UEE1303
Objective-Oriented Programming

C++_Lecture 01:

C++ as a Better C

C: How to Program 8th ed.

Agenda

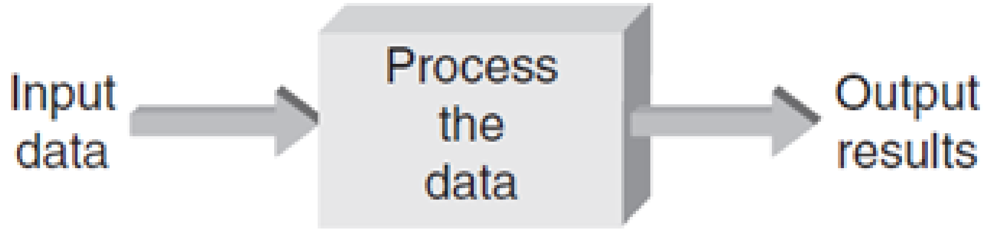
- Introduction to Programming
- Fundamentals of C/C++ programming
  - Datatypes and variables
  - Flow of control
  - Functions
  - Arrays
- Inline Function
- Empty Parameter Lists
- Default Arguments
- Unary Scope Resolution Operator
- Programming Style
- Function Overloading
- Function Template

Introduction to Programming

- Computer program
  - Data and instructions used to operate a computer
- Programming
  - Writing computer program in a language that the computer can respond to and that other programmers can understand
- Programming language
  - Set of instructions, data, and rules used to construct a program
    - Machine languages and assembly languages
    - High-level languages

Introduction to Programming (cont.)

- Procedural language
  - Instructions are used to create self-contained units (procedures)
  - Procedures accept data as input and transform data to produce a specific result as an output
  - Initially, high-level programming languages were predominately procedural



Introduction to Programming (cont.)

- Object-oriented languages
  - Program must define objects that it is manipulating
  - Such definitions include:
    - The general characteristics of objects
    - Specific operations to manipulate objects
- C++ is an object-oriented language
  - Has procedures and objects
  - Supports code reuse

Introduction to Programming (cont.)

- C++ began as extension to C
  - C is a procedural language developed in the 1970s at AT&T Bell Laboratories
- In early 1980s, Bjarne Stroustrup (also at AT&T) used his background in simulation languages to develop C++
- Object-orientation and other procedural improvements were combined with existing C language features to form C++

Program Translation

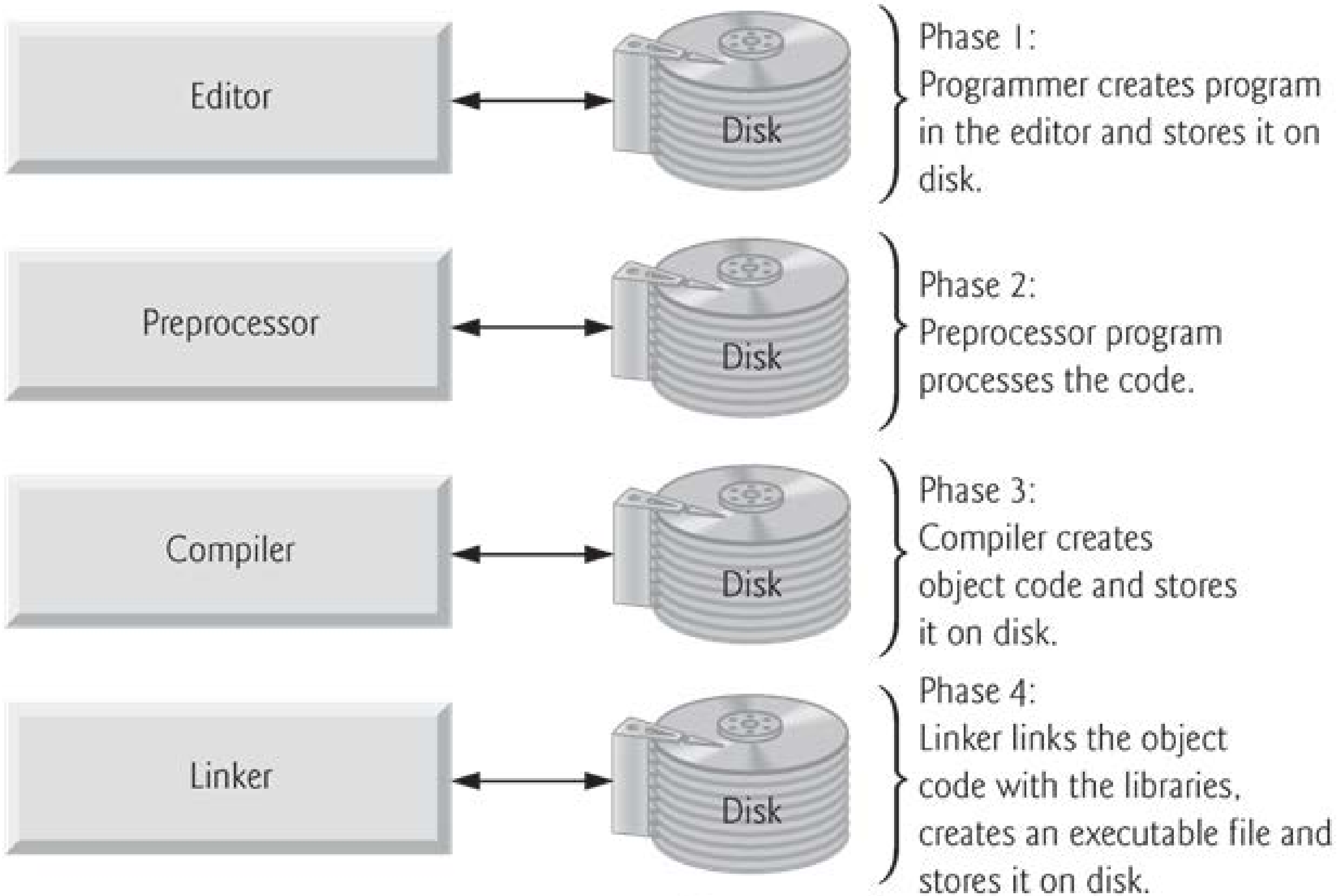
- C++ source program
  - Set of instructions written in C++ language
- Machine language
  - Internal computer language
  - Consists of a series of 1s and 0s
- Source program cannot be executed until it is translated into machine language
  - Interpreted language translates one statement at a time
  - Compiled language translates all statements together

