C ++ Programming
for Non-C Programmers

Gracy Koshy
Robert J. Oberg

Student Guide
Revision 8.0

Object Innovations Course 153
This Student Guide consists of two modules:
  Introduction to C++
  Object-Oriented C++ Programming Fundamentals

Information in this document is subject to change without notice. Companies, names and data used in examples herein are fictitious unless otherwise noted. No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without the express written permission of Object Innovations.

Product and company names mentioned herein are the trademarks or registered trademarks of their respective owners.

Copyright © 2006 Object Innovations Enterprises, LLC
All rights reserved.

Object Innovations
877-558-7246
www.objectinnovations.com

Printed in the United States of America.
Table of Contents

Module 1. Introduction to C++

Chapter 1  First C++ Programs
Chapter 2  Data Types and Operators
Chapter 3  Logical and Bit Operations
Chapter 4  Loops and Structured Programming
Chapter 5  Functions and Program Structure
Chapter 6  Pointers and Arrays
Chapter 7  Structures and Unions

Module 2. Object-Oriented C++ Programming Fundamentals

Chapter 1  Concepts of Object-Oriented Programming
Chapter 2  Classes in C++
Chapter 3  Functions in C++
Chapter 4  Constructors and Destructors
Chapter 5  Memory Management in C++
Chapter 6  References and Argument Passing in C++
Chapter 7  Operator Overloading, Initialization, and Assignment
Chapter 8  Scope and Access Control
Chapter 9  Introduction to Inheritance
Chapter 10 Polymorphism and Virtual Functions

Appendix A  Reading List
Directory Structure

- The course software installs to the root directory \texttt{OCI\ IntroCpp} for Module 1 and \texttt{OIC\ FndCpp} for Module 2.
  
  - Example programs for each chapter are in named subdirectories of chapter directories \texttt{Chap01}, \texttt{Chap02}, and so on.

  - The \textbf{Labs} directory contains one subdirectory for each lab, named after the lab number. Starter code is frequently supplied, and answers are provided in the chapter directories.

  - The \textbf{Demos} directory is provided for doing in-class demonstrations led by the instructor.

  - The \textbf{CaseStudy} directory in Module 2 contains a case study in multiple steps.
# Table of Contents (Detailed)

## Module 1. Introduction to C++

### Chapter 1  First C++ Programs ................................................................. 1

- Hello, World .................................................................................................. 3
- Compiling and Running .................................................................................. 4
- C++ Program Components ............................................................................... 5
- Another C++ Program (Preview) ................................................................... 6
- I/O in C++ ....................................................................................................... 7
- Exercise ........................................................................................................... 8
- Answer ............................................................................................................ 9
- Variables ....................................................................................................... 10
- Expressions .................................................................................................... 11
- Assignment .................................................................................................... 12
- Using C++ as a Calculator ............................................................................ 13
- Sample Program ............................................................................................ 14
- Input in C++ ................................................................................................... 15
- Echo Program ................................................................................................ 16
- Lab 1 ............................................................................................................. 17
- Summary ....................................................................................................... 18

### Chapter 2  Data Types and Operators ...................................................... 21

- Data Types .................................................................................................... 23
- Strong Typing ............................................................................................... 24
- Typing in C++ ............................................................................................... 25
- Conversions in C++ ....................................................................................... 26
- Typing in Visual Basic .................................................................................... 27
- Primitive C++ Data Types ............................................................................ 28
- Signed and Unsigned ..................................................................................... 29
- Twos Complement ........................................................................................ 30
- C++ Integer Type Ranges ............................................................................ 31
- IntegerRange.cpp .......................................................................................... 32
- Integer Constants ......................................................................................... 33
- Variables in C++ ........................................................................................... 34
- Const Variables ............................................................................................. 35
- Preprocessor Macros ..................................................................................... 36
- Identifiers ...................................................................................................... 37
- Floating Point Numbers ................................................................................. 38
- Floating Point Data Types ............................................................................. 39
- Calculations .................................................................................................. 40
- Integer Operations ........................................................................................ 41
- Increment and Decrement ............................................................................ 42
- Precedence of Integer Operations ............................................................... 43
- Floating Point Operations ............................................................................. 44
- $\text{op}= $ ...................................... 45
### Chapter 5  Functions and Program Structure ................................................................. 101
- Basics of Functions ........................................................................................................ 103
- Example ......................................................................................................................... 104
- Function Prototypes and Type Checking ....................................................................... 105
- Returning Value ............................................................................................................ 106
- Argument Passing .......................................................................................................... 107
- Pass-by-Reference .......................................................................................................... 108
- Alternative to Pass-by-Reference .................................................................................... 109
- External Variables ......................................................................................................... 110
- Extern Example ............................................................................................................. 111
- Block Structure .............................................................................................................. 112
- Scope Rules .................................................................................................................... 113
- Scope Example .............................................................................................................. 114
- Header Files .................................................................................................................. 116
- Preprocessor Directives ............................................................................................... 117
- Conditional Compilation ............................................................................................... 118
- Recursion ...................................................................................................................... 119
- Lab 5 ............................................................................................................................ 120
- Summary ....................................................................................................................... 121

### Chapter 6  Pointers and Arrays.................................................................................. 125
- Pointers and Addresses ................................................................................................. 127
- Pointers and Function Arguments ............................................................................... 128
- Pointers and Arrays ...................................................................................................... 129
- Array Example ............................................................................................................. 131
- Address Arithmetic ..................................................................................................... 132
- Pointer Math Example ................................................................................................. 133
- Dynamic Memory Management .................................................................................... 134
- Using `malloc` and `free` ............................................................................................ 135
- Dynamic Memory Example ......................................................................................... 136
- Array of Pointers ......................................................................................................... 137
- Strings ......................................................................................................................... 138
- String Functions .......................................................................................................... 139
- Array of Strings .......................................................................................................... 140
- Command Line Arguments ......................................................................................... 141
- Command Line Argument Demo ................................................................................. 142
- Pointers to Functions (Optional) .................................................................................. 143
- Lab 6 ............................................................................................................................ 146
- Summary ....................................................................................................................... 147

### Chapter 7  Structures and Unions............................................................................ 151
- Fundamentals of Structures .......................................................................................... 153
- Structure Declaration ................................................................................................... 154
- Structures and Functions ............................................................................................ 155
- Structure Example ....................................................................................................... 156
- Typedef ......................................................................................................................... 157
- Arrays of Structures .................................................................................................... 158
- Array of Structures Initialization .................................................................................. 159
Chapter 4 Constructors and Destructors ............................................................ 227
The Problem of Initialization ............................................................................. 229
Constructors and Initialization ......................................................................... 230
Constructor in Stack Class .................................................................................. 231
Object Creation and Destruction ....................................................................... 232
Destructors .......................................................................................................... 233
Multiple Constructors ......................................................................................... 234
String Class Implementation ............................................................................. 235
String Class Demo ............................................................................................... 236
Hidden Constructors ........................................................................................... 237
Using a Default Argument .................................................................................. 238
Lab 4A .................................................................................................................. 239
Lab 4B .................................................................................................................... 240
Summary .............................................................................................................. 241

