This Student Guide consists of two modules:
   Object-Oriented C++ Programming Fundamentals
   Intermediate C++ Programming

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 (Overview)

**Module 1. Object-Oriented C++ Programming Fundamentals**

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

**Module 2. Intermediate C++ Programming**

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1</td>
<td>ANSI C++ Library</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>Templates</td>
</tr>
<tr>
<td>Chapter 3</td>
<td>Input/Output in C++</td>
</tr>
<tr>
<td>Chapter 4</td>
<td>Practical Aspects of C++ Programming</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>Exception Handling</td>
</tr>
<tr>
<td>Chapter 6</td>
<td>Runtime Type Information</td>
</tr>
</tbody>
</table>

Appendix A  Reading List
Directory Structure

- The course software installs to the root directory \OCI\FndCpp for Module 1 and \OIC\ImdCpp for Module 2.
  - Example programs for each chapter are in named subdirectories of chapter directories Chap01, Chap02, and so on.
  - The 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 Demos directory is provided for doing in-class demonstrations led by the instructor.
  - The CaseStudy directory in Module 1 contains case studies in multiple steps.
# Table of Contents (Detailed)

## Module 1. Object-Oriented C++ Programming Fundamentals

### Chapter 1  Concepts of Object-Oriented Programming

- Object .............................................................................................................................. 3
- State and Behavior .......................................................................................................... 4
- Abstraction ...................................................................................................................... 5
- Encapsulation .................................................................................................................. 6
- Class and Instantiation .................................................................................................... 7
- Abstract Data Types ........................................................................................................ 8
- Abstract Data Type Example .......................................................................................... 9
- Methods ........................................................................................................................ 10
- Invoking Methods ......................................................................................................... 11
- Messages ....................................................................................................................... 12
- Class Inheritance ........................................................................................................... 13
- Polymorphism ............................................................................................................... 14
- Summary ....................................................................................................................... 15

### Chapter 2  Classes in C++

- C Data Encapsulation Example .................................................................................... 19
- Data Encapsulation in C ............................................................................................... 20
- The C++ Class .............................................................................................................. 21
- Example of C++ Class Specification ............................................................................ 22
- Structures and Classes in C++ ...................................................................................... 23
- Implementation of a C++ Class .................................................................................... 24
- Demo: Use of a C++ Class .......................................................................................... 25
- Fixed Example .............................................................................................................. 27
- **this** Pointer ............................................................................................................. 28
- Code Organization ........................................................................................................ 29
- Scope Resolution Operator ........................................................................................... 30
- Abstract Data Types ...................................................................................................... 31
- Test Programs for C++ Classes .................................................................................... 32
- Lab 2A .......................................................................................................................... 33
- Lab 2B ........................................................................................................................... 34
- Summary ....................................................................................................................... 35

### Chapter 3  Functions in C++

- Function Prototypes in C++ .......................................................................................... 41
- Strong Type Checking .................................................................................................. 42
- Conversion of Parameters ............................................................................................. 43
- Inline Functions ............................................................................................................ 44
- Inline Functions in Header Files ................................................................................... 45
- Default Arguments ....................................................................................................... 46
- Function Overloading ................................................................................................... 47
- Argument Matching ..................................................................................................... 48
- Argument Matching through Promotion ....................................................................... 49
- Match through Type Conversion .................................................................................. 50
Chapter 4  Constructors and Destructors ................................................................. 57
The Problem of Initialization ........................................................................ 59
Constructors and Initialization ................................................................ 60
Constructor in Stack Class ......................................................................... 61
Object Creation and Destruction .............................................................. 62
Destructors ................................................................................................. 63
Multiple Constructors ............................................................................... 64
String Class Implementation ................................................................... 65
String Class Demo ...................................................................................... 66
Hidden Constructors .................................................................................. 67
Using a Default Argument ........................................................................ 68
Lab 4A ........................................................................................................... 69
Lab 4B ............................................................................................................. 70
Summary ....................................................................................................... 71