Typical C/C++ Development Environment

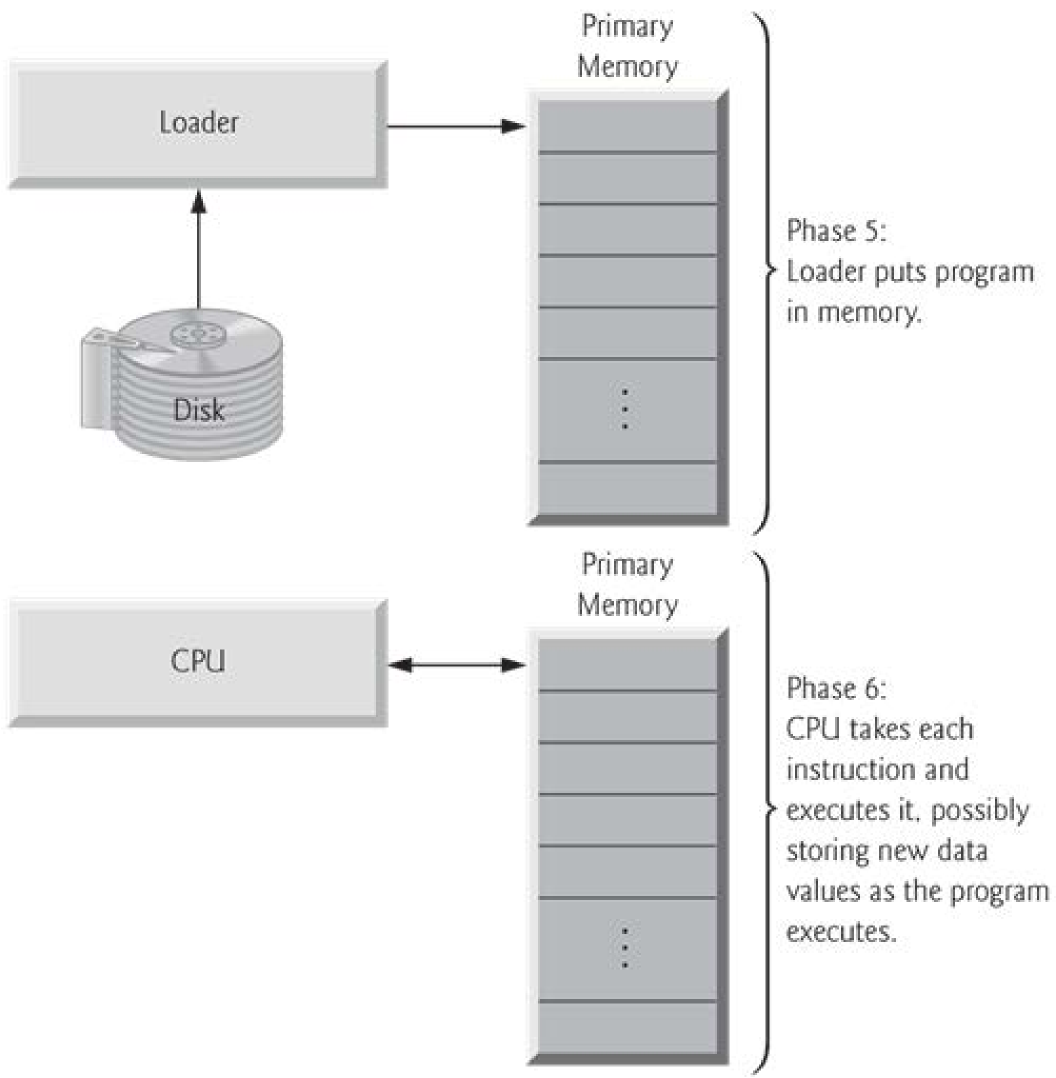
- C/C++ systems generally consist of three parts: a program development environment, the language and the C/C++ Standard Library

- C/C++ programs typically go through six phases: edit, preprocess, compile, link, load and execute

Typical C/C++ Development Environment (cont.)



Typical C/C++ Development Environment (cont.)



Phase 1: Creating a Program in an Editor

- Type a C/C++ program (source code) using the editor
- C/C++ source code filenames
  - often end with the .c/.cpp extensions
- Two editors widely used on UNIX systems
  - vi (vim) and emacs
- C/C++ software packages (which has editors integrated in the programming environment) in Windows systems
  - Microsoft Visual C++
  - Dev-C++
  - Code::Blocks

Phases 2 and 3: Preprocessing and Compiling a C/C++ Program

- In phase 2, you give the command to compile the program
  - In Linux, we use gcc/g++ command to compile the c/c++ program
- In a C/C++ system, a preprocessor program executes automatically before the compiler's translation phase begins
- The C/C++ preprocessor obeys commands called preprocessor directives, which indicate that certain manipulations are to be performed on the program before compilation
- In phase 3, the compiler translates the C/C++ program into machine-language code (also referred to as object code)

Phrase 4: Linking

- The object code produced by the C/C++ compiler typically contains “holes” due to missing parts, such as references to functions from standard libraries

- A linker links the object code with the code for the missing functions to produce an executable program

Phrase 5 and 6:
Loading and Executing a Program

- Before a program can be executed, it must first be placed in memory
- This is done by the loader, which takes the executable image from disk and transfers it to memory
- Additional components from shared libraries that support the program are also loaded

- Finally, in Phase 6, the computer, under the control of its CPU, executes the program one instruction at a time