Chapter 5: Memory Management in C++ ......................................................... 247
Why Is Memory Management Important in C++? .......................................... 249
Choices for an Object's Memory ......................................................................... 250
Typical Memory Layout ....................................................................................... 251
Free Store Allocation .......................................................................................... 252
new Operator ....................................................................................................... 253
new vs. malloc ..................................................................................................... 255
delete Operator .................................................................................................... 256
Destructor (Review) .............................................................................................. 257
Hiding Memory Management ............................................................................. 258
String Class Specification (Version 2) ................................................................. 259
String Class Implementation ............................................................................. 260
String Class Demo ............................................................................................... 261
String Class Bug ................................................................................................... 262
String Class Demo Workaround ........................................................................ 263
String Class Exerciser Program ........................................................................ 264
Exerciser Program Sample Run .......................................................................... 266
Lab 5 .................................................................................................................... 267
Summary .............................................................................................................. 268

Chapter 6 References and Argument Passing in C++ ...................................... 273
Variables .............................................................................................................. 275
Argument Passing ............................................................................................... 276
Call-by-Value ....................................................................................................... 277
Reference Declarations ....................................................................................... 278
Call-by-Reference ............................................................................................... 279
Copy Constructor ............................................................................................... 280
Default Copy Constructor ................................................................................... 281
Bug in String Class ............................................................................................. 282
Specification of String Class ............................................................................. 283
Implementation of String Class ......................................................................... 284
### Chapter 7  Operator Overloading, Initialization, and Assignment  .................................. 303
- Operator Overloading ...................................................................................................... 305
- Operator Functions ........................................................................................................ 306
- Concatenation in String Class – Demo .......................................................................... 307
- Semantics of return ........................................................................................................ 310
- Returning a Temporary Object .................................................................................... 311
- Returning a Reference .................................................................................................. 312
- Initialization vs. Assignment ......................................................................................... 313
- Semantics of Assignment .............................................................................................. 314
- Assignment .................................................................................................................... 315
- Assignment Bug ............................................................................................................. 316
- Overloading = ................................................................................................................ 317
- String Assignment Demo ............................................................................................ 318
- Review of this Pointer .................................................................................................. 319
- Type Conversions ......................................................................................................... 320
- Conversion by Construction ......................................................................................... 321
- Conversion Examples ................................................................................................... 322
- Conversion Example Output ........................................................................................ 323
- Overloading Cast Operator .......................................................................................... 324
- Test Program ................................................................................................................ 325
- Lab 7 .............................................................................................................................. 326
- Summary ....................................................................................................................... 327

### Chapter 8  Scope and Access Control ........................................................................ 333
- Scoping in C++ ................................................................................................................ 335
- Block and Function Scope ............................................................................................. 336
- File and Global Scope .................................................................................................... 337
- Class Scope ..................................................................................................................... 338
- Constant Types and Scope ........................................................................................... 339
- Enumeration Types ....................................................................................................... 340
- Enumeration Types and Class Scope .......................................................................... 341
- :: for Global Data ......................................................................................................... 342
- Static Class Members ................................................................................................... 343
- Initialization of Static Member .................................................................................... 344
- Static Function Class Members ................................................................................... 345
- Static Member Example ............................................................................................... 346
- Access Control .............................................................................................................. 347
Module 1:  

Introduction to C++

Top Level Directory:  OIC\IntroCpp
IntroCpp
Chapter 1

First C++ Programs
First C++ Programs

Objectives

After completing this unit you will be able to:

• Write a basic “Hello, World” program in C++.
• Compile and run C++ programs in your local development environment.
• Describe the basic structure of C++ programs.
• Perform output in C++.
• Use variables and simple expressions in C++ programs.
• Write C++ programs that can perform simple calculations.
• Perform input in C++.
Hello, World

- Whenever learning a new programming language, a good first step is to write and run a simple program that will display a single line of text.
  - Such a program demonstrates basic structure of language, including output.
  - You must learn the pragmatics of compiling and running the program.

- Here is “Hello, World” in C++.
  - See Demos\Hello\Hello.cpp, backed up in Hello in the current chapter directory (Chap01).

```cpp
// Hello.cpp

#include <iostream>
using namespace std;

int main()
{
    cout <<"Hello, World" << endl;
    return 0;
}
```
Compiling and Running

• The exact commands for compiling and running your program depend on your system.
  – Your instructor will show you how to compile and run in your environment.

• For example, with the GNU C++ compiler on Unix systems you may do the following:

  g++ Hello.cpp -o Hello

  – Depending on your compiler, the exact syntax may vary, and you may also need a separate linking step.
  – With GNU C++ this creates an executable file called Hello, which you may run as follows:

    ./Hello

    – The program will now execute, and you should see displayed the greeting. That’s all there is to it!

Hello, World

• With an Integrated Development Environment (IDE), you may compile, debug, and run your programs all within the same easy-to-use environment.

  – This course provides an optional lab distribution with supplied project files for a popular IDE, Visual C++ 2005 (which is a free download from Microsoft).
C++ Program Components

• **Statement**
  - An expression terminated by `;`
  - Smallest independent unit of a C++ program

• **Function is a logical grouping of statements containing:**
  - Return type
  - Function name
  - Argument list
  - Function body

• **main()**
  - Every C++ program contains a function called `main`
  - Execution begins with first statement of `main`
  - An integer is returned to the operating system, 0 on success, non-zero if an exception occurred

• **Comments**
  - Single line comments `// ...`
  - Block comments `/* ... */`

• **Header file for bringing in libraries**

  ```
  #include <iostream>
  using namespace std;
  ```
Another C++ Program (Preview)

/*   Triangle.cpp
 *  
 *  Calculate area and hypotenuse of right triangle
 */

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

int main()
{
    double   a, b, c;
    double   area;
    cout << "first leg: ";
    cin >> a;
    cout << "second leg: ";
    cin >> b;
    if ((a <= 0) || (b <= 0))
    {
        cerr << "Illegal input\n";
        return 1;
    }
    c = _hypot(a, b);
    area = .5 * a * b;
    cout << "hypotenuse = " << c << '\n';
    cout << "area = " << area << '\n';
    return 0;
}
I/O in C++

- Input and output are not part of the C++ language but are supported by a standard library known as the `iostream` library.

- To use the library provide an `include` directive.

