C++'s clarity, extensibility, efficiency and ease of maintenance makes it the language of choice for development of large software projects.
- used widely in the area of communication, personal file systems and databases.

What is a syntax?

* The **syntax** of a computer language is the set of rules that defines the combinations of symbols that are considered to be a correctly structured document or fragment in that language.
* The syntax for every programming language is different.

error C2059: syntax error : '.'
error C2059: syntax error : '}'

Special Characters in C++ programming syntax

Character	Name	Meaning
//	Double slash	Beginning of a comment
#	Pound sign	Beginning of preprocessor directive
< >	Open/close brackets	Enclose filename in #include
( )	Open/close parentheses	Used when naming a function
{ }	Open/close brace	Encloses a group of statements
" "	Open/close quotation marks	Encloses string of characters
;	Semicolon	End of a programming statement

C ++ program structure : comments

- **Comments**

  What is comments?

  - pieces of source code discarded from the code by the compiler

  What uses of comments :

  - only to allow the programmer to insert notes or descriptions embedded within the source code.
  - provide the easiest way to set off specific parts of code
  - providing a visual "split" between various parts of your code.
  - easier to remember what specific parts of your code do.
  - do nothing (do not have any effect on the behavior of the program)

C ++ program structure : comments

* Entries in the source code which are ignored by the compiler
* C++ support 2 style/form of comments:

i. //Single Line Comments
   - for simple 'side' notes
   - best places to put these comments are next to variable declarations, and next to pieces of code that may need explanation.

ii. / *block comment */or /*Multi-Line Comments*/
   - most useful for long explanations of code.
   - useful for two reasons: They make your functions easier to understand, and they make it easier to spot errors in code.

C ++ program structure : comments

* Illustrate the valid & invalid comments within a program

Comment statements	
`// this program displays a message`	valid
`/ this program displays a message`	invalid
`// this comment is invalid because its extend over two lines`	invalid
`// this comment is used a comment that //extend across two lines`	valid
`/*this comment is a block comment that spans three lines*/`	valid

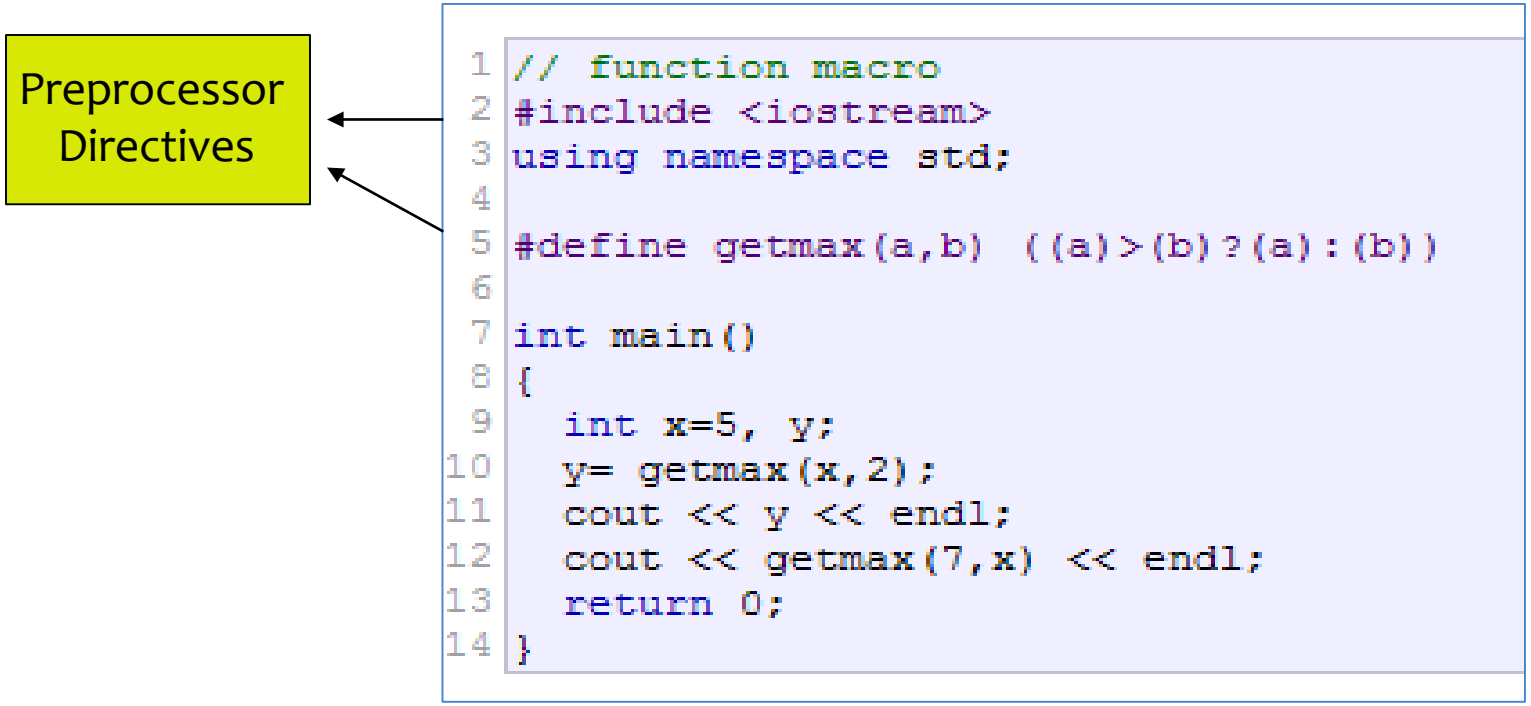
C ++ program structure : Pre processor directives

❑ **Pre processor directives**

* What's a #?
  * **Any line that begins with # symbol is a *pre-processor directive***
  * **Processed by preprocessor before compiling**
* What's pre-processor?
  * **A utility program,** which **processes special instructions** are written in a C/C++ program.
  * Include a library or some special instructions to the compiler about some certain terms used in the program
  * Different preprocessor directives (commands) perform different tasks.
* Uses for?
  * It is a message directly to the compiler.