First Program in C++

First Program in C:
Printing a Line of Text

- source code:

```
/* Fig. 2.1: fig02_01.c
   A first program in C    */
#include <stdio.h>

/* function main begins program execution */
int main()
{
    printf( "Welcome to C!\n" );

    return 0; /*indicate the program ended successfully*/
} /* end function main */
```

comments

preprocessor directive

Main function

- screen output:

```
Welcome to C!
```

First Program in C++: Printing a Line of Text

- source code:

```
// Fig. 2.1: fig02_01.cpp
// Text-printing program.
#include <iostream>

// function main begins program execution
int main()
{
    std::cout << "Welcome to C++!\n"; //display message

    return 0; // indicate the program ended successfully
} // end function main
```

comments

preprocessor directive

Main function

- screen output:

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

Comments

- Explanatory remarks written within program
  - Clarify purpose of the program
  - Describe objective of a group of statements
  - Explain function of a single line of code
- Computer ignores all comments
  - Comments exist only for convenience of reader
- A well-constructed program should be readable and understandable
  - Comments help explain unclear components

Comments (cont.)

- Line comment
  - Begins with two slashes(//) and continues to the end of the line
  - Can be written on line by itself or at the end of line that contains program code

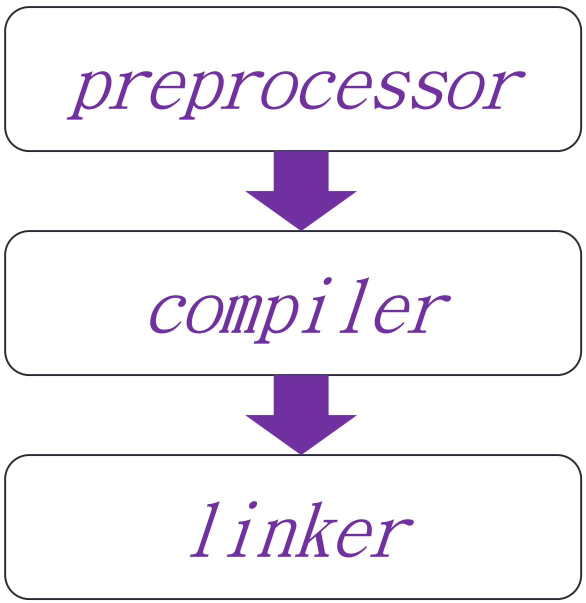
```
// hello.cpp
```

- Block comment
  - Multiple-line comment begins with the symbols /* and ends with the symbols */

```
/* This is a block comment that
 * spans
   three lines */
```

Preprocessor Directive

- Building a C++ program is a 3-step process



  - recognize *meta-information* about the code
  - translate source code into *machine-dependent* object code
  - link together all individual object files into an *application*
- Preprocessor aims at *directives* which starts with the # character

```
#include <iostream>
```

Preprocessor Directive (cont.)

- A preprocessor directive is a message to the C++ preprocessor
- Lines that begin with `#` are processed by the preprocessor before the program is compiled

- `#include <iostream>` notifies the preprocessor to include in the program the contents of the input/output stream header file `<iostream>`
  - Must be included for any program that outputs data to the screen or inputs data from the keyboard using C++-style stream input/output

C++ Standard Library

C++ Standard Library header file	Explanation
`<iostream>`	Contains function prototypes for the C++ standard input and standard output functions. This header file replaces header file `<iostream.h>`. This header is discussed in detail in Chapter 2, Stream Input/Output.
`<iomanip>`	Contains function prototypes for stream manipulators that format streams of data. This header file replaces header file `<iomanip.h>`. This header is used in Chapter 2, Stream Input/Output.
`<cmath>`	Contains function prototypes for math library functions. This header file replaces header file `<math.h>`.
`<cstdlib>`	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. This header file replaces header file `<stdlib>`.
`<ctime>`	Contains function prototypes and types for manipulating the time and date. This header file replaces header file `<time.h>`.

http://www.cplusplus.com/reference/

C++ Standard Library (cont.)

C++ Standard Library header file	Explanation
<vector>, <list>, <deque>, <queue>, <stack>, <map>, <set>, <bitset>	These header files contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution.
<cctype>	Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. This header file replaces header file <ctype.h>.
<cstring>	Contains function prototypes for C-style string-processing functions. This header file replaces header file <string.h>.
<typeinfo>	Contains classes for runtime type identification (determining data types at execution time).
<exception>, <stdexcept>	These header files contain classes that are used for exception handling (discussed in Chapter 2, Exception Handling).

The `cout` Object

- The `cout` object sends data to the standard output display device
  - The display device is usually a video screen.
  - Name derived from Console OUTput and pronounced "see out"
- Data is passed to `cout` by the insertion symbol

```
std::cout << "Welcome to C++!\n";
```

- The `<<` operator is referred to as the stream insertion operator.
  - The value to the operator's right, the right operand, is inserted in the output stream

The `cout` Object (cont.)

- The `std::` before `cout` is required when we use names that we've brought into the program by the preprocessor directive `#include <iostream>`
- The notation `std::cout` specifies that we are using a name, in this case `cout`, that belongs to "namespace" `std`
- The names `cin` (the standard input stream) and `cerr` (the standard error stream) also belong to namespace `std`

I/O Streams

- Common escape characters used with I/O
  - `\n`: new line
  - `\r`: carriage return
  - `\t`: tab
  - `\\`: the backslash character
  - `\"`: quotation mark
- `std::cin` accepts to the input from the user
  - Use `>>` operator with an input stream
  - Use input can be tricky since you can never know what kind of data the user input

Namespaces

- A program includes many identifiers defined in different scopes.
  - Identifier overlapping occurs frequently in third-party libraries that happen to use the same names for global identifiers (such as functions)
    - This can cause compiler errors
- Namespaces solve the naming conflicts between different pieces of code

Namespaces (cont.)