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

- Symbols in the library such as `cout` are in the “namespace” `std`, and to make it easy to use these symbols, provide the following statement:

```cpp
using namespace std;
```

- Output operator is `<<`, and standard output device is `cout`.

```cpp
cout << "Hello, world" << endl;
```

  - `endl` writes an end-of-line and flushes the output buffer

- Input operator is `>>`, and standard input device is `cin`.

```cpp
cin >> num;  // reads num
```
Exercise

• Take a few minutes to add two more lines of code to your Hello.cpp program.
  – Print out the phrase “My name is XXXX” (use your own name).
  – Then print out “Goodbye”.

• Save your file, compile the program, and run it.
  – Your output should be something like this:

Hello, World
My name is Bob
Goodbye
Answer

// Hello.cpp

#include <iostream>
using namespace std;

int main()
{
    cout <<"Hello, World" << endl;
    cout <<"My name is Bob" << endl;
    cout <<"Goodbye" << endl;
    return 0;
}

– See HelloGoodbye in the chapter directory.
Variables

• In C++ you can define *variables* to hold data.

• Variables represent storage locations in memory.

• In C++ variables are of a specific data *type*.
  
  – Some common types are `int` for integers and `double` for floating point (decimal) numbers.

• You must *declare* variables before you can use them.

  – A variable declaration reserves memory space for the variable and may optionally specify an initial value.

    ```c
    int kilo = 1024; // reserves space and assigns // an initial value
    int mega; // reserves space but does // not initialize
    ```

  – If an initial value is not specified, the starting value is undefined, which may cause unpredictable results.
Expressions

• You can combine variables and constants (or “literals”) via operators to form expressions.

• Examples of operators include the standard arithmetic operators:

  + addition
  - subtraction
  * multiplication
  / division

• Here are some examples of expressions:

  kilo * 1024
  (fahrenheit - 32) * 5 / 9
  3.1416 * radius * radius
Assignment

• You can assign a value to a variable by using the = symbol.
  - On the left hand side is a variable.
  - On the right hand side is an expression.
  - The expression is evaluated and its value is assigned to the variable on the left.
  - Assignment is a statement and must be terminated by a semicolon:

    \[
    \text{mega} = \text{kilo} \times 1024; \\
    \text{celsius} = (\text{fahrenheit} - 32) \times \frac{5}{9}; \\
    \text{area} = 3.1416 \times \text{radius} \times \text{radius};
    \]

• Note that the same variable can be used on both sides of an assignment statement:

    \[
    \text{int item} = 5; \\
    \text{int total} = 30; \\
    \text{total} = \text{total} + \text{item};
    \]

  - The expression \text{total} + \text{item} evaluates to 35, using the old value of \text{total}, and this value is assigned to \text{total}, creating a new value.
Using C++ as a Calculator

• You can easily use C++ to perform calculations by adding code to the *main* function.
  – Declare whatever variables you need.
  – Create expressions and assign values to your variables.
  – Print out the answer using `cout <<`.

• Note that you can get several items on the same line of output, including mixed string and numerical data, simply by not using `endl` until the end.

  ```
  cout << "celsius = " << celsius << endl;
  ```
Sample Program

- This program will convert from Fahrenheit to Celsius.
  - See Convert\Step1.

```cpp
// Convert.cpp - Step 1
// Program converts a hardcoded temperature in Fahrenheit to Celsius

#include <iostream>
using namespace std;

int main()
{
    int fahrenheit = 86;
    int celsius = (fahrenheit - 32) * 5 / 9;
    cout << "fahrenheit = " << fahrenheit << endl;
    cout << "celsius = " << celsius << endl;
    return 0;
}
```
Input in C++

- Our first Convert program is not too useful because the Fahrenheit temperature is hardcoded.
  - To convert a different temperature you would have to edit the source file and recompile.

- What we really want to do is allow the user of the program at runtime to enter a value for the Fahrenheit temperature.

- Fortunately, basic input in C++ is quite easy:
  - Declare a variable
  - Use `cin >>` to read data into the variable

- Basic I/O is an area where C++ is actually easier than C.
Echo Program

- We illustrate interactive input by a simple “echo” program:
  - Program prompts user for a number and then prints back what the user entered
  - See Echo.

// Echo.cpp
//
// Prompt user for an integer and echo it back

#include <iostream>
using namespace std;

int main()
{
    int num;
    cout << "Enter an integer: ";
    cin >> num;
    cout << "You entered " << num << endl;
    return 0;
}
Lab 1

C++ Programs for Calculation

In this lab you modify or implement several C++ programs to perform calculations. You need to perform input, do a calculation, and output your result. Do as many of these exercises as time permits. If you have extra time, do some of the optional experiments suggested in some of the exercises, or make up some experiments on your own.

Detailed instructions are contained in the Lab 1 write-up at the end of the chapter.

Suggested time: 30 minutes
Summary

• Every C++ application has a function `main`, which is the entry point into the application.

• C++ programs are compiled into object code, which is linked with library code to form an executable.

• The `iostream` library includes a built in output stream `cout`.

• Expressions in C++ are formed from constants, variables and operators.

• With the assignment statement you can assign a value computed by an expression to a variable.

• Input in C++ is done through the input stream `cin`. 
Lab 1

C++ Programs for Calculation

Introduction

In this lab you modify or implement several C++ programs to perform calculations. You need to perform input, do a calculation, and output your result. Do as many of these exercises as time permits. If you have extra time, do some of the optional experiments suggested in some of the exercises, or make up some experiments on your own.

Suggested Time: 30 minutes

Root Directory: OIC\IntroCpp

Directories:

- Labs\Lab1\Convert (Exercise 1 work)
- Chap01\Convert\Step1 (Backup of Exercise 1 starter files)
- Chap01\Convert\Step2 (Exercise 1 answer)
- Chap01\Convert\Step3 (Exercise 1 answer to optional)
- Labs\Lab1\Circle (Exercise 2 work)
- Examples\Circle\Step1 (Exercise 2 answer)
- Examples\Circle\Step2 (Exercise 2 answer to optional)

Exercise 1. Fahrenheit to Celsius Conversion

Examine the code of the starter program. Build and run. Notice that the Fahrenheit temperature to be converted is hardcoded. Modify the program to prompt the user for a Fahrenheit temperature, read in the value entered by the user, and print out the result.

The starter program uses int as the data type for temperatures. An optional experiment is to use double as the data type. Could you input the Fahrenheit temperature as an int and calculate the Celsius temperature as a double?

Exercise 2. Calculate the Area of a Circle

Write a C++ program to calculate the area of a circle. Prompt the user for the radius. Read in the value entered by the user, calculate the area of the circle, and print out the result. For pi use the approximation 3.1416. What is an appropriate data type to use for radius and area?

As an optional experiment use the math library <cmath> and the arctangent function atan to obtain a more accurate value of pi. (Use fact that the arctangent of 1 is equal to pi / 4).