C ++ program structure : Pre processor directives

* How to write/syntax ?
  * Begin with #
  * No semicolon (;) is expected at the end of a preprocessor directive.
  * Example 1: `#include` and `#define`



C ++ program structure : Pre processor directives

* Example:

pre-processor directive	Meaning
`#include <FILE>`	*Include a header file*
`#include` `<iostream>`	Tells preprocessor to include the input/output stream header file <iostream>
`#define NAME “boB”`	Define a constant
`#ifdef`	Conditional compilation…
`#endif`	

C ++ program structure : Pre processor directives

* **`#define`**
  * Symbolic constants
    * Constants represented as symbols
    * When program compiled, all occurrences replaced
  * Format:
    * **`#define`** ***`identifier replacement-text`***
    * **`#define PI 3.14159`**
  * Everything to right of identifier replaces text
    * **`#define PI=3.14159`**
    * Replaces **`PI`** with **`"=3.14159"`**
    * Probably an error

C ++ program structure : Pre processor directives

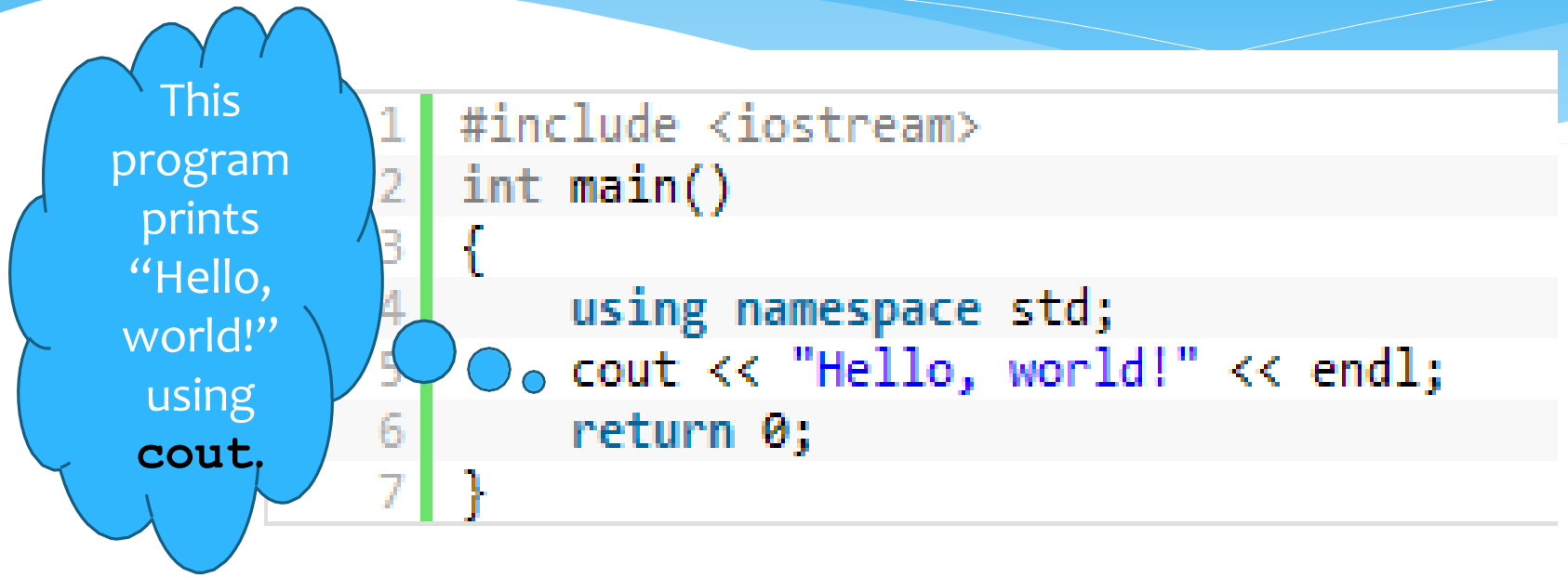
* Cannot redefine symbolic constants
* Advantages of using #define:
  * Takes no memory
* Disadvantages:
  * Name not be seen by debugger (only replacement text)
  * Do not have specific data type
* **`const`** variables preferred.

C ++ program structure : Header files

❑ Header files

* **header file**, sometimes known as an **include file**. Header files almost always have a `.h` extension.
* Why use?
  * For big projects having all the code in one file is impractical. But if we split the project into smaller files, how can we share functions between them? Headers!
- The purpose of a header file?
  - To hold declarations for other files to use.
  - When we use the line `#include <iostream>`, we are telling the compiler to locate and then read all the declarations from a header file named

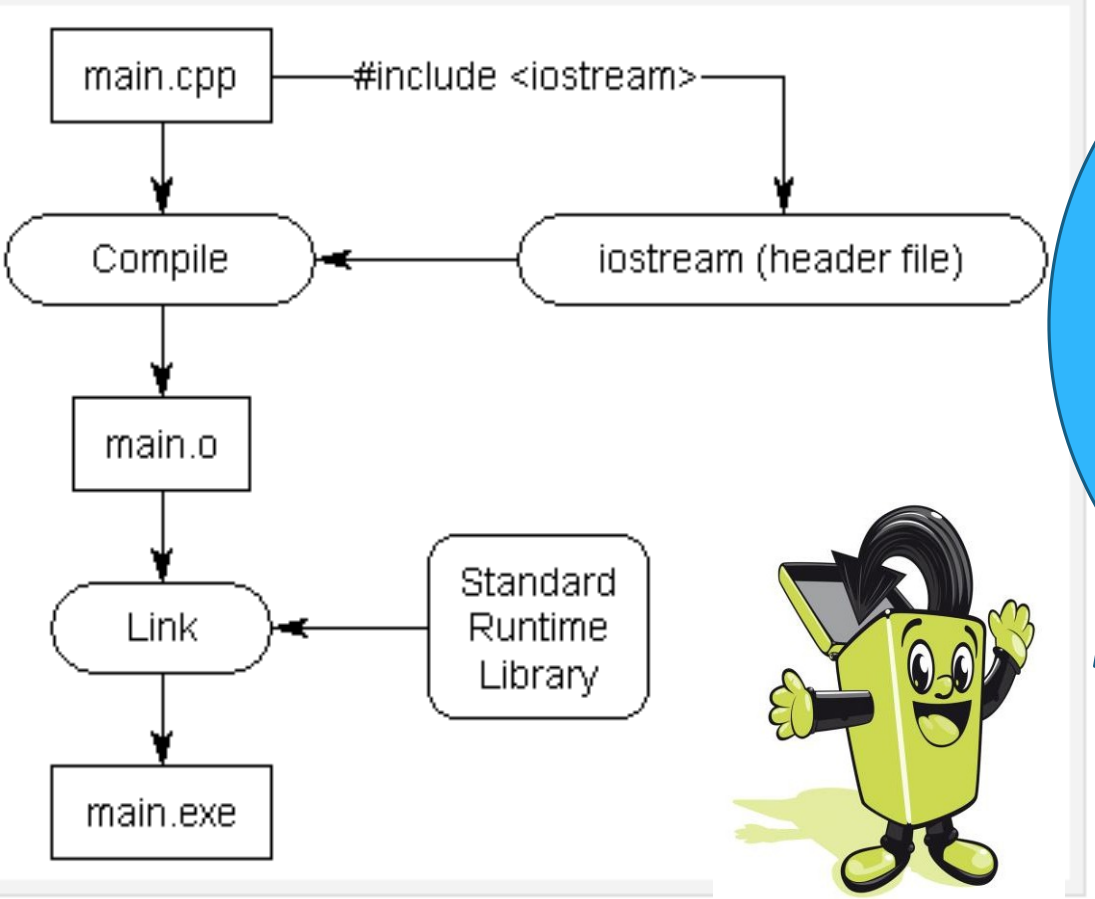
C ++ program structure : Header files



- In program never defines **cout**, so how does the compiler know what **cout** is?
  - The answer is that **cout** has been declared in a header file called "**iostream**".