- Each namespace defines a scope in which identifiers and variables are placed.
- Ex: Having your own `foo()` and `foo()` from a third-party library
  - Compiler does not know which to call

```
//myspace.h
namespace mycode {
    void foo() {
        ...
    }
}
```

```
//usenamespaces.cpp
#include "myspace.h"
using namespace mycode;
int main(){
    foo();
    //use mycode::foo()
}
```

Namespaces (cont.)

- Namespace
  - File accessed by compiler when looking for prewritten classes or functions
- Sample namespace statement:
  - `using namespace std;`
  - `iostream` contained in a namespace called `std`
  - Compiler uses `iostream`'s `cout` object from `std` whenever `cout` is referenced

Example of namespace

```
// Fig. 24.3: fig24_03.cpp
// Demonstrating namespace
#include <iostream>
using namespace std;

int integer1 = 98; // global variable
// create namespace Example
namespace Example {
    const double PI = 3.14159;
    const double E = 2.71828;
    int integer1 = 8;

    void printValues(); // prototype
    // nested namespace
    namespace Inner {
        enum Years {FISCAL1 = 1990, FISCAL2, FISCAL3};
    }
}
```

Example of namespace (cont.)

```
// create unnamed namespace
namespace {
    double doubleInUnnamed = 88.22;
}
int main ()
{
    cout << "doubleInUnnamed = " << doubleInUnnamed;
    cout << "\n(global) integer1 = " << integer1;
    cout << "\nPI = " << Example::PI
         << "\nE = " << Example::E
         << "\ninteger1 = " << Example::integer1
         << "\nFISCAL3 = " << Example::Inner::FISCAL3
         << endl;

    Example::printValues();
    return 0;
}
```

Example of namespace (cont.)

```
void Example::printValues() {
    cout << "In printValues: \n integer1 = " << integer1
         << "\nPI = " << PI
         << "\nE = " << E
         << "\ndoubleInUnnamed = " << doubleInUnnamed
         << "\n(global) integer1 = " << ::integer1
         << "\nFISCAL3 = " << Inner::FISCAL3
         << endl;
}
```

Example of namespace (cont.)

```
doubleInUnnamed = 88.22
(global) integer1 = 98
PI = 3.14159
E = 2.71828
integer1 = 8
FISCAL3 = 1992

In printValues:
integer1 = 8
PI = 3.14159
E = 2.71828
doubleInUnnamed = 88.22
(global) integer1 = 98
FISCAL3 = 1992
```

First Program in C++: Printing a Line of Text (Modified Again)

- source code:

```
// Modified from Fig. 2.1: fig02_01.cpp
// Text-printing program.
#include <iostream>
using namespace std;
// function main begins program execution
int main(int argc, char *argv[])
{
    cout << "Welcome to C++!\n"; //display message

    return 0; // indicate the program ended successfully
} // end function main
```

comments

preprocessor directive

Main function

- screen output:

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

`main()` Function

`int main(int argc, char *argv[]);`

- where the program starts
- An `int` is returned indicate the result status; typically `return` 0
- `argc` gives the number of arguments passed to the program
- `argv` contains those arguments

- Ex: `>./prog 5 4.4 test1.txt`
  - `argc` = 4
  - `argv[0]` is `"prog"`, `argv[1]` is `"5"`, `argv[2]`is `"4.4"` and `argv[3]` is `"test1.txt"`

Variables and Datatypes

- C++ allows variables to be declared anywhere and hereafter uses them in the current block
- Datatypes: a set of *values* and *operations* that can be applied to these values
- Example of a data type: Integer
  - The values: set of all Integer (whole) numbers
    - Ex: 23, -5, 0 and 31932
  - The operations: familiar mathematical and comparison operators
    - Ex: +, -, > , <

Datatypes

- Built-In data type: Provided as an integrated part of C++
  - Also known as a *primitive* type
  - Requires no external code
  - Consists of basic numerical types (e.g. `int`, `float`)
  - Majority of operations are symbols (e.g. +, -, *,...)
- **`Class`** data type :
  - Programmer-created data type
  - Set of acceptable values and operations defined by a programmer using C++ code

Built-in Datatypes

Type	Description	Usage
`int`	Positive and negative integers	`int i = 7;`
`short/long`	Short/long integers	`short s = 13;` `long l = -55;`
`unsigned`	Limits the preceding types to ≥0	`unsigned int i = 2;` `unsigned long l = 23;`
`float` `double`	Floating-point and double-precision values	`float f = 7.2` `double d = 7.2`
`char`	Single characters	`char ch = 'm';`
`bool`	True or false	`bool b = true;`

Coercion

- Value on right side of a C++ expression is converted to data type of variable on the left side
- Example:
  - If temp is an integer variable, the assignment

```
temp = 25.89;
```

causes integer value 25 to be stored in integer variable `temp`

Type Casting

- Casting: explicitly convert the data type of a value to another data type
  - method 1: most common used; from C

```
bool someBool = (bool)someInt;
```

  - method 2: naturally byte rarely seen

```
bool someBool = bool(someInt);
```

  - method 3: verbose but clean

```
bool someBool =
static_cast<bool>(someInt);
```

- Coercion: automatically casting

```
int someInt = someDouble;
```

Operators

- Arithmetic: +, -, *, /, %
  - Shorthand: +=, -=, *=, /=, %=
  - Increment/decrement: ++, --
- Relational: ==, !=, >, >=, <, <=
  - used to compare operands
  - Format: x <op> y
- Logical: &&, ||, ! (AND, OR, NOT)
  - (age > 40) || (term < 10)
    - Compound condition is true if age > 40 or if term < 10 or if both conditions are true

Operator Precedence and Associativity

Operator	Associativity
`!(unary), -, ++, --`	right to left
`*, /, %`	left to right
`+, -`	left to right
`<, <=, >, >=`	left to right
`==, !=`	left to right
`&&`	left to right
`||`	left to right
`=, +=, -=, *=, /=`	right to left

User-defined Datatypes

- Enumerated type: the sequence of numbers
  - Format: **`enum`** `typename {id1,id2,id3,…};`
  - where `id1 < id2 < id3 < …`
- `struct` type: encapsulate one or more existing types into a new one

```
struct Structure_Tag
{
    Type_1 Member_Name_1;
    Type_2 Member_Name_2;
    …
    Type_N Member_Name_N;
};
```

  - access members by dot operator (.)