Chapter 5: Memory Management in C++ ........................................................................ 77
Why Is Memory Management Important in C++? .................................. 79
Choices for an Object's Memory ............................................................... 80
Typical Memory Layout ........................................................................... 81
Free Store Allocation ................................................................................ 82
new Operator ............................................................................................. 83
Memory Allocation Errors ....................................................................... 84
new vs. malloc .......................................................................................... 85
delete Operator ......................................................................................... 86
Destructor (Review) .................................................................................. 87
Hiding Memory Management ................................................................. 88
String Class Specification (Version 2) ....................................................... 89
String Class Implementation .................................................................. 90
String Class Demo ..................................................................................... 91
String Class Bug ......................................................................................... 92
String Class Demo Workaround ............................................................... 93
String Class Exerciser Program ............................................................... 94
Exerciser Program Sample Run .............................................................. 96
Lab 5 ............................................................................................................. 97
Summary ....................................................................................................... 98

Chapter 6  References and Argument Passing in C++ .............................................. 103
Variables .................................................................................................. 105
Argument Passing .................................................................................... 106
Call-by-Value ............................................................................................ 107
Reference Declarations .......................................................................... 108
Call-by-Reference .................................................................................... 109
Copy Constructor ..................................................................................... 110
Default Copy Constructor ....................................................................... 111
Bug in String Class .................................................................................. 112
<table>
<thead>
<tr>
<th>Chapter 9</th>
<th>Introduction to Inheritance</th>
<th>189</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance Concept</td>
<td>191</td>
<td></td>
</tr>
<tr>
<td>Inheritance Example</td>
<td>192</td>
<td></td>
</tr>
<tr>
<td>Inheritance in C++</td>
<td>193</td>
<td></td>
</tr>
<tr>
<td>Employee Example</td>
<td>194</td>
<td></td>
</tr>
<tr>
<td>Employee Test Program</td>
<td>195</td>
<td></td>
</tr>
<tr>
<td>Protected Members</td>
<td>196</td>
<td></td>
</tr>
<tr>
<td>Base Class Initializer List</td>
<td>197</td>
<td></td>
</tr>
<tr>
<td>Composition</td>
<td>198</td>
<td></td>
</tr>
<tr>
<td>Member Initializer List</td>
<td>199</td>
<td></td>
</tr>
<tr>
<td>Order of Initialization</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>Inheritance vs. Composition</td>
<td>201</td>
<td></td>
</tr>
<tr>
<td>Lab 9A</td>
<td>202</td>
<td></td>
</tr>
<tr>
<td>Lab 9B</td>
<td>203</td>
<td></td>
</tr>
<tr>
<td>Summary</td>
<td>204</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 10</th>
<th>Polymorphism and Virtual Functions</th>
<th>211</th>
</tr>
</thead>
<tbody>
<tr>
<td>A Case for Polymorphism</td>
<td>213</td>
<td></td>
</tr>
<tr>
<td>Dynamic Binding</td>
<td>214</td>
<td></td>
</tr>
<tr>
<td>Pointer Conversion in Inheritance</td>
<td>215</td>
<td></td>
</tr>
<tr>
<td>Polymorphism Using Dynamic Binding</td>
<td>216</td>
<td></td>
</tr>
<tr>
<td>Virtual Function Specification</td>
<td>217</td>
<td></td>
</tr>
<tr>
<td>Invoking Virtual Functions</td>
<td>218</td>
<td></td>
</tr>
<tr>
<td>Virtual Functions Demo</td>
<td>219</td>
<td></td>
</tr>
<tr>
<td>Vtable</td>
<td>223</td>
<td></td>
</tr>
<tr>
<td>Virtual Destructors</td>
<td>224</td>
<td></td>
</tr>
<tr>
<td>Abstract Class Using Pure Virtual Function</td>
<td>225</td>
<td></td>
</tr>
<tr>
<td>Employee as an Abstract Class</td>
<td>226</td>
<td></td>
</tr>
<tr>
<td>Heterogeneous Collections</td>
<td>227</td>
<td></td>
</tr>
<tr>
<td>Polymorphic Code Example</td>
<td>228</td>
<td></td>
</tr>
<tr>
<td>Lab 10</td>
<td>229</td>
<td></td>
</tr>
<tr>
<td>Summary</td>
<td>230</td>
<td></td>
</tr>
</tbody>
</table>
Module 1:

Object-Oriented C++ Programming Fundamentals

Top Level Directory:  OIC\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

  – 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 of Encapsulation]

  - 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:

  ![Diagram](image)

  - 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
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:

  - 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

Salary

SalaryEmployee

Name

Wage

Hours

WageEmployee

Name
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);}
     void SetName(const char *name)
     {strcpy(m_name, name);}
     const char* GetName() const {return m_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
// DemoEmp.cpp
// Demo program for Employee classes