C ++ program structure : Header files

If cout is only defined in the “iostream” header file,where is it actually implemented?

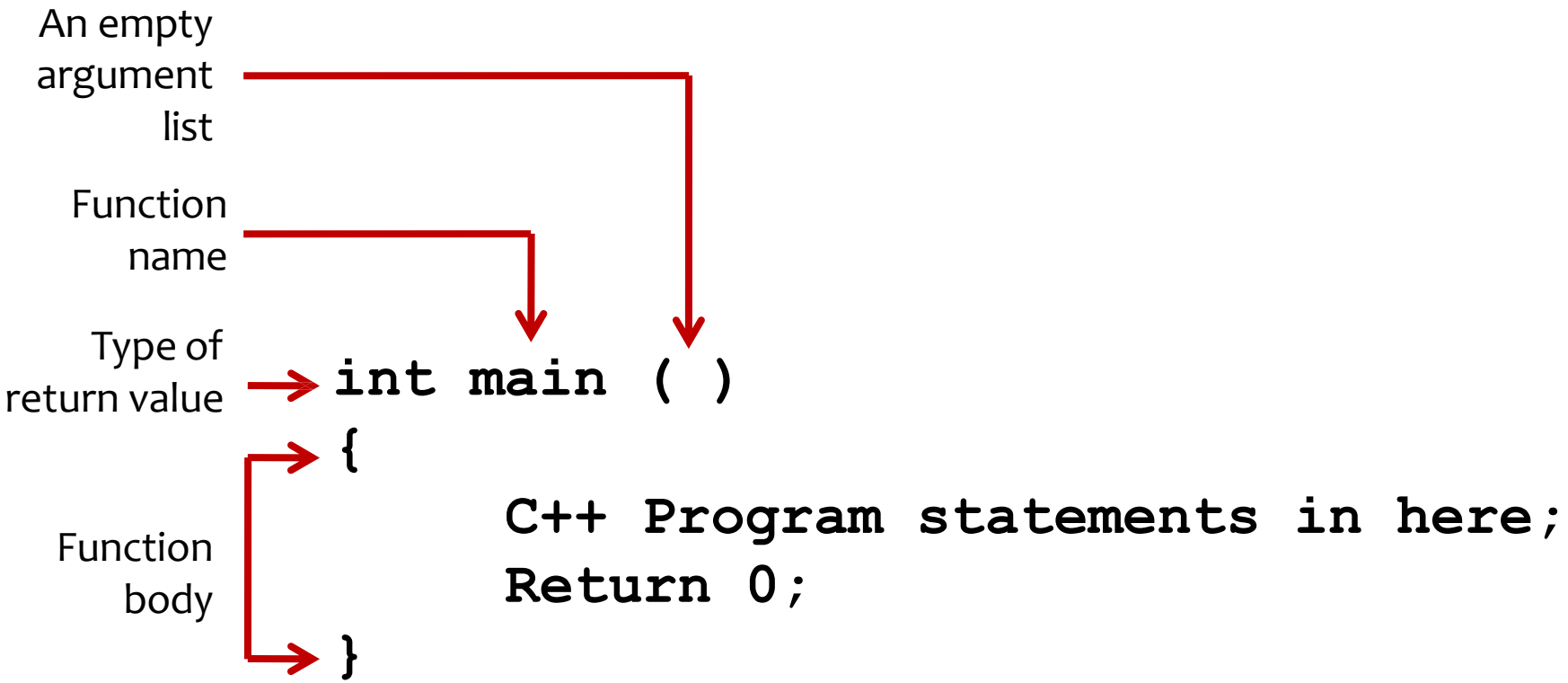


It is implemented in the runtime support library, which is automatically linked into your program during the link phase.

C ++ program structure : Main function

❑ Main function

The structure of a **`main`** Function:



C ++ program structure : Main function

* 4 common ways of main declaration:

```
int main(void)
{

   return 0;
}
```

```
void main(void)
{


}
```

```
main(void)
{


}
```

```
main( )
{


}
```

C ++ program structure : Return statement

- **Return statement**
  - Means?
    - One of several means to exit a function
    - most usual way to terminate a program that has not found any errors during its execution.
    - Give signal that the particular sub-program has finished, and return a value, along with the flow of control, to the program level above.
  - When used?
    - At the **end of `main`**
    - Example: **`return 0;`**

C ++ program structure : Return statement

Two situations for return statement:

i. Return optional in <u>void functions</u>

* A void function doesn't have to have a return statement when the end is reached, it automatically returns.

```
void printChars(char c, int count) {
    for (int i=0; i<count; i++) {
       cout << c;
    }//end for

    return;  // Optional because it's a void function
}//end printChars
```

C ++ program structure : Return statement

Example 2: The max function below requires one or more return statements because it returns an **int** value.

```
// Multiple return statements often increase complexity.
int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}//end max
```

It's generally considered better style to have one return at the end, unless that increases the complexity.

C ++ program structure : Return statement

Example 3: Here is a version of the max function that uses only one return statement by saving the result in a local variable. The use of a single return probably improves the clarity of the max function slightly.

```
// Single return at end often improves readability.
int max(int a, int b) {
    int maxval;
    if (a > b) {
        maxval = a;
    } else {
        maxval = b;
    }
    return maxval;
}//end max
```

Some authors insist on only one return statement at the end of a function. Readable code is much more important than following such a fixed rule.