`typedef` & `struct` in C/C++

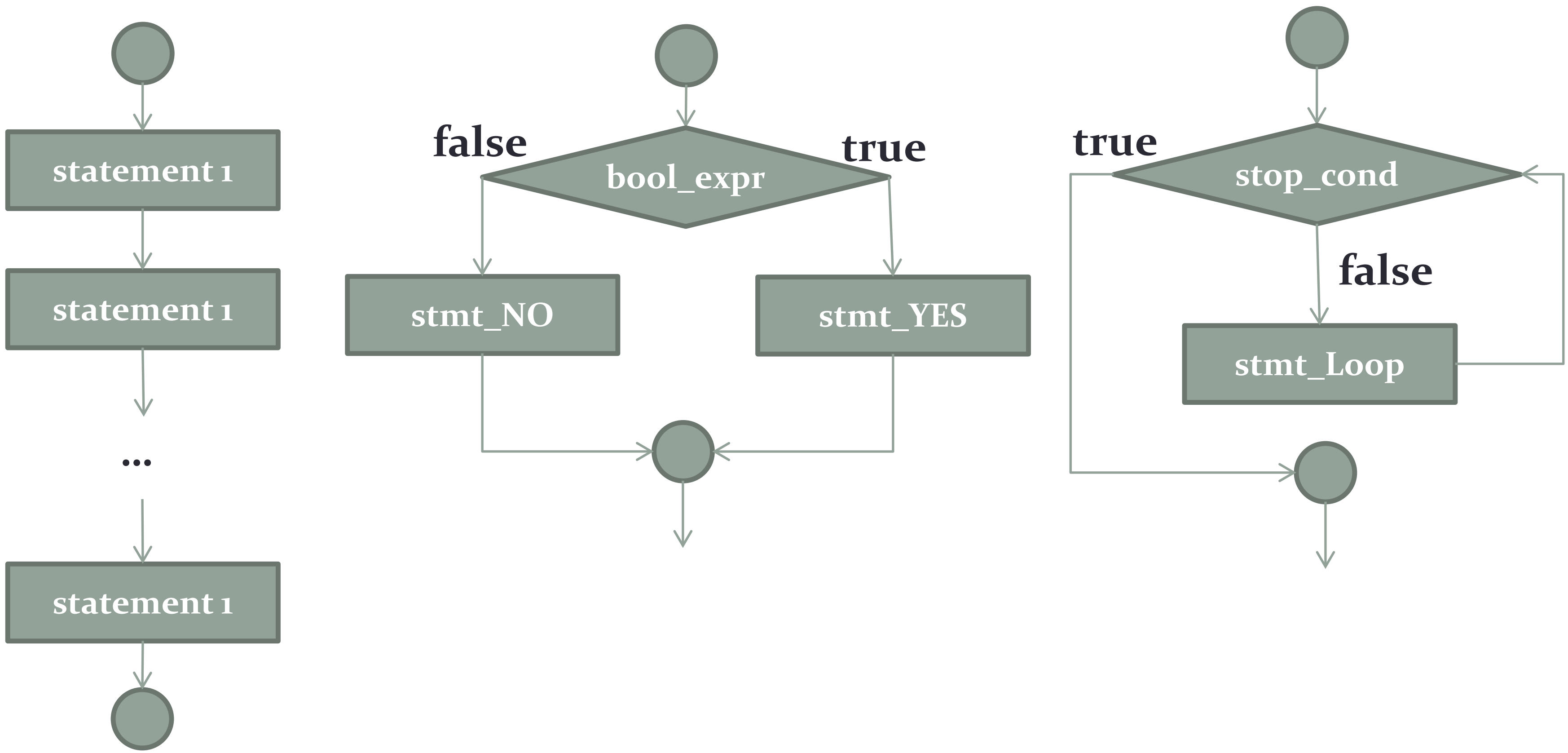
- [Type 1] use `struct` only

```
struct struct_tag
{
    struct_body;
};
```

- [Type 2] use `struct` with `typedef` ⇨ most common in C/C++

```
typedef struct
{
    struct_body;
} struct_tag;
```

Flow of Execution



(1) Sequence (2) Selection (3) Repetition

Flow of Control: Selection

- `if/else` statement

```
if (score >= 90) {
    grade = 'A';
} else if (score >= 80) {
    grade = 'B';
} else {
    grade = 'C';
}
```

- Ternary operator

```
cout << ((grade > 60)? "pass" : "fail");
```

Flow of Control: Selection (cont.)

- `switch` statement

```
switch (menu) {
    case item1:
        //do something
        break;
    case item2:
    case item3:
        //do something
        break;
    ...
    default:
        //do something
        break;
}
```

Flow of Control: Repetition

- `while` loop

```
while (i < 5) {
    cout << "good!" << endl;
    i++;
}
```

- `do/while` loop

```
do {
    cout << "good!" << endl;
    i++;
} while (i < 5);
```

Flow of Control: Repetition (cont.)

- `for` loop

```
for (int i = 0; i < 5; i++) {
    cout << "good!" << endl;
}
```

  - the most convenient

- Any `for` loop can be converted into a `while` loop

```
int i = 0;
while (i < 5) {
    cout << "good!" << endl;
    i++;
}
```

`break` & `continue` in Repetition

- Flow of Control
  - Recall how loops provide "graceful" and clear flow of control in and out
  - In RARE instances, can alter natural flow
- `break`
  - force the loop to exit immediately
- `continue`
  - skip the rest of loop body
- These statements violate natural flow
  - only used when absolutely necessary!