#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;
}
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  \texttt{String::String(const char*)}  does the initialization.

b) The overloaded operator  \texttt{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  \texttt{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 (\texttt{SalaryEmployee}) is constructed first and destroyed last. The output from running the program shows the invocation of constructors and destructors:

\begin{verbatim}
String::String(const char *str)
Employee::Employee(const char*)
SalaryEmployee::SalaryEmployee(const char *, int)
String::String(const char *str)
Employee::Employee(const char*)
WageEmployee::WageEmployee(const char*,int,int)
operator const char* () const
Sally's name = Sally
Sally's salary = 500
operator const char* () const
Wally's name = Wally
Wally's hours = 40
Wally's wage = 10
WageEmployee::~WageEmployee()
Employee::~Employee()
String::~String()
\end{verbatim}

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 \texttt{SalaryEmployee} and \texttt{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)" );
    }
    ~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 name = ", wally.GetName());
    Trace("Sally's salary = ", sally.GetSalary());
    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;
}
Module 2:

Intermediate C++ Programming

Top Level Directory: OIC\ImdCpp
ImdCpp
Chapter 1

ANSI C++ Library
ANSI C++ Library

Objectives

After completing this unit you will be able to:

- Outline the functionality of the ANSI C++ library.
- Explain the new header files.
- Explain the use of namespaces in ANSI C++.
- Describe the string class in ANSI C++
- Describe the use of templates in ANSI C++.
- Implement simple programs using features of the ANSI C++ library.
ANSI C++ Library

- ANSI C++ comes with a greatly enhanced library with new header files and organization.
  - Header files do not take a .h extension.
  - Symbols in the library are in the namespace “std”.
  - Templates are used extensively.

- There are a number of features in the library
  - The standard C library is still there.
  - Iostreams now has templates, throws exceptions, and supports strings.
  - There is a standard string class.
  - The “Standard Template Library” has been incorporated, including many container classes and algorithms.
  - There is support for numeric processing, including support for complex numbers, higher precision, and compiler optimizations.
  - Diagnostic support includes a number of exception classes.
Hello ANSI C++

• Best way to get a feel for the library is to try a simple program.
  – The example programs are in the chapter directory.

  // tryhello.cpp

  #include <iostream>

  int main()
  {
    cout << "Hello, ANSI C++\n";
    return 0;
  }

• Type in and try to build the above program. What happens?
  – You get errors on the symbols `cout` and `<<`.
  – In ANSI C++ the symbols are in a `namespace`.
Namespaces

- C++ provides a single global namespace in which all names declared in global scope are entered.
  - Single namespace is difficult for library providers and users.
  - Global names in a library may collide with the global names in a user application or another library (e.g. there may be two Vector classes).

- ANSI C++ provides a “namespace” mechanism to avoid such conflicts.
  - The ANSI C++ standard library is in the namespace “std”.
  - To gain access to the symbols in it you may employ a “using namespace” statement in your program.

```cpp
// ansihello.cpp
#include <iostream>
using namespace std;

int main()
{
    cout << "Hello, ANSI C++\n";
    return 0;
}
```
ANSI C++ String Class

- The ANSI C++ has a class *string* with quite intuitive semantics.

```cpp
// trystring.cpp

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

int main()
{
    string a = "bat";
    string b = "man";
    string c = a + b;
    cout << c << endl;
    cout << "length = " << c.length() << endl;
    return 0;
}
```
Templates

• A template can be used to generate a function or class based on type parameters.
  
  – You instantiate a template by passing an actual type parameter in angle brackets.
  
  – For example, the Standard Library type string is actually a typedef for a template instantiation of the fundamental type basic_string.

  
  typedef basic_string<char> string;

• Templates are discussed in the next chapter.
basic_string Example

- Example program illustrates a basic_string of int.

```cpp
// intstring.cpp

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

int main()
{
    int alpha[] = {2, 3, 5};
    int beta[] = {7, 11, 13};
    basic_string<int> a(alpha, 3);
    basic_string<int> b(beta, 3);
    basic_string<int> c;
    c = a + b;
    cout << "length = " << c.length() << endl;
    for (int i = 0; i < c.length(); i++)
        cout << c[i] << endl;
    return 0;
}
```
Lab 1

Developing a Familiarity with the ANSI Libraries

These exercises show that the ANSI libraries include the functionality of the old C++ libraries. While the use of templates may be confusing, we want you to be ready to use the new libraries when we will cover templates in the next chapter (where you will also learn about new features such as STL).