Coding standard practices

- ❑ Choose meaningful variable and function names
- ❑ Camel case vs snake case
- ❑ Use of comments and whitespace effectively
- ❑ Using indentation and consistent formatting
- ❑ Documentation and communication
- ❑ Avoiding unnecessary loops and iterations
- ❑ Memory management and optimization techniques

At the end of this sub-chapter, students should be able to:

Explain identifier and data types

- Explain identifier, variables and constant
- State the rules for naming an identifier
- Name the variables according the standard
- Explain the data types

Identifier, variable and constant

* **IDENTIFIER** is a words used to represent certain program entities (variables, function names, etc).
* Example:
  * `int my_name;`
    * my_name is an identifier used as a program variable
  * `void CalculateTotal(int value)`
    * CalculateTotal is an identifier used as a function name

Identifier, variable and constant

* **VARIABLE** is identifier whose value can change during the course of execution of a program.
* It is a location in memory which we can refer to by an identifier and which a data value that can be change.
* Common data types (fundamental, primitive or built-in)
  i. int – integer numbers : 1, 2, 4,….
  ii. char – characters : ‘a’, ‘c’, …
  iii. float, double: floating point numbers: 2.5, 4.96

* The value of a variable could be changed while the program is running.
* declaring a variable means specifying both its name and its data type

Identifier, variable and constant

* Example of Variables:
- **`int`** is an abbreviation for integer.
  * Integer store value: 3, 102, 3211, -456, etc.
  * Example variable : number_of_bars

- **`double`** represents numbers with a fractional component
  * double store value: 1.34, 4.0, -345.6, etc.
  * Example variable : one_weight ,total_weight

Identifier, variable and constant

* **CONSTANTS** are values that do not change during program execution. They can be any type of integer, character or floating-point.
* Entities that appear in the program code as fixed values.
* Any attempt to modify a CONSTANT will result in error.
* Declared using the `const` qualifier.
* Also called named constants or read-only variables.
* Must be initialized with a constant expression when they are declared and cannot be modified thereafter.
* Example: `const int size = 5;`

Rules for naming identifier

* They are formed by combining letters, digits & underscores.
* Blank space is not allowed in an identifier.
* The first character of an identifier must be a letter.

valid	invalid	Reason
Monthly_Salary	Monthly Salary	Blank space cannot be used
Month1	1stMonth	Digit cannot be used as a first character
Email_add	email@	Special characters cannot be used

Rules for naming identifier

Rules	Example
Can contain a mix of character and numbers. However it cannot start with a number	H2o
First character must be a letter or underscore	Number1, _area
Can be of mixed cases including underscore character	XsquAre my_num
Cannot contain any arithmetic operators	R*S+T
… or any other punctuation marks	#@x%!!
Cannot be a C keyword/reserved word	struct; cout;
Cannot contain a space	My height
… identifiers are case sensitive	Tax != tax

Name the variables according the standard

Name the variables according the standard

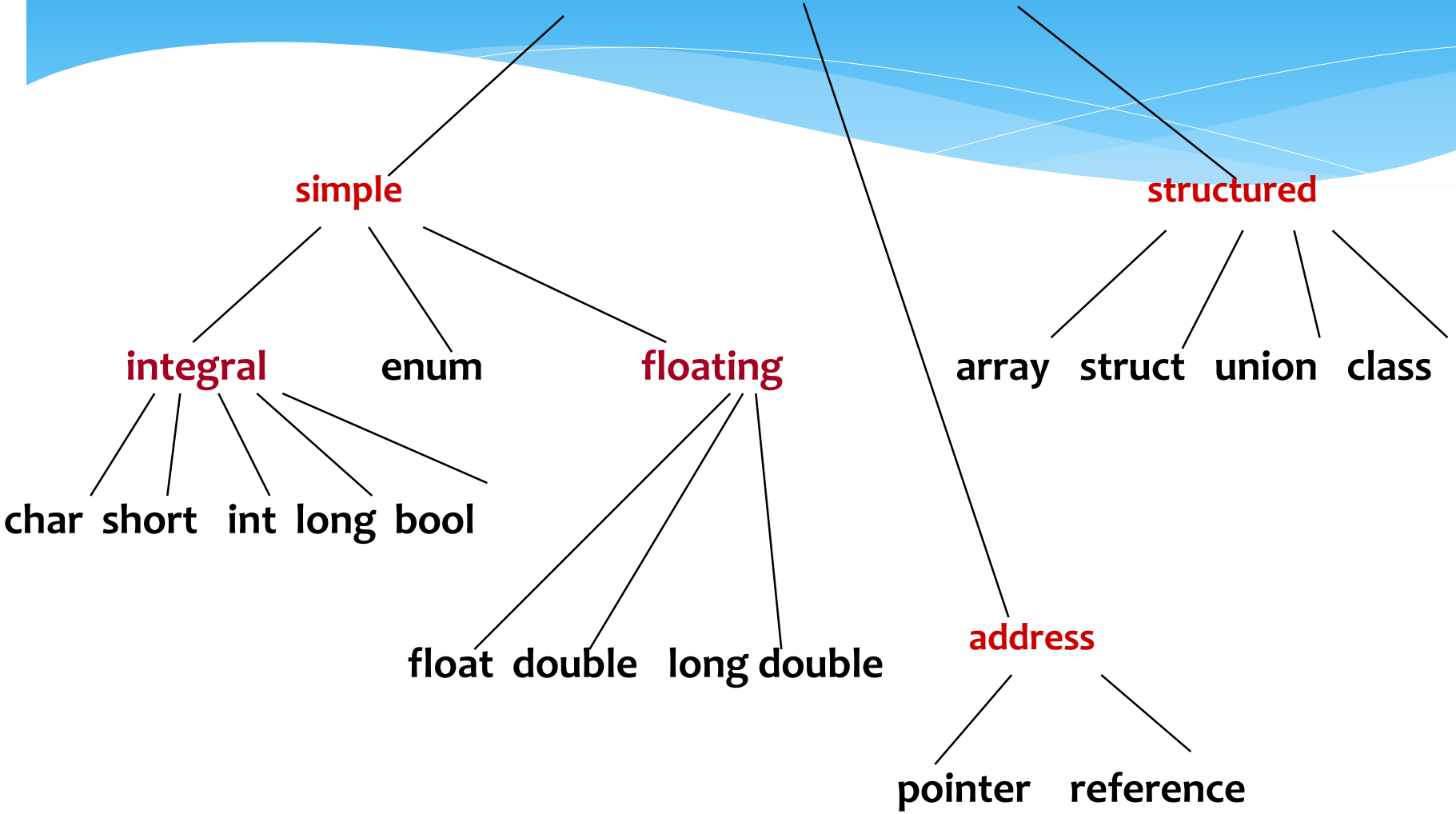
Data Types

Classify data types**:**

i. **Explain 6 basic data types:**
   a. **char**
   b. **int**
   c. **Double**
   d. **float**
   e. **Bool**
   f. **string**

ii. **Design, implement, test and debug program using data types**

Data Types


simple
structured
integral
enum
floating
array
struct
union
class
char
short
int
long
bool
float
double
long double
address
pointer
reference

Data Types

* There are 6 basic ***data types*** :
  i. **char**
  * equivalent to 'letters' in English language
  * Example of characters:
    * Numeric digits: 0 - 9
    * Lowercase/uppercase letters: a - z and A - Z
    * Space (blank)
    * Special characters: , . ; ? " / ( ) [ ] { } * & % ^ < > etc
  * single character
  * keyword: char
  * Example code:

```
char my_letter;
my_letter = 'U';
```

  * sample values

    'B' 'd' '4' '?' '*'

The declared character must be enclosed within a single quote!