A final optional experiment is to capture the output data in a file. This is easy to do at the command line by using the > to “redirect” output to a file.
circle > circle.txt

When using this technique the prompts are written to the output file and not displayed on the screen. Hence you need to know what to type! If you type 10 for the radius and then press Enter, your output file should look like:

radius: Using, 3.1416, area = 314.16
Using, arctangent, area = 314.159
Chapter 5

Functions and Program Structure
Functions and Program Structure

Objectives

After completing this unit you will be able to:

- Break down program into discrete tasks and declare and define functions to perform each task.
- Declare functions using function prototypes.
- Invoke functions with proper parameters.
Basics of Functions

- Functions allow programmers to break down complex programs into manageable units.

- When creating functions, we must be clear about the function itself and how it interacts with other functions.

- A function is invoked by giving the function name and a comma-separated actual argument list in parentheses.

- The function result is returned to the caller using return statement:

  ```
  return  expression;
  ```

- When no value is returned, the return type of the function is void.

- The calling function has the option of ignoring the returned value.

- The body of the function is enclosed in braces and it is referred to as a block.
**Example**

```c
int max (int a, int b)  // function definition.
{
    if (a > b)
        return  a;
    else
        return  b;
}
```

```c
maxvalue = max(15,20);  // function invocation.
```

```c
max ("turbo", "C++");
    // invalid invocation of max. Implicit
    // conversion of arguments to int is
    // is not possible.
```

```c
max (2358);  // invalid invocation of max.
    // ncorrect number of arguments.
```
Function Prototypes and Type Checking

• Function prototype gives the number and type of each argument and the type of return value:

```cpp
int max (int x, int y);
```

• When a function is used before it is defined or it is used in a file other than where it is defined, function prototype is used to declare the function.

• Function prototypes can be placed in a header file and this header file can be included in all the files where the function is used.

• Since C++ is a strongly typed language, the compiler checks both argument list and the type of returned value in every function call.

• Function prototype provides the necessary information to the compiler for type-checking.

• Implicit type conversion of the arguments is done if possible. If it is not possible or if the number of arguments is incorrect, compiler generates an error.
Returning Value

• A function that returns a value must include its data type in the header line before the function name:

```c
float max (float x, float y);
// max function returns a float.
double max (double x, double y);
// max function returns a double
```

• To return a value, function must use a return statement.

```c
return; // no value is returned
return (expression); // expression that produces //the appropriate data type.
```

• If return type is missing in the header of function definition, default return type is int.

• A return type of void is used if a function does not return a value.

• An array or a function cannot be used as a return type. But a return type of pointer to an array or function is allowed. A return statement terminates the execution of the current function and passes the control to the calling function.

• If no value is returned, return statement is optional.

• It can be used to do a premature return. An implicit return is automatically done on completion of a function.
Argument Passing

- Calling function passes data to the called function through argument list.

- The longer the argument list, the greater the likelihood of error in argument passing.

- If a function has an empty argument list, it is indicated by empty parentheses.

- Different argument names can be used in the function prototype and in the actual function definition.

- Each formal argument is allocated sufficient storage based on its type within the activation record.

- The formal arguments get initialized when the function is invoked.

- The default initialization method is to copy the value of the actual argument into the storage of formal argument pass-by-value.

- When pass-by-value is used, modification of formal arguments within the function will not affect the actual arguments.

- On returning from the function, storage allocated for formal arguments is released and hence their values are lost.
Pass-by-Reference

- Pass-by-value mechanism can be overridden by declaring reference argument. Reference arguments receive the address of actual arguments, instead of their value.

- Reference arguments allow the function to return more than one result:

```cpp
void readdata (int &arg1, int &arg2)
   // arg1 & agr2 are reference arguments
{
    cout << "Enter two integer numbers\n";
    cin >> arg1 >> arg2;
}
readdata (a1, a2);  // function invocation.
```

- Reference arguments are convenient for large objects to void copying the entire object on each function call.

- Care must be taken to prevent unwanted modification of these objects.
Alternative to Pass-by-Reference

- Another alternative is to declare formal parameters as pointers and pass the address of actual arguments when the function is invoked:
  - Pointers will be discussed in the next chapter
  - Here is a preview!

```cpp
void readdata (int *arg1, int *arg2)
    // arg1 & arg2 are pointers
{
    cout << "Enter two integer numbers\n";
    cin >> *arg1 >> *arg2;
}
readdata (&a1, &a2);  // function invocation.
```
External Variables

- External variables are defined outside of a function as opposed to local variables and shared by all the functions that come after the definition in the same file.

- All references to them have to be made using the same name.

- Since external variables are accessible throughout the program, they provide a means for passing data between calling and called functions.

- Careless use of external variables could introduce hidden bugs.

- To expand the scope of these extern variables to multiple files, an extern declaration is necessary.
EXTERN EXAMPLE

See ExternDemo.

// file1.cpp
//
// This file defines external variable

int a = 5;

// file2.cpp
//
// This file accesses variable defined in file1.cpp

#include <iostream>
using namespace std;

extern int a;

int main()
{
    cout << "a = " << a << endl;
    return 0;
}

• Extern declaration statement does not allocate new storage for the variable. The purpose of this declaration is to inform the compiler that it already exists and can be used.
Block Structure

- Variables can be defined in a block structured manner within a function:
  
  - Variables defined as part of a compound statement are known only within that block.
  
  - They exist only until the right brace that terminates that block is encountered.
  
  - Variables declared in a block will hide all the variables with the same name in outer blocks.

```cpp
// BlockDemo.cpp

#include <iostream>
using namespace std;

int main()
{
    int i = 5;
    if (i >= 0)
    {
        int i = 10;
        cout << "value of i in inner block is "
             << i << endl;
    }
    cout << "value of i in outer block is "
         << i << endl;
    return 0;
}
```

**Output**

value of i in inner block is  10
value of i in outer block is  5
Scope Rules

- The scope of a variable is the range of program statements over which it is known.

- A variable can have either local scope or global scope:
  - A variable with local scope is defined inside a function and its scope is limited to that function.
  - Multiple declaration of variable with the same name is allowed in different functions since they are independent of each other.
  - Variables with global scope are declared outside of a function. Their scope can be extended to functions in other files using `extern` declaration.
  - Scope of a global variable can be restricted to a file by making it a static variable.
Scope Example

- See ScopeDemo.