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

Suggested time: 45 minutes
Summary

- The ANSI C++ Standard Library has extensive functionality, including support for numeric processing and strings, and it incorporates the Standard Template Library.

- The Standard Template Library (STL) has extensive support for containers and algorithms.

- The new header files do not take a .h extension, and all symbols are in the namespace `std`.

- The `string` class in ANSI C++ is a template instantiation of `basic_string`, which can be used to implement sequences of other types besides `char`.
Lab 1

Developing a Familiarity with the ANSI Libraries

These exercises show that the ANSI libraries include the functionality of the old C++ libraries. While the use of templates may be confusing, we want you to be ready to use the new libraries when we will cover templates in the next chapter (where you will also learn about new features such as STL).

Suggested Time: 45 minutes

Root Directory: OIC\ImdCpp

Directories: Labs\Lab1\StringStack (working directory)
Chap01\StringStack\Step0 (source of starter files)
Chap01\StringStack\Step1 (answer for Exercise 1)
Chap01\StringStack\Step2 (answer for Exercise 2)

Exercise 1

You will use the standard C++ library string class to simplify the exerciser program for the stack class. Starting point is Chap01\StringStack\Step0. This is a lab from the fundamentals module that implements a stack of integers. The challenge is to convert over to using the new ANSI C++ style header files.

1. Create a new empty directory StringStack. (If you are using an IDE, the directory may be created as part of creating a project.) Copy the files testdynm.cpp, intstack.h, intstack.cpp, trace.h from Chap02\StringStack\Step0 to your new StringStack directory. Build the program and play with the stack program to see how it works. You can create and destroy a stack, push and pop items off of it, list the items on the stack, and quit. Entering a “?” at the command prompt gives a list of commands. (NOTE: If you experience difficulty getting your compiler to build using the “old” style C++ headers, just focus on getting it working with the new headers!)

2. In trace.h replace <iostream.h> by <iostream>.

3. In intstack.cpp insert an include of <iostream> and a using namespace std; before the include of trace.h

4. In testdynm.cpp replace <iostream.h> by <iostream> and <string.h> by <string>, followed by using namespace std;:

5. Replace the definition of cmd by defining it to be of type string. You can now replace the strcmp C library function by overloaded operators such as ==.

6. After you have completely converted over to using string operators, build and test. You should have exactly the same behavior as the Step0 IntStack project. Code at this point is StringStack\Step1. Exercise the various stack commands to convince yourself.
Exercise 2

In this exercise you will implement a StringStack class based on the IntStack. All your code should be based on the ANSI C++ standard library. Continue working in the StringStack directory.

1. Save intstack.h as stringstack.h and save intstack.cpp as stringstack.cpp. Remove intstack.h and intstack.cpp from your project, and add stringstack.h and stringstack.cpp to your project.

2. In stringstack.h replace all references to IntStack by StringStack. Where int is used as the data type for stack elements, replace by string. Add #include <string>.

3. In stringstack.cpp replace all references to IntStack by StringStack. Where int is used as the data type for stack elements, replace by string. Add #include <string>.

4. In trace.h add #include <string> and a new overloaded trace function taking a string argument.

5. In testdynm.cpp replace all references to IntStack by StringStack. Replace references to the stack element int with the string type. You can now build and test. Code is now StringStack\Step2.
Chapter 5

Exception Handling
Exception Handling

Objectives

After completing this unit you will be able to:

• Define the C++ exception mechanism and contrast it with handling errors by function return codes as in C.

• Describe “throw”, “try” and “catch” as they are used to implement exception handling in C++.

• Implement exception handling in your programs.

• Explain the concepts of context and stack unwinding.

• Describe what happens to an uncaught exception.

• Explain the automatic cleanup process that occurs with C++ exception handling.

• Describe how matching of a thrown exception is done in the case of multiple catch handlers.
Exception Handling

• The normal way of making a call that may result in an error is to have a "status" return value.

```cpp
status = some_call(...);
if (status != OK)
    // error handling
```

• Not always feasible in C++:
  
  – Overloaded operators

  – Constructors

• C++ exception mechanism supports "catching" exceptions without having to check a return code.

  – Exception handling is an important feature of C++, part of the ANSI/ISO standard.

• Exceptions are "thrown."

  – In code that detects an exception use a `throw` statement.

  – Enclose code that might cause an exception in a `try` block.