Data Types

ii. int

* used to declare numeric program variables of integer type
* whole numbers, positive and negative
* keyword: `int`
* Example code :

```
int number;
number = 12;
```

* Sample values:

4578 **-4578** **0**

Data Types

iii. **double**

* used to declare floating point variable of higher precision or higher range of numbers
* exponential numbers, positive and negative
* keyword: `double`
* Code example:

```
double valuebig;
valuebig = 12E-3;
```

Data Types

iv. **float**

* fractional parts, positive and negative
* **real numbers with a decimal point**
* keyword: `float`
* Example code:

```
float height;
height = 1.72;
```

* sample values

95.274 **95.** **.265**

Data Types

v. **Bool** (Short for boolean)

* bool is a new addition to C++
* Boolean values are either true or false
* To declare a variable of type bool:

```
bool old_enough;
```

Data Types

vi. String

* a string is a sequence of characters enclosed in double quotes

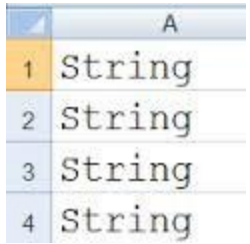
* **string sample values**

  `"Hello"  "Year 2000"  "1234"`

* **the empty string (null string) contains no characters and is written as " "**

Data Types

* More About Type String:
* **string is not a built-in (standard) type**
  * **it is a programmer-defined data type**
  * **it is provided in the C++ standard library**
* **string operations include**
  * **comparing 2 string values**
  * **searching a string for a particular character**
  * **joining one string to another**



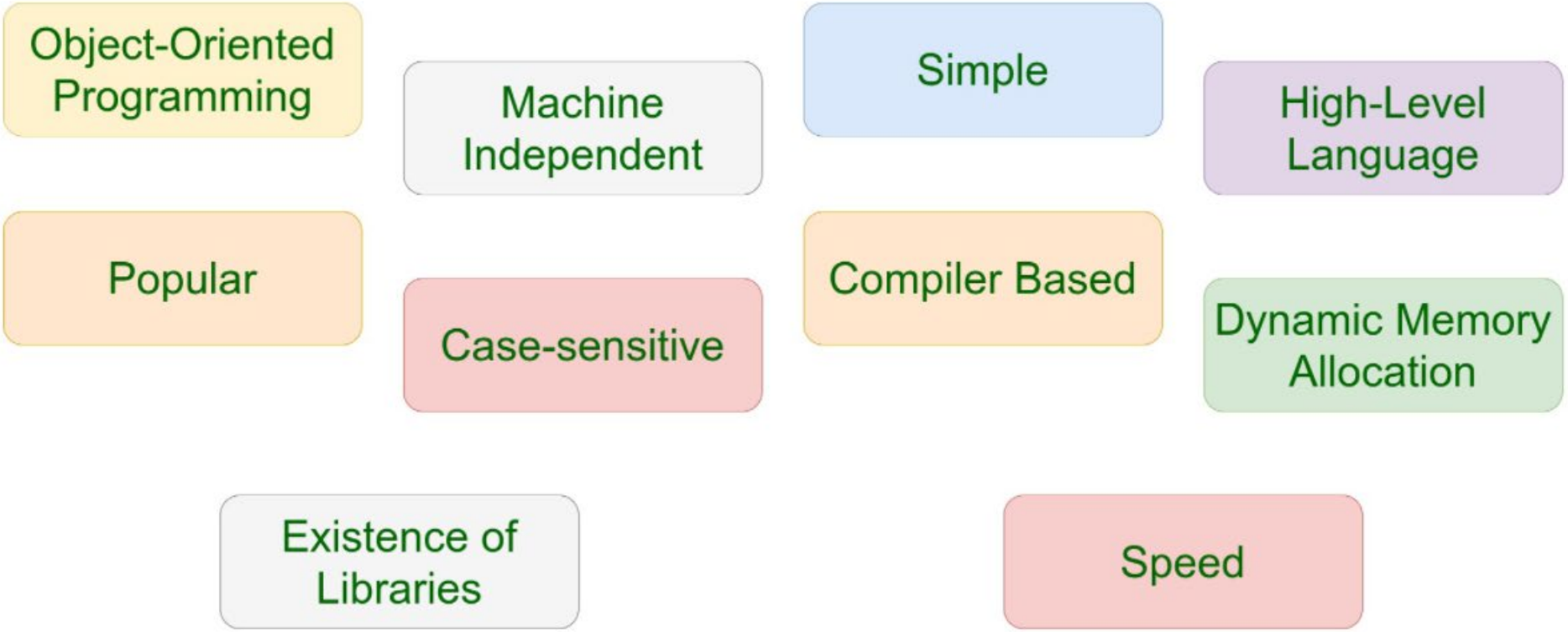
Learning Outcomes 1.3

At the end of this sub-chapter, students should be able to:

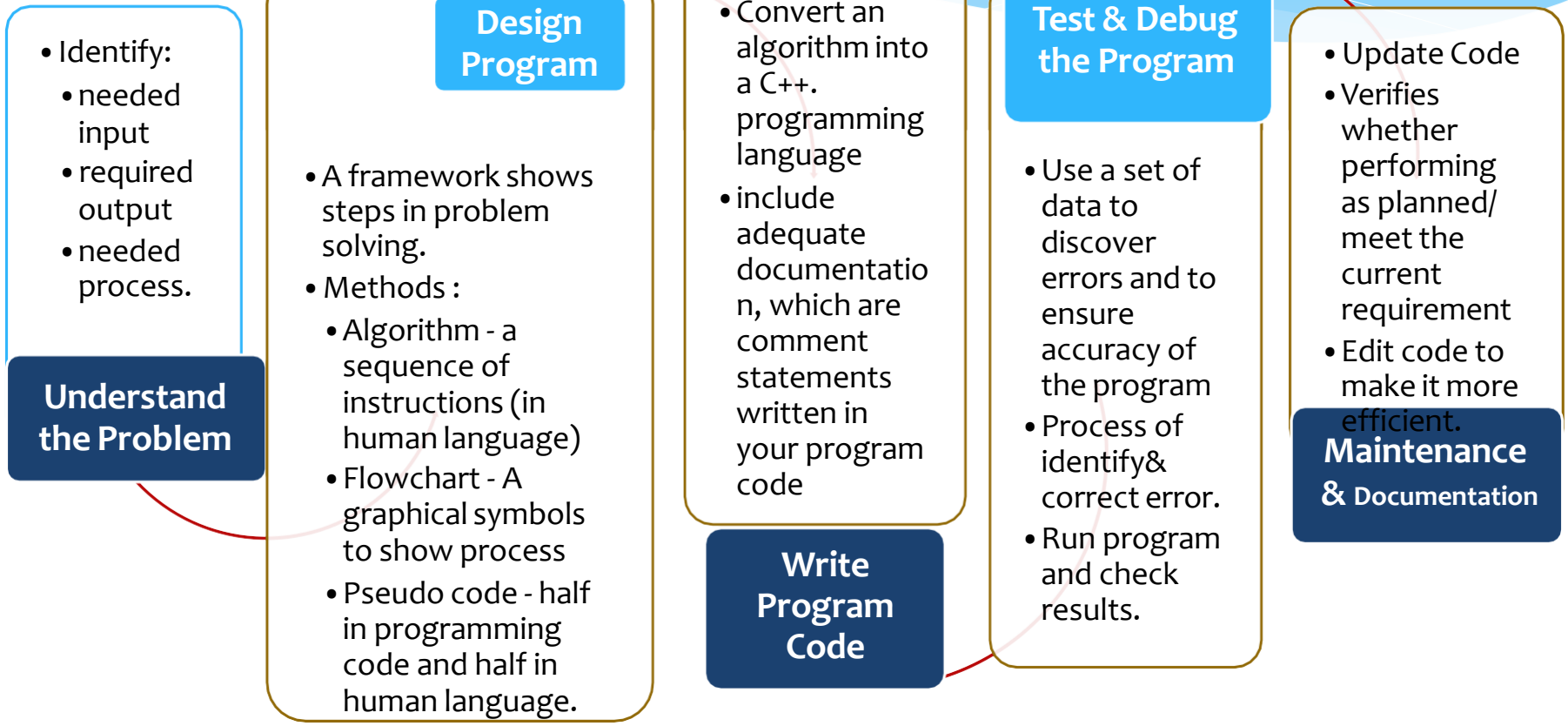
Identify the basic of computer program

- Describe features of C++ language
- Develop C++ using IDE

Features of C++



C++ Programming Development Process



Develop C++ program using Integrated Development Environment (IDE)

What IDE?

- An **integrated development environment (IDE)** a software application that provides comprehensive facilities to computer programmers for software development.
- All of the tools for writing, compiling and testing code in one place
- An IDE normally consists of:
  - a source code editor
  - a compiler and/or an interpreter
  - build automation tools
  - a debugger

A simple C++ program structure

Source code (The name of the file end in .cpp)

```
// my first program in C++

#include <iostream.h>
int main ()

{
    cout << "Welcome to C++!";
    return 0;
}
```

When executed, this program displays the message 'Welcome to C++! ' on the screen.

When it is compiled, it is called a executable code (running of the program)