// file1.cpp

#include <iostream>
using namespace std;

int num1;       // global scope
static int num2; // scope is limited to file1
int f1();      // function prototype

int main ()
{
    int num3 = 30;   // local to main
    num1 = 10;
    cout << "first call of f1 returns: " << f1()
        << "\n";
    cout << "value of num3 after calling f1: "
        << num3 << "\n";
    cout << "second call of f1 returns: " << f1()
        << "\n";
    return 0;
}
Scope Example (Cont’d)

//file2.cpp

extern int num1;  // extended the scope of num1 to
                // file2
int f1()
{
    int num3;  // local to f1
    static int num4 = 0;  // retains its value
                        // between calls
    num4 += num1;
    num3 = num4;
    return num4;
}

**Output**

first call of f1 returns: 10
value of num3 after calling f1: 30
second call of f1 returns: 20
Header Files

- In multiple file programs, shared declarations and definitions are kept in a header file and included as necessary in files that make reference to these.
  - User-defined header files typically have `.h` extension, and the file name is shown in quotes.
    ```
    #include "MyHeader.h"
    ```
  - System include files do not have this extension, and the file name is shown in angle brackets.
    ```
    #include <iostream>
    ```
- Based on the size of the program, it may be better to distribute shared information in multiple header files.
- `<iostream>` is an example of a header files needed to use library functions in programs.
- We will use header files extensively in the second part of the course.
Preprocessor Directives

- Directives are processed before compiling a program.
- The program that processes the directive is called the preprocessor.
- Directives start with a "#" symbol in column one of a line.

- **#include** directive is used to read a header file:

  ```
  #include <iostream> // from standard library.
  #include "declare.h" // user defined file.
  ```

- **#define** directive is used for macro substitution:

  ```
  #define macroname replacement information
  #define SIZE 100  // wherever SIZE appears, // 100 is substituted.
  ```

  - Arguments are allowed in macro definitions. In this case, use of macros will look like function calls. But macros will expand to in-line code.

  - If expressions are used as arguments in macro calls, parentheses must be used to enforce correct order of evaluation.
Conditional Compilation

- Conditional substitution directive allows the inclusion or exclusion of a group of statements depending on the value of a condition:

  ```cpp
  #ifdef name
    // if name evaluates to nonzero, the
    // statements following will be made
    // part of the program; otherwise skipped
  statement;
  statement;
  #endif

  #ifndef name
    // if name evaluates to zero (which will
    // be the case if symbol has not been
    // defined), the
    // statements following will be made
    // part of the program; otherwise skipped
  statement;
  statement;
  #endif
  ```
Recursion

- A function that calls itself, directly or indirectly is called a recursive function.

- A recursive function should always provide a terminating condition.

- Recursive functions are likely to run slower than iterative functions due to the overhead involved.

- But the code will be compact:

```c
unsigned long factorial (int n) // n >= 0
{
    unsigned long  fact;
    if  ( n == 0) // terminating condition.
        fact = 1;
    else
        fact = n * factorial(n-1);
            // recursive call of factorial
    return  fact;
}
```
Lab 5

Using Functions

In this lab you will provide new implementations of two earlier programs, this time making use of functions. You will also get some additional practice with loops and if tests. The first exercise involves determining whether a year is a leap year, and your function will return a single bool value. In the second exercise, your function needs to return two different values (count of odd and count of even), and so you must make use of reference arguments. In the third exercise you use global variables for passing the data, and the function is in a separate file from the main program.

Detailed instructions are contained in the Lab 5 write-up at the end of the chapter.

Suggested time: 60 minutes
Summary

- A function definition consists of a function header, local declarations (if any), and body of the function.

- A function usually returns a single value through return statement.

- If it does not return a value, the type of the function is void. Default type of int is assumed, if type is left out in the header.

- A function is invoked using function name and correct number of arguments of the proper type.

- Arguments can be passed to the function using pass-by-value or pass-by-reference mechanism.

- Pass-by-reference mechanism allows the function to return multiple values.

- Global variables are variables declared outside of a function. All the functions that appear after this definition have access to these variables.

- The scope of a static global variable is limited to one file. Global variables can be shared by multiple files using extern declaration.
Lab 5

Using Functions

Introduction

In this lab you will provide new implementations of two earlier programs, this time making use of functions. You will also get some additional practice with loops and if tests. The first exercise involves determining whether a year is a leap year, and your function will return a single `bool` value. In the second exercise, your function needs to return two different values (count of odd and count of even), and so you must make use of reference arguments. In the third exercise you use global variables for passing the data, and the function is in a separate file from the main program.

Suggested Time: 60 minutes

Root Directory: OIC\IntroCpp

Directories:
- Labs\Lab5\LeapYear
- Chap03\LeapYear\Step2 (backup copy of starter files)
- Chap05\LeapYear\Step3 (answer to first part)
- Chap05\LeapYear\Step4 (answer to optional part)
- Labs\Lab5\OddEven (do Exercise 2 - 4 work here)
- Chap04\OddEven\Step1 (backup copy of starter files)
- Chap05\OddEven\Step2 (answer to Exercise 2)
- Chap05\OddEven\Step3 (answer to Exercise 3)
- Chap05\OddEven\Step4 (answer to Exercise 4)

Instructions

Exercise 1. Leap Year Test as a Function

1. Build the starter project and make sure that the program is working.

2. Create a function `Leap` that takes an `int` parameter and returns a `bool`. This function should be placed before `main`.

3. Copy the code that tests whether a year is a leap year into your new function. Modify it so that it returns `true` or `false` instead of printing out a message.

4. Modify your main program so that it performs the test by calling the `Leap` function.(Step3).

5. (Optional) Simplify your code for the `Leap` function by using a sequence of `else if` clauses in place of a single complicated `bool` expression. You will know unequivocally that a year is a leap year if it is divisible by 400. If it fails this test but it
is divisible by 100, then it is definitely not a leap year. If it fails this test but is
divisible by 4, it is a leap year. Finally, if it fails all of these tests, it is not a leap year.

6. Write a loop in the main program to test for a series of years, with 0 for end-of-file.
(Step4).

Exercise 2. Counting Odd and Even Numbers in a Function, Using Global Data

In the following exercises you will work with the program that reads in numbers and
counts how many are odd and even.

1. Build the starter project and make sure that the program is working.

2. Create a function Count that does all the work of reading in the numbers and counts
how many are odd and even (copy the code from main). Declare the counters odds
and evens as global int variables outside of any function.

3. In the main program temporarily keep the statements that declare and initialize odds
and evens. Call the Count function and then print out the results. Run the program
and enter some data. You should observe 0 for both counts. Why?

4. The odds and evens variables in the main function hide the global variables. To
remove this problem, comment out the declarations inside main. You will still need
to initialize the counts to 0. You can do this with one statement

   odds = evens = 0;

5. Build and run the program. It should now work (Step2).

Exercise 3. Counting in a Function, Using External Global Data

In this exercise you will move main to a separate file and access the global data using
eextern.

1. Create a new file called TestOddEven.cpp, and move the main function into it. You
will also need to include iostream.h.

2. Add an extern declaration for odds and evens.

3. Add a prototype for Count function.

4. Build the project consisting of the two files OddEven.cpp and TestOddEven.cpp.
The program should work as before (Step3).
Exercise 4. Counting in a Function, Passing Data by Reference

In this exercise you will continue with main in a separate file but will remove the global external data. Instead you will use pass by reference to get the count data back.