`break` in Loop

- `break`: forces immediate exits from structures:
  - in `switch` statements:
    - the desired case is detected/processed
  - in `while`, `for` and `do…while` statements:
    - an unusual condition is detected
- Example:

```
for (i = 10; i <= 50; i += 2)
{
    if ( i%9 == 0)
        break;
    cout << i << " ";
}
```

`continue` in Loop

- `continue`: cause the next iteration of the loop to begin immediately
  - execution transferred to the top of the loop
  - apply only to `while`, `for` and `do…while` statements
- Example:

```
i = 0;
while ( i < 100 ){
    i++;
    if ( i == 50)
        continue;
    cout << i << endl;
}
```

Functions

- Functions are building blocks of programs
  - Available for other code to use
  - *Declaration* (in header files) + *Definition* (in source files) + *Call* (used in the code)
- Declaration (a.k.a. *prototype* or *signature*)
  - how the function can be accessed
  - syntax: `<type> FnName(<parameters>);`
  - placed before any calls in *declaration space* of main() or in *global space*

```
double totalCost(int num, double price);
```

Functions (cont.)

- Definition is the *implementation* of function
  - The *link* stage searches the right function

```
double totalCost(int num, double price)
{
    return (num * price * 1.05);
}
```

- Calls to the function in the program
  - pass in *constants* or *variables*

```
totalCost(8, 9.5);

totalCost(inum, 9.5);

totalCost(10, dprice);
```

Array

- An array are a *collection of data* of same type
  - in C++, size must be a constant
  - C++ allow multidimensional arrays
  - three-dimensional or higher is rarely used
- An example of Tic-Tac-Toe board

```
char ticTacToe[3][3];
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        ticTacToe[i][j] = 'x';
    }
}
```

  - The first element is always at position 0
  - The last element is at position (size -1)

Another C++ Program: Adding Integers

```
// Fig. 15.1: fig15_01.cpp
// Addition program
#include <iostream>

int main( )
{
    int number1;   // first integer to add          declaration

    std::cout << "Enter first integer: ";
    std::cin >> number1;

    int number2;   // second integer to add         declaration
    int sum;       // sum of number1 and number2

    std::cout << "Enter second integer: ";
    std::cin >> number2;
```

Another C++ Program: Adding Integers (cont.)

```
    // add the integers; store result in sum
    sum = number1 + number2;

    // display sum; end line
    std::cout << "Sum is " << sum << std::endl;

    return 0; // indicate the program ended successfully
} // end function main
```

Another C++ Program: Adding Integers (cont.)

- screen output:

```
Enter first integer
45
Enter second integer
72
Sum is 117
```

Inline Function

- Calling functions: creates overhead
  - Placing arguments in reserved memory
  - Passing control to the function
  - Providing memory space for any returned value
  - Returning to proper point in calling program

- Overhead is justified when function is called many times
  - Better than repeating code

Inline Functions (cont.)

- An `inline` function
  - where compiler performs *inline expansion*
  - reduce the program execution time
  - improve over macros
- Good candidates are *small*, *frequently* called functions

```
inline double circleArea(double x) {
    return (3.14159 * x * x);
}

inline double tempvert(double inTemp) {
    return (5.0/9.0) * (inTemp - 32.0);
}
```

Inline Functions (cont.)

- Only need to include the **`inline`** keyword before the declaration or definition:

```
inline int min(int v1, int v2) {
    return (v1 < v2 ? v1 : v2);
}
```

- The inline function is expanded “inline” at each point in the program in which it is invoked
  - Ex: `int minVal = min(i, j);`

    is expanded during compilation into

    `int minVal = i < j ? i : j;`

Example of Inline Function

```
// Fig. 15.3: fig15_03.cpp
// Using an inline function to calculate the volume
// of a cube
#include <iostream>
using std::cout;
using std::cin;
using std::endl;

// definition of inline function cube.
inline double cube ( const double side )
{
    return side * side * side; // calculate cube
}
```

Example of Inline Function (cont.)

```
int main ()
{
    double sideValue; // stores value entered by user

    for ( int i = 1; i <= 3; i++ )
    {
        cout << "\nEnter the side length of your cube: ";
        cin >> sideValue;
        cout << "Volume of cube with side " << sideValue
             << " is " << cube(sideValue) << endl;
    }
    return 0;
}
```

Empty Parameter Lists

- In C++, an empty parameter list is specified by writing either `void` or nothing at all in parentheses.
- The prototype

```
void print();
```

specifies that function print does not take arguments and does not return a value.

Example of Functions with Empty Parameter Lists

```
// Fig. 5.16: fig05_16.cpp
// Function that take no arguments.
#include <iostream>
using namespace std;

void function1(); // function prototype
void function2( void ); // function prototype

int main ()
{
    function1();
    function2();
    return 0;
}
```

Example of Functions with Empty Parameter Lists (cont.)

```
void function1()
{
    cout << "function1 takes no arguments” << endl;
}
void function2( void )
{
    cout << "function2 also takes no arguments" << endl;
}
```

- screen output

```
function1 takes no arguments
function2 also takes no arguments
```

Default Arguments

- It isn't uncommon for a program to invoke a function repeatedly with the same argument value for a particular parameter
- In such cases, you can specify that such a parameter has a default argument, i.e., a default value to be passed to that parameter
- When a program omits an argument for a parameter with a default argument in a function call, the compiler rewrites the function call and inserts the default value of that argument

Example of Default Arguments

```
// Fig. 15.8: fig15_08.cpp
// Using default arguments.
#include <iostream>
using namespace std;

// function prototype
int boxVolume( int length = 1, int width = 1, int height =
1);

int main ()
{
    // no arguments - use default value
    cout << "The default box volume is " << boxVolume();
    // specify length; default width and height
    cout << "\n\nThe volume of a box with length 10,\n"
         << "width 1 and height 1 is: " << boxVolume(10);
```

Example of Default Arguments (cont.)

```
    // specify length, and width; default height
    cout << "\n\nThe volume of a box with length 10,\n"
         << "width 5 and height 1 is: "
         << boxVolume(10, 5);
    // specify all arguments
    cout << "\n\nThe volume of a box with length 10,\n"
         << "width 5 and height 2 is: "
         << boxVolume(10, 5, 2);
    return 0;
}
// function boxVolume
int boxVolume( int length, int width, int height)
{
    return length * width * height;
}
```

Example of Default Arguments (cont.)