Output

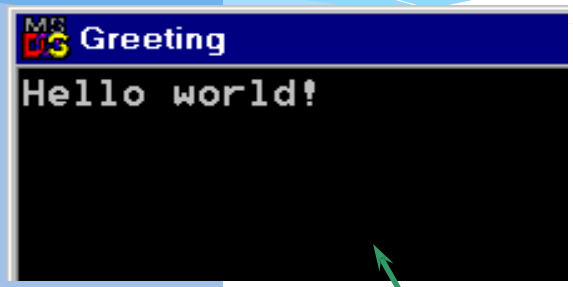
```
Welcome to C++!
```

First C++ Program - Greeting.cpp

```
// Program: Display greetings
/* Hello World!*

#include <iostream>

using namespace std;
void main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```



Preprocessor directives

Comments

Function named main() indicates start of program

Output

Ends executions of main() which ends program

Insertion statement

New line

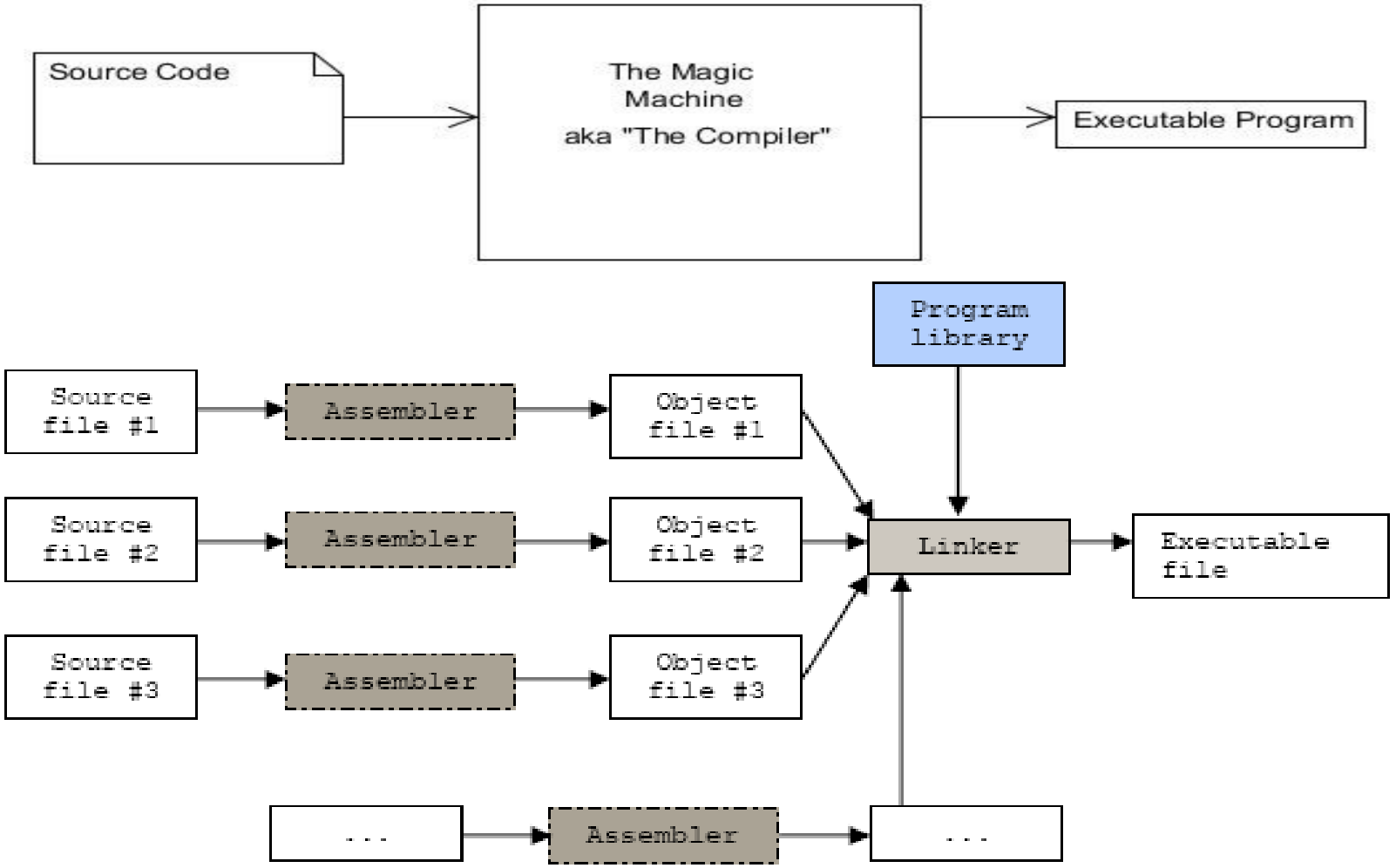
Learning Outcomes 1.4

At the end of this sub-chapter, students should be able to:

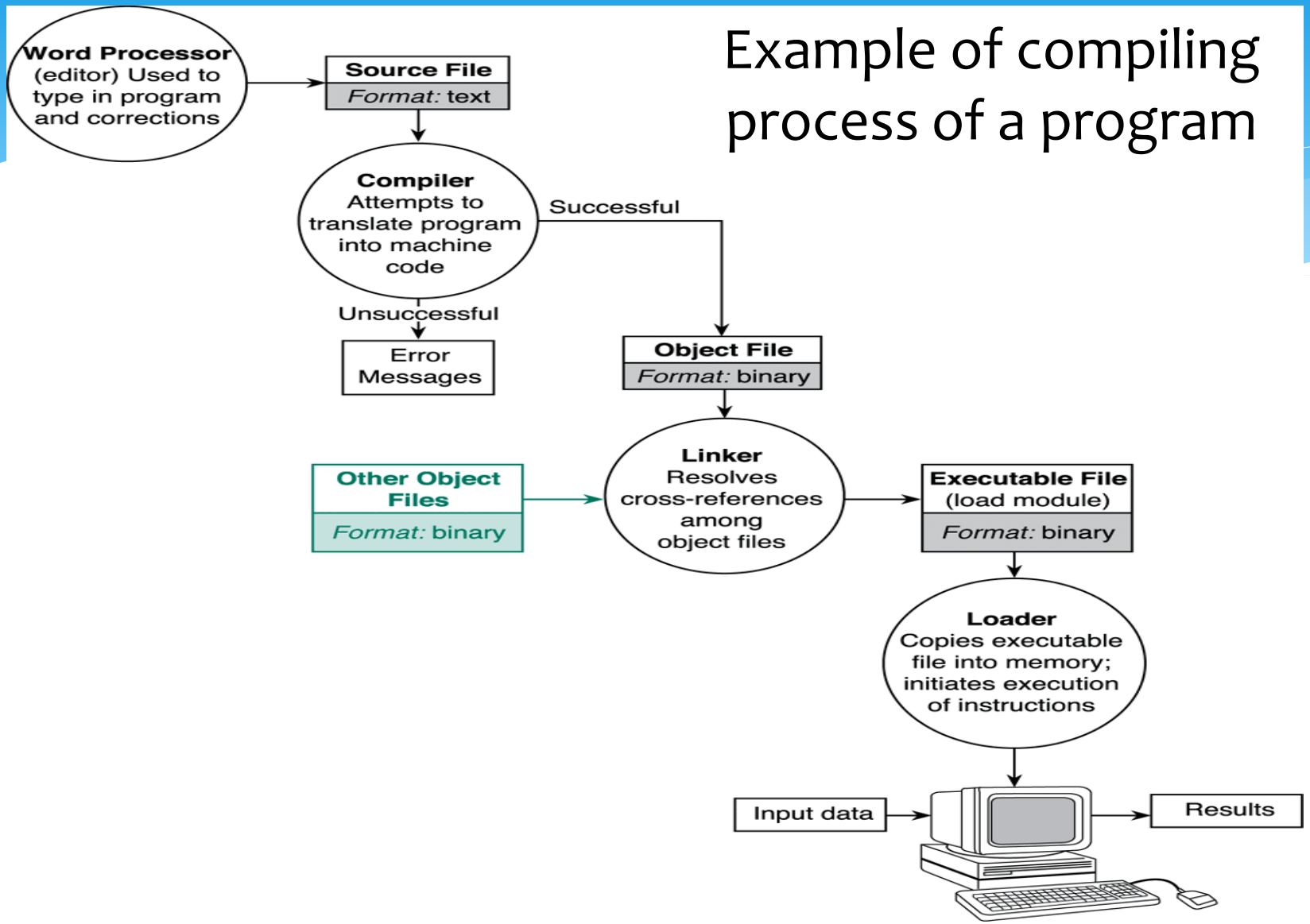
Compiling and debugging process & errors in programming

- Describe the compiling process
- Errors in programming
- Effective of debugging process
- Debug simple program

Compiling process



Example of compiling process of a program



Compiling process

1. Syntax Checking

- The code checking for valid syntax use compiler.
- This includes checking for semi-colons, matching braces, etc.
- This doesn't mean the code is correct, but it does determine whether the code can be turned into machine code as written.

2. Converting To Assembly

- This step doesn't exist for all languages/compilers.
- Compilers convert source code (C++ language) to machine code

3. Linking Machine Code Into An Executable

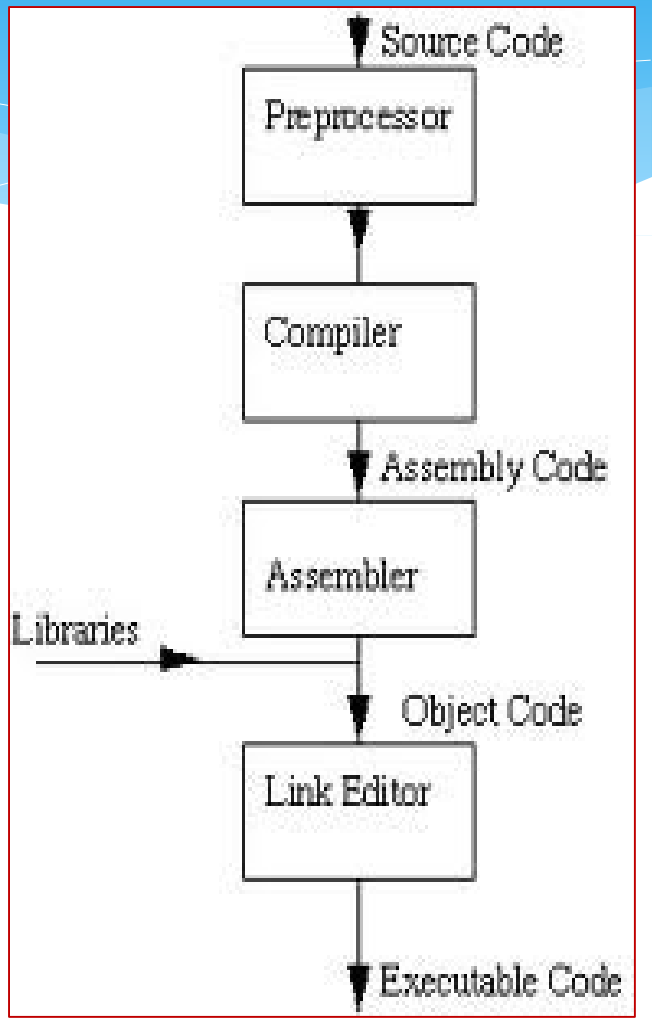
- At this step, many linkers will do final checks to make sure all the required pieces, functions, components, etc have been accounted for.

* **Compiler**

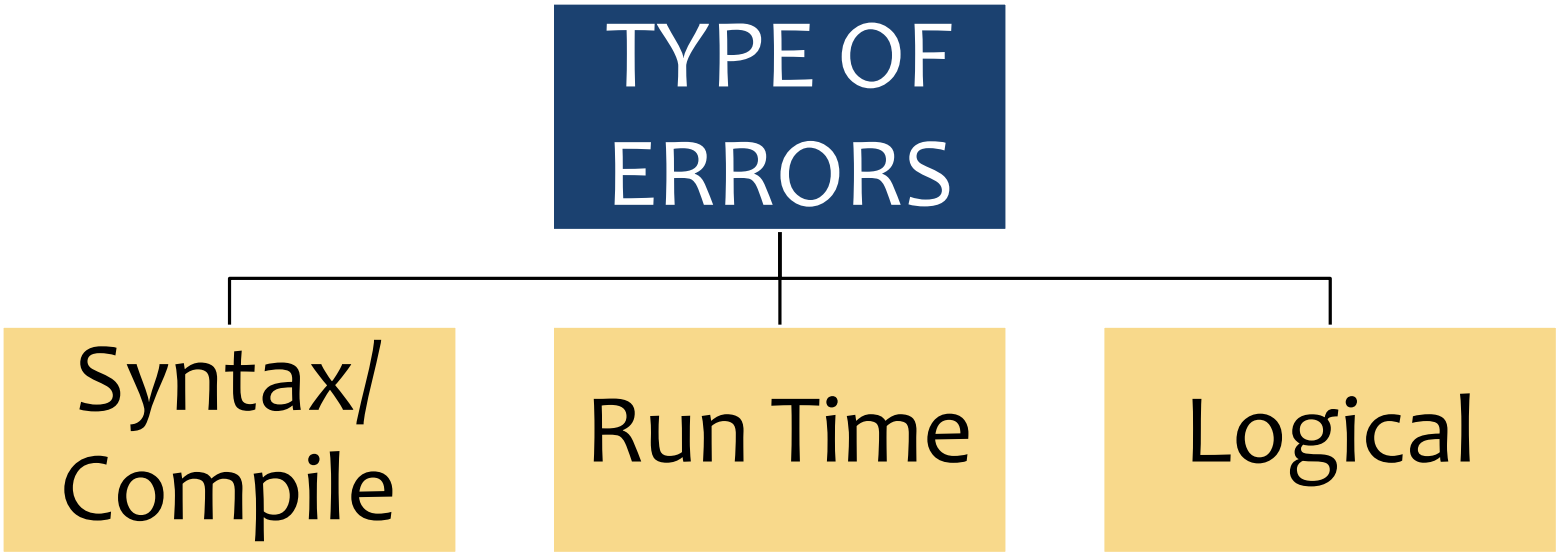
  This is where the "work" is done. High level code is translated into machine code, the result is object files.

  Linker