1. In OddEven.cpp remove the global variables odds and evens.

2. Change the definition of the function Count so that it takes reference parameters odds and evens. The skeleton of the function should thus look like:

```cpp
void Count(int& odds, int& evens)
{
}
```

3. In TestOddEven.cpp remove the extern for odds and evens.

4. Change the prototype of Count to match the way it is now defined.

5. In the body of main declare the int variables odds and evens, but you do not have to initialize them (that happens in Count). Leave the rest of the code unchanged.

6. Build the two file project. The program should continue to work as before (Step4).
Module 2:

Object-Oriented C++
Programming Fundamentals

Top Level Directory: OIC\FndCpp
FndCpp
Chapter 1

Concepts of Object-Oriented Programming
Concepts of Object-Oriented Programming

Objectives

After completing this unit you will be able to:

• Describe the basic concepts of an object and a class.

• Explain how the object model provides the framework for abstraction, encapsulation and instantiation.

• Define the concept of an abstract data type.

• Define the terms method and message.

• Explain the use of class inheritance for enhancing reusability of code.

• Define the term polymorphism and explain how it can be used to make object oriented programs more flexible and easy to maintain.
Object

- An object is a software entity containing data and related functions as a self contained module:

Bank Account Object

- Objects hold state and specify behavior.
- Objects provide the means for abstraction, encapsulation, and instantiation.
State and Behavior

- An object has data, i.e. a set of properties or attributes, which are its essential characteristics.
  - The state of an object is the value of these attributes at any point in time.

- The behavior of an object is the set of operations or responsibilities it must fulfill for itself and for other objects.

- The data and operations are packaged together:

  ![Data and Operations Diagram]

  - As part of software design, this packaging aids conceptualization and abstraction.
  - Disparate items are turned into a conceptual unit.
Abstraction

- An abstraction captures the essential features of an entity, suppressing unnecessary details.
- All instances of an abstraction share these common features.
- Abstraction helps us deal with complexity.
Encapsulation

- The implementation of an abstraction should be hidden from the rest of the system, or encapsulated.

- Objects have a public and a private side.

- Public side is what the rest of the system knows, while private side implements the public side:

  ![Diagram showing public interface and private]

- Data itself is private, walled off from the rest of the program.

- Data can only be accessed through functions with a public interface.

- There are two kinds of protection:
  - Internal data is protected from corruption.
  - Users of the object are protected from changes in the representation.
Class and Instantiation

- A class groups all objects with common behavior and common structure.

- A class allows production of new objects of the same type. An object is an instance of some class:

```
Class

CAR

Instances (objects)

194AFX
821ZAY
715GVN
```
Abstract Data Types

- Data types describe a set of objects with the same representation.
  - There can be several operations associated with each data type.
  - The data itself is directly accessible to the rest of the program.
  - Data types can be implemented via a typedef in C.

- An Abstract Data Type (ADT) is a data type that "hides" the implementation of its operations:
  - There is a public set of operations.
  - The data itself is private.
  - ADT's can be implemented via a class in C++.
  - ADT's in C++ may have “sugar coating”, such as overloaded operators, that may make their usage appear identical to that of built-in types.
Abstract Data Type Example

- C++ does not have a built-in type for complex numbers.

- But we can create a new abstract data type called Complex with operations such addition, subtraction, multiplication, etc.
  - In C++ we can even use + to represent addition, etc.

- We can then write code like the following:

```cpp
Complex a(3, 5);
Complex b(10, 20);
Complex c;
c = a + b;
// etc.
```

- In later chapter you will learn how to implement abstract data types in C++. 
Methods

- In object oriented software, a function defined for an object is called a method.

- A method is only accessible via its object, while a function is a free standing entity.

- Objects communicate between each other through methods:

  - The terms service, method, message or operation are synonymously used.
Invoking Methods

- A method consists of:
  - Id of the receiving object
  - Name of the method
  - Parameters

```
Bank Account
```

```
Transfer (Money, Account)
```

```
Bank Account
  Object Id

Transfer
  Method name

Money, Account
  Parameters
```
Messages

- Requests for object services are made by "sending a message" to the object.

- Each object responds to a prescribed set of messages that comprise the object's interface.

- "Message" terminology is used in some OOP systems, notably Smalltalk.

- Semantics of "sending a message" and "invoking a method" are identical.
  - There is no implication of asynchronous operation in the OOP concept of sending a message.
Class Inheritance

- Inheritance is the ability of defining a new class by specifying only its difference from another class.

- Derived classes (subclasses) inherit behavior from the base class (superclass).

- Inheritance promotes software refinement:

  ![Class Inheritance Diagram]

  - Text File and Binary File classes inherit all properties of a file such as file protections and file structure. Only information contained in the file is different.
Polymorphism

- Polymorphism is the ability of two or more classes of objects to respond to the same message, each in its own way:
  - An object does not need to know to whom it is sending a message.
  - It just needs to know that many different kinds of objects have been defined to respond to that particular message.
  - The different classes should belong to a common hierarchy:

- The receiving object with its method is determined at run time by a process known as dynamic binding.

- Polymorphism improves software extension as new classes can be added more easily.
Summary

• An object has both state and behavior.

• The object model provides the framework for abstraction, encapsulation and instantiation.

• An abstraction captures the essential features of an entity, suppressing unnecessary details.

• Encapsulation protects internal data from corruption and isolates users of an object from changes in data representation.

• A class is a basis for objects with similar behavior and structure, supporting instantiation.

• An abstract data type has private data and a public set of operations.

• Class inheritance supports reuse by providing the capability to define new objects by specifying differences and extensions to an existing object.

• Polymorphism is the ability of two or more classes of objects to respond to the same message, each in its own way, making object oriented programs more flexible and extensible.
Chapter 9

Introduction to Inheritance
Introduction to Inheritance

Objectives

After completing this unit you will be able to:

• Use inheritance to model your problem domain and achieve greater code reuse.

• Use C++ class derivation to implement inheritance.

• Use public, protected and private to control access to class members.

• Use an initialization list for proper base class initialization and embedded member initialization.

• Determine order of invocation of constructors and destructors.

• Distinguish between use of inheritance and composition.
Inheritance Concept

• Inheritance is a key feature of the object oriented programming paradigm.
  – You abstract out common features of your classes and put them in a high level base class.
  – You can add or change features in more specialized derived classes, which "inherit" the standard behavior from the base class.
  – Inheritance facilitates code reuse and extensibility.

• Consider Employee as a base class, with derived classes WageEmployee and SalaryEmployee.
  – All employees share some attributes, such as name.
  – Wage employees and salaried employees differ in other respects, such as in how their pay is computed.
Inheritance Example

Employee

WageEmployee

SalaryEmployee

Name

Employee

Name

SalaryEmployee

Salary

Name

WageEmployee

Wage

Hours
Inheritance in C++

• Inheritance is implemented in C++ by a mechanism known as *class derivation*:

```cpp
class DerivedClass : public BaseClass {
    . . .
};
```

• Base class must be declared prior to the derived class.