- screen output

```
The default box volume is: 1

The volume of a box with length 10,
Width 1 and height 1 is: 10

The volume of a box with length 10,
Width 5 and height 1 is: 50

The volume of a box with length 10,
Width 5 and height 2 is: 100
```

Unary Scope Resolution Operator

```
// Fig. 15.9: fig15_09.cpp
// Using the unary scope resolution operator.
#include <iostream>
using namespace std;

int number = 7; // global variable

int main ()
{
    double number = 10.5; // local variable
    cout << "Local double value of number = "
         << number
         << "\nGlobal int value of number = "
         << ::number << endl;
    return 0;
}
```

Motivation of Programming Style

- An sample program (look like yours?)

```
#include <iostream>
using namespace std;
int main(){
int x,y,z;
double p,q,r;
cin>>x>>r;
if(x>r){ y=r;
cout<<(x*=y);}
else{y=x; q=++r;
for(int i=0;i<q;i++){
if(y>r)cout<<(i*y);
}
else cout<<(i/r)}reurn 0;
}
```

Good Program Format

```
int main ()
{
    first statement;
    second statement;
    third statement;
    fourth statement;

    return 0;
}
```

Function Overloading

- Function overloading: using same function name for more than one function
  - Compiler must be able to determine which function to use based on data types of arguments (not data type of return value)
  - Requirement: the function arguments must have different data types
    - So that the compile can determine which function to call
- Each function must be written separately
  - Each acts as a separate entity
- Use of same function name does not require code to be similar
  - *Good programming practice*: functions with the same name perform similar operations

Example of Function Overloading

```
// Fig. 15.10: fig15_10.cpp
// Overloaded functions.
#include <iostream>
using namespace std;

int square( int x )
{
    cout << "Square of integer " << x << " is ";
    return x * x;
}
double square( double y )
{
    cout << "Square of double " << y << " is ";
    return y * y;
}
```

Example of Function Overloading (cont.)

```
int main ()
{
    cout << square(7);  // calls int version
    cout << endl;
    cout << square(7.5); // calls double version
    cout << endl;
    return 0;
}
```

- screen output

```
Square of integer 7 is 49
Square of double 7.5 is 56.25
```

Example2 of Function Overloading

```
//computes average of two numbers
double average(double n1, double n2)
{
    return ((n1 + n2) / 2.0);
}
//compute average of three numbers:
double average(double n1, double n2, double n3)
{
    return ((n1 + n2 + n3) / 3.0);
}
```

Example2 of Function Overloading (cont.)

- Which function gets called?
- Depends on function call itself:

```
avg = average(5.2, 6.7);
```

call "two-parameter average()"

```
avg = average(6.5, 8.5, 4.2);
```

call "three-parameter average()"

- Compiler resolves invocation based on the signature of function call
  - match call with appropriate function
  - each considered as a separate function

Overloading Pitfall

- Only overload same-task functions
  - A `average()` function should always perform same task, in all overloads
  - Otherwise, unpredictable results
- C++ function call resolution:
  - 1st order: looks for exact signature
  - 2nd order: looks for compatible signature

Overloading Resolution

- 1st exact match: look for exact signature
  - no argument conversion required
- 2nd compatible match: look for compatible signature where automatic type conversion is possible:
  - 1st with promotion (e.g., `int` => `double`)

    => no loss of data
  - 2nd with demotion (e.g., `double` =>`int`)

    => possible loss of data

Example of Overloading Resolution

- Given the following functions:

  (1) `int func(int n, double m);`

  (2) `int func(double n, int m);`

  (3) `int func(int n, int m);`

- Consider these calls:

  `func(98, 99);` => call `func(3)`

  `func (5.3, 4);` => call `func(2)`

  `func (4.3, 5.2);` => call ???

- Should avoid such confusing overloading

Type Conversion in Overloading

- Numeric formal parameters typically made "**`double`**" type
- Allows for "any" numeric type
  - any "subordinate" data automatically promoted
  - Ex:

```
int   =>  double
float =>  double
char  =>  double
```

Example of Automatic Type Conversion

```
double mpg(double miles, double gallons)
{
    return (miles/gallons);
}
```

- Examples of function calls:

  `mpgComputed = mpg(5,20);`

  => convert 5 & 20 to 5.0 & 20.0, then passes

  `mpgComputed = mpg(5.8,20.2);`

  => no conversion necessary

  `mpgComputed = mpg(5,2.4);`

  => convert 5 to 5.0, then passes values

Function Templates

- Most high-level languages require each function to have its own name
  - Can lead to a profusion of names
- Example: functions to find the absolute value
  - Three separate functions and prototypes are required

```
void abs(int);
void fabs(float);
void dabs(double);
```

- Each function performs the same operation
  - Only difference is the data type in argument

Example of Function Template

```
template <class T>
void showabs(T number)
{
    if (number < 0)
        number = -number;
    cout << "The absolute value of the number "
         << " is " << number << endl;
    return;
}
```

- Template allows for one function instead of three
  - T represents a general data type
  - T is replaced by an actual data type when compiler encounters a function call

Example of Function Template (cont.)

```
int main()
{
    int num1 = -4;
    float num2 = -4.23F;
    double num3 = -4.23456;
    showabs(num1);
    showabs(num2);
    showabs(num3);
    return 0;
}
```

- screen output

```
The absolute value of the number is 4
The absolute value of the number is 4.23
The absolute value of the number is 4.23456
```

Example2 of Function Template

```
// Fig. 15.12: maximum.h
// Definition of function template maximum.
template <class T> // or template<typename T>
T maximum(T value1, T value2, T value3)
{
    T maximumValue = value1; // assume value1 is maximum

    if (value2 > maximumValue)
        maximumValue = value2;

    if (value3 > maximumValue)
        maximumValue = value3;

    return maximumValue;
}
```

Example2 of Function Template (cont.)

```
// Fig. 15.13: fig15_13.cpp
// Function template maximum test program.
#include <iostream>
using namespace std;

#include "maximum.h"
int main ()
{
    int int1, int2, int3;
    cout << "Input three integer value: ";
    cin >> int1 >> int2 >> int3;

    cout << "The maximum integer value is: "
         << maximum (int1, int2, int3);
```

Example2 of Function Template (cont.)