  All object files and relevant resources are linked together. Symbol information is verified (run into a Linker error may be more then once) and an executable is made.



Errors in programming



Errors in programming

Error	Decriptions	Common examples
Syntax errors/ compile error	*Grammar errors* in the use of the programming language.	✓ Misspelled variable and function names. ✓ Missing semicolons(;) ✓Improperly matches parentheses, square brackets[ ], and curly braces{ } ✓ Incorrect format in selection and loop statements
Run time errors	Occur when a program with no syntax errors asks the computer to do something that the computer is unable to reliably do.	✓Trying to divide by a variable that contains a value of zero ✓Trying to open a file that doesn't exist ✓There is no way for the compiler to know about these kinds of errors when the program is compiled.
Logic errors	Logic errors occur when there is a design flaw in your program.	✓Multiplying when you should be dividing ✓Adding when you should be subtracting ✓Opening and using data from the wrong file ✓Displaying the wrong message

Effective of debugging process

- Testing: Using a set of data to discover errors and to ensure accuracy of the program.
- Debugging : process of identifying and correcting error.

Discussion : Errors

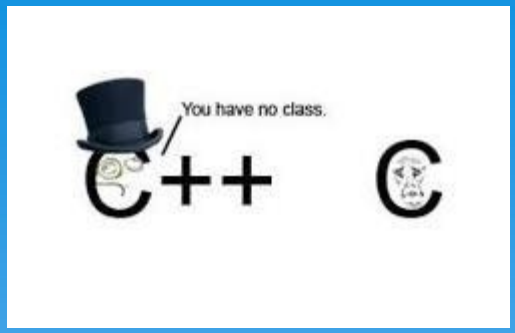
```
/ This is the first comment of the program
/* This is also a valid comment

#include <stdlib.h>
#include <iostream>
using namespace std;

void main ( )
{
   /* This is the first valid comment inside the main routine*/ */
   int X; Here is a variable X
   X = 5;
   cout << X;
}
```

1. Circle the syntax error in the following code

2. Write modification program (without error)



~~ The End ~~

Thank You