• *DerivedClass* can use all public (and protected) members of *BaseClass*, but it does not have any special access to the private members of *BaseClass*.
  
  − If a derived class did have access to private members of its base class, the access security could be defeated simply by deriving a class.
Employee Example

class Employee
{
public:
    enum {MAXNAME = 20};
    Employee(const char *name = "")
        {strcpy(m_name, name);} // Constructor
    Employee(const char *name)
        {strcpy(m_name, name);} // Parameterized constructor
    void SetName(const char *name)
        {strcpy(m_name, name);} // Method to set name
    const char* GetName() const {return m_name;} // Method to get name

protected:
    char m_name[MAXNAME];
};
class SalaryEmployee : public Employee
{
public:
    SalaryEmployee(const char *name, int salary)
        : Employee(name) {m_salary = salary;}
    void SetSalary(int salary) {m_salary = salary;}
    int GetSalary() {return m_salary;}

private:
    int m_salary;
};
class WageEmployee : public Employee
{
public:
    WageEmployee(const char* name, int hours, int wage)
        : Employee(name)
        {m_hours = hours; m_wage = wage;}
    void SetHours(int hours) {m_hours = hours;}
    int GetHours() {return m_hours;}
    void SetWage(int wage) {m_wage = wage;}
    int GetWage() {return m_wage;}

private:
    int m_hours;
    int m_wage;
};
Employee Test Program

- See Employee\Step0 in the chapter directory.

```cpp
#include "Trace.h"
#include "Employee.h"

int main()
{
    SalaryEmployee sally("Sally", 500);
    WageEmployee wally("Wally", 40, 10);

    Trace("Sally's name = ", sally.GetName());
    Trace("Sally's salary = ", sally.GetSalary());
    Trace("Wally's name = ", wally.GetName());
    Trace("Wally's hours = ", wally.GetHours());
    Trace("Wally's wage = ", wally.GetWage());
    return 0;
}
```

**Output:**

Sally's name = Sally  
Sally's salary = 500  
Wally's name = Wally  
Wally's hours = 40  
Wally's wage = 10
Protected Members

- So far we have seen two access privileges: public and private.

- Class derivation introduces a different kind of user: the derived class.
  - SalaryEmployee is derived from Employee but has no special privileges to access the private members of Employee.

- To allow special privilege for this user, protected access privilege is provided as the third type of access privilege.
  - Since m_name is declared as protected in the Employee base class, the derived class could access it, but classes not derived from Employee could not.

- Members specified as protected become public to the derived class, but remain private to all other classes and program.

- Rules for private and public are same for the derived classes.
Base Class Initializer List

- When the base class constructor requires arguments, the arguments are passed via an "initialization list"

```cpp
class SalaryEmployee : public Employee
{
  public:
    SalaryEmployee(const char *name, int salary)
    {
      m_salary = salary;
    }
};
```

- An initializer list will be used in the constructor to pass arguments to the base class constructor for `Employee`:

```cpp
class SalaryEmployee : public Employee
{
  public:
    SalaryEmployee(const char *name, int salary) :
      Employee(name)
    {
      m_salary = salary;
    }
};
```
Composition

- Another way for a new class to reuse code of an old class is to simply create an object of the old class inside the new class.
  - This technique is called composition because one class is composed of objects of other classes.

- For example, class Employee could use a String object to represent employee name:

```cpp
class Employee
{
public:
    Employee(const char *name = "") { m_name = name; }
    void SetName(const char *name)
    { m_name = name; }
    const char* GetName() const { return m_name; }
private:
    String m_name;
};
```

- We want the argument name to be used for initializing the member object m_name.

- If you don't do anything special, the compiler will generate code to implicitly call the default constructor for the member object before constructing the containing object.
Member Initializer List

- A better approach is to use a "member initializer list", which has similar syntax to a base class initializer list:

```cpp
Employee::Employee(const char* name) : m_name(name) {
}
```

- This syntax causes the String class constructor to be invoked with the argument name.

- The String class constructor is called first before the Employee constructor starts executing.

- The member object get data assigned exactly once.

- The same syntax can also be used for built-in data types, and member object initialization and base class initialization can be combined:

```cpp
WageEmployee::WageEmployee(const char* name, int hours, int wage) : Employee(name),
    m_hours(hours), m_wage(wage)
{
}
```
Order of Initialization

- C++ has a defined order for the construction and destruction of base class objects, derived class objects, and member objects:

```cpp
class DerivedClass : public BaseClass
{
public:
    member1;
    member2;
};
```

- The order of construction is:
  
  Constructor of  `BaseClass`
  Constructor of  `member1`
  Constructor of  `member2`
  Constructor of  `DerivedClass`

- Destructors are invoked in exact reverse order.

- It is important to know this order in cases where there are interdependencies among classes.
  
  - You should avoid a situation where an object gets prematurely destroyed while another object refers to its data.
Inheritance vs. Composition

- Inheritance and composition are both code reuse techniques in which data from one class is contained within another class.
  - When do you prefer one technique over the other?
- Inheritance is used when an “Is-A” relationship exists:
  - A SalaryEmployee is a Employee
  - The derived class supports the same interface as the base class, plus some additional features
- Composition is used when a “Has-A” relationship exists:
  - Employee has a String as a data member to represent the name
  - Composition is suitable when you want the features of another class but not its interface
Lab 9A

An Employee Class Hierarchy

In these exercises you will work with the Employee class hierarchy to reinforce basic inheritance concepts. You will practice the initialization of both base class and embedded class objects and verify the order of invocation of constructors and destructors. You will define a function in the base class that is overridden differently in derived classes.

Detailed instructions are contained in the Lab 9A write-up at the end of the chapter.

Suggested time: 30 minutes.
Lab 9B

More Practice with Inheritance

In this lab you will gain some more practice working with inheritance. You will work with a bare bones example in which there is a base class, a derived class, and an embedded member object.

Detailed instructions are contained in the Lab 9B write-up at the end of the chapter.

Suggested time: 15 minutes.
Summary

- C++ has special features to allow class inheritance, which allows you to better model your problem domain and to achieve greater code reuse.

- Members of a base class are also members of derived classes.

- Protected members of a base class can be accessed by derived classes but not by any other classes.

- Initialization lists can be used to properly initialize member objects and base class objects.

- The order of invoking constructors is from the base class to the derived class.

- Inheritance models “Is-A” relationships and composition models “Has-A” relationships.
Lab 9A

An Employee Class Hierarchy

Introduction

In these exercises you will work with the Employee class hierarchy to reinforce basic inheritance concepts. You will practice the initialization of both base class and embedded class objects and verify the order of invocation of constructors and destructors. You will define a function in the base class that is overridden differently in derived classes.

Suggested Time: 30 minutes.