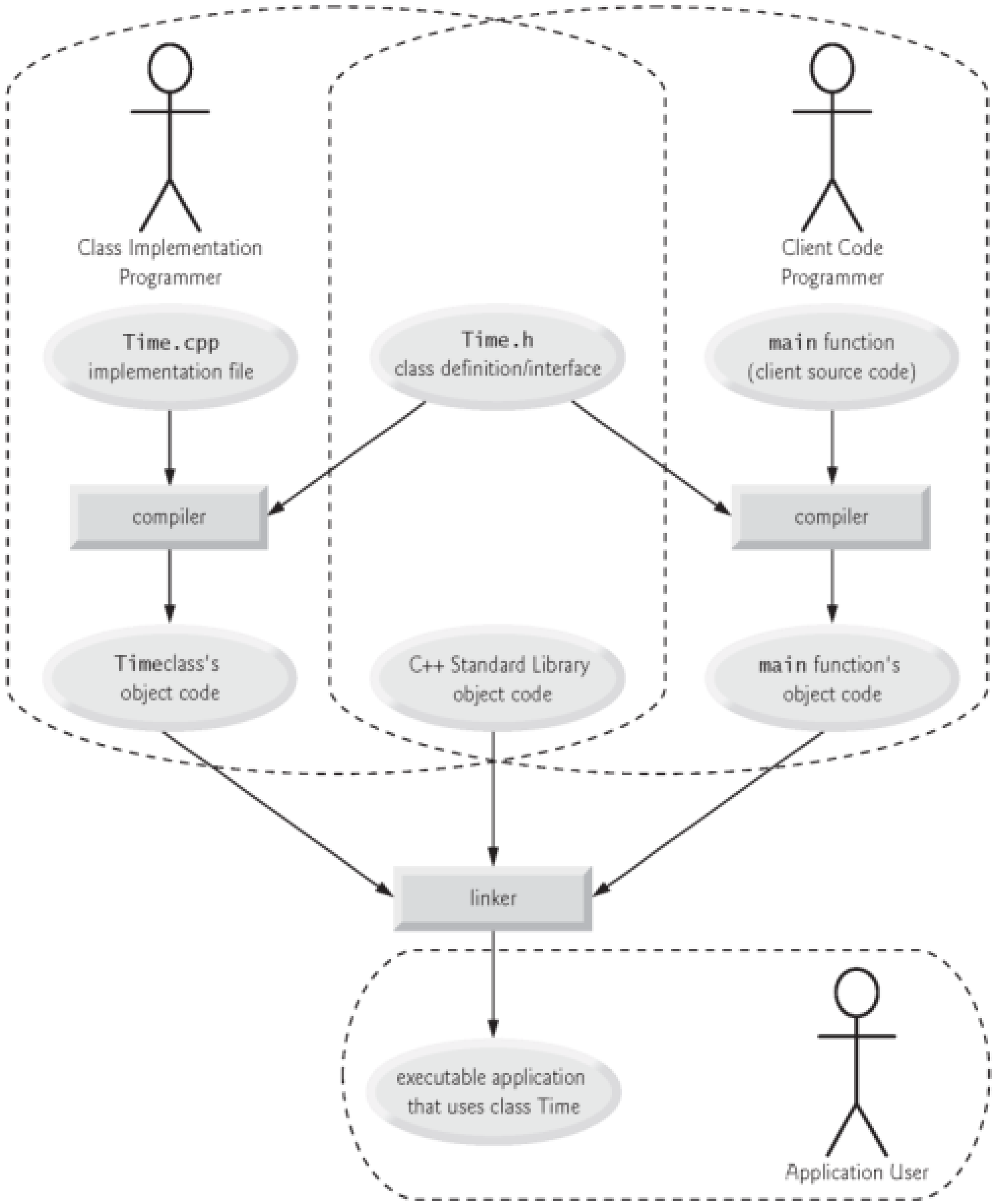
```
    double double1, double2, double3;
    cout << "\n\nInput three double value: ";
    cin >> double1 >> double2 >> double3;

    cout << "The maximum double value is: "
         << maximum (double1, double2, double3);

    char char1, char2, char3;
    cout << "\n\nInput three char value: ";
    cin >> char1 >> char2 >> char3;

    cout << "The maximum char value is: "
         << maximum (char1, char2, char3);
    return 0;
}
```

Summary

- Review basic C/C++ programming
  - Comments: line vs. block
  - Preprocessor directives
  - Datatypes: built-in and user-defined
  - Variables: local vs. global
  - Flow of control: selection (`if/else`, `switch`, ternary) + repetition (`for`, `while`, `do/while`)
  - Functions: declaration, definition and call
- Introduce new characteristics of C++.

References

- Paul Deitel and Harvey Deitel, “C How to Program” Seventh Edition (Eighth Edition)
  - Chapter 2-6
  - Chapter 15
- Paul Deitel and Harvey Deitel, “C++ How to Program (late objects version)” Seventh Edition
  - Chapter 2-7
  - Chapter 24.4
- W. Savitch, “Absolute C++,” Fourth Edition
  - Chapter 1~5


Class Implementation Programmer
Client Code Programmer
Time.cpp implementation file
Time.h class definition/interface
main function (client source code)
compiler
compiler
Timeclass's object code
C++ Standard Library object code
main function's object code
linker
executable application that uses class Time
Application User