Root Directory: OIC\FndCpp

Directories: Labs\Lab9A\Employee (do your work here)
             Chap09\Employee\Step1 (backup copy of starter files)
             Chap09\Employee\Step2 (contains lab solution)

Files to Modify: DemoEmp.cpp
                 Employee.h
                 Employee.cpp:

Instructions

1. In the working directory there are files for an “Employee” project demonstrating inheritance from an Employee class and composition with the String class. Study the code in the files Employee.h and in the demo program DemoEmp.cpp. Make sure you understand the syntax used, including initializer lists with inline code in the constructors. Then answer the following questions. Then build and run the program to verify your answers. You may want to redirect the output to a file, which you can save and examine.

   a. In the constructors for Employee a character pointer is used as an input argument. As the code is written, what member of the String class is used to initialize m_name.

   b. If instead the following code was used, how would your answer change? Which code is better and why?

      Employee(const char *name = "")
      {
        Trace("Employee::Employee(const char*)");
        m_name = name;
      }

      Employee(const char *name = ")
      {
        Trace("Employee::Employee(const char*)");
        m_name = name;
      }
c. In the **GetName** function of the **Employee** class, a **String** object is used as the return value. What member of the **String** class is used to convert the **String** object to a character pointer?

d. What is the order of constructor and destructor calls when the test program is run?

2. Implement a **GetPay** function for each employee class that will calculate the pay appropriately for an employee. Add code to **DemoEmp.cpp** to calculate and print out the pay for the employees Sally and Wally. What is an appropriate way to implement **GetPay** in the base class? (This is a preview of some ideas we will discuss in the next chapter!)
Lab 9B

More Practice with Inheritance

Introduction

In this lab you will gain some more practice working with inheritance. You will work with a bare bones example in which there is a base class, a derived class, and an embedded member object.

Suggested Time: 15 minutes.

Root Directory: OIC\FndCpp

Directories: Labs\Lab9B\Inheritance (do your work here)
             Chap09\Inheritance\Step1 (backup copy of starter files)
             Chap09\Inheritance\Step2 (answer)

Instructions

1. Run the starter project and observe the order of constructors and destructors for Base, Derived and Member. Make sure you understand the operation of this program.

2. Add code to initialize the embedded Member object in Derived to be initialized to 99.

3. Modify the implementation of Derived::Print so that you do not call Base::Print but instead access xx directly. What other change do you have to make to enable this solution? Do you think this solution is better or worse than the original?
Lab 9A  Answers

1. a) The constructor `String::String(const char*)` does the initialization.

b) The overloaded operator `String::operator=(const char*)` does the initialization. The code as originally written is better, because the embedded String object gets initialized directly by one call to a constructor. The second version is less efficient, because first the default String constructor is invoked to create an “empty” String object, and then the overloaded assignment operator is called.

c) The overloaded cast operator `String::operator const char* () const` does the conversion.

d) The order of constructors is embedded member, base class, derived class. The destructors are invoked in the reverse order. Sally (SalaryEmployee) is constructed first and destroyed last. The output from running the program shows the invocation of constructors and destructors:

```cpp
class String { 
public:
  String(const char *str) 
  { // constructor 
  } 

private:
  char *m_str;
};
```

```cpp
class Employee { 
public:
  Employee(const char *str) 
  { // constructor 
  } 

private:
  char *m_str;
};
```

```cpp
class SalaryEmployee : public Employee { 
public:
  SalaryEmployee(const char *, int) 
  { // constructor 
  } 

private:
  char *m_str;
};
```

```cpp
class WageEmployee : public Employee { 
public:
  WageEmployee(const char *, int, int) 
  { // constructor 
  } 

private:
  char *m_str;
};
```

```cpp
char const *Sally's name = Sally;
char const *Wally's name = Wally;
int Sally's salary = 500;
int Wally's hours = 40;
int Wally's wage = 10;
```

```cpp
Sally's salary = 500;
Wally's hour = 40;
Wally's wage = 10;
```

```cpp
Sally's name = Sally;
Wally's name = Wally;
Sally's salary = 500;
Wally's hours = 40;
Wally's wage = 10;
```

```cpp
WageEmployee::~WageEmployee();
Employee::~Employee();
String::~String();
SalaryEmployee::~SalaryEmployee();
Employee::~Employee();
String::~String();
```

2. If you have a GetPay member function in the base class, about all you can do is assign some default pay to a “generic” employee. In the next chapter we will see that Employee can be an “abstract” base class, and we can specify a function that we don’t have to implement. The highlighted code shows implementation of GetPay for the two “concrete” classes SalaryEmployee and WageEmployee, and the usage in the demo program.
// Employee.h

#ifndef _EMPLOYEE_H
#define _EMPLOYEE_H

#include "Strn.h"

class Employee {
public:
    Employee(const char *name = "") : m_name(name) {
        Trace("Employee::Employee(const char*)");
    }
    ~Employee() {
        Trace("Employee::~Employee()");
    }
    void SetName(const char *name) {
        m_name = name;
    }
    const char* GetName() const {return m_name;}
    int GetPay() {return 100; } // arbitrary default value

private:
    String m_name;
};

class SalaryEmployee : public Employee {
public:
    SalaryEmployee(const char *name = "", int salary = 0) : Employee(name), m_salary(salary) {
        Trace("SalaryEmployee::SalaryEmployee(const char *, int)" gn3,f);  
    }
    ~SalaryEmployee() {
        Trace("SalaryEmployee::~SalaryEmployee()");
    }
    void SetSalary(int salary) {m_salary = salary;}
    int GetSalary() {return m_salary;}
    int GetPay() { return m_salary; }

private:
    int m_salary;
};
class WageEmployee : public Employee
{
public:
    WageEmployee(const char* name = "", int hours = 0, int wage = 0) : Employee(name), m_hours(hours), m_wage(wage)
    {
        Trace("WageEmployee::WageEmployee(const char*,int,int)");
    }
    ~WageEmployee()
    {
        Trace("WageEmployee::~WageEmployee()");
    }
    void SetHours(int hours) {m_hours = hours;}
    int GetHours() {return m_hours;}
    void SetWage(int wage) {m_wage = wage;}
    int GetWage() {return m_wage;}
    int GetPay() {return m_hours * m_wage; }
private:
    int m_hours;
    int m_wage;
};

#include "Trace.h"
#include "Employee.h"

int main()
{
    SalaryEmployee sally("Sally", 500);
    WageEmployee wally("Wally", 40, 10);

    Trace("Sally's name = ", sally.GetName());
    Trace("Sally's salary = ", sally.GetSalary());
    Trace("Wally's name = ", wally.GetName());
    Trace("Wally's hours = ", wally.GetHours());
    Trace("Wally's wage = ", wally.GetWage());

    Trace("Sally's pay = ", sally.GetPay());
    Trace("Wally's pay = ", wally.GetPay());
    return 0;
}