  – Put exception handling code in a `catch` block.
Exception Handling Example

- Array class is modified to throw an exception on an “out of bounds” error.
  - See array.h in Array\Step2.

```cpp
... template<class T> void Array<T>::SetAt(int i, const T& x) {
    // assert((i >= 0) && (i < m_size));
    if ( (i < 0) || (i >= m_size) )
        throw ("Out of bounds");
    m_array[i] = x;
}

template<class T> T Array<T>::GetAt(int i) {
    // assert((i >= 0) && (i < m_size));
    if ( (i < 0) || (i >= m_size) )
        throw ("Out of bounds");
    return m_array[i];
}
```
try and catch

while (strcmp(cmd, "quit") != 0)
{
    try
    {
        if (strcmp(cmd, "set") == 0)
        {
            cout << "index: ";
            cin >> index;
            cout << "set item: ";
            cin >> item;
            array.SetAt(index, item);
        }
        else if (strcmp(cmd, "get") == 0)
        {
            cout << "index: ";
            cin >> index;
            item = array.GetAt(index);
            cout << "get item = " << item << '\n';
        }
        else if (strcmp(cmd, "print") == 0)
            array.Print();
        else
        {
            // display list of legal commands
            ...
        }
    }
    catch (char * errmsg)
    {
        cerr << errmsg << '\n';
    }
    cout << ": ";
    cin >> cmd;
}
Exception Flow of Control

• Code which might cause an exception to be thrown should be enclosed in a “guarded” section of code known as a *try block*.

• Directly below the try block is one or more *catch handler*.
  
  − Each catch handler has a parameter specifying the data type of exception that it can handle.
  
  − The exception data type can be any built-in type or a class type.
  
  − If an exception is thrown, the *first* catch handler that matches the exception data type is executed, and then control passes to the statement just after the catch block(s).
  
  − If no handler is found, the exception is thrown to the next higher “context” (e.g. the function that called the current one).
  
  − If no exception is thrown inside the try block, all the catch handlers are skipped.
**Context and Stack Unwinding**

- As the flow of control of a program passes into nested blocks, local variables are pushed onto the stack and a new “context” is entered.
  - Likewise a new context is entered on a function call, which also pushes a return address onto the stack.

- If an exception is not handled in the current context, the exception is passed to successively higher contexts until it is finally handled (or else is “uncaught” and is handled by a default `terminate` function).

- When the higher context is entered, C++ adjusts the stack properly, a process known as stack unwinding.
  - In C++ exception handling, stack unwinding involves both setting the program counter and cleaning up variables.
Handling Exceptions in Best Context

- One of the benefits of the C++ exception handling mechanism is that it is easier to handle an exception in the appropriate context.

  - The exception automatically propagates to higher contexts until an appropriate handler is found.

  - In C you must use status return codes, and be carefully to keep passing the right return code at each level of function call.

- As an example consider the array program, with the handling of commands delegated to an `execute` function inside the while loop.

  - The best place to handle the exception is inside the `while` loop, where an error message can be printed and a new command read in.

  - But exceptions may be thrown inside the `execute` function, at a lower context.
Context Example (tstarray.cpp)

- The exception handling is done at the top level
  - See tstarray.cpp in Array\Step3.

```cpp
void execute(const char* cmd);

Array<int> array(5);

int main()
{
    char cmd[80]; // command
    cout << ": "; // prompt
    cin >> cmd;
    while (strcmp(cmd, "quit") != 0)
    {
        try
        {
            execute(cmd);
        }
        catch (char * errmsg)
        {
            cerr << errmsg << '
';
        }
        cout << ": ";
        cin >> cmd;
    }
    return 0;
}
```
Context Example (Cont’d)

• In the lower level `execute` function, where you actually do the various array operations, you do not have to have any error processing code!
  – Since there are no `try ... catch` blocks, and thrown exceptions will automatically propagate to higher context.

```c++
void execute(const char* cmd)
{
    int item;
    int index;

    if (strcmp(cmd, "set") == 0)
    {
        cout << "index: ";
        cin >> index;
        cout << "set item: ";
        cin >> item;
        array.SetAt(index, item);
    }
    else if (strcmp(cmd, "get") == 0)
    {
        cout << "index: ";
        cin >> index;
        item = array.GetAt(index);
        cout << "get item = " << item << '\n';
    }
    else if (strcmp(cmd, "print") == 0)
    { array.Print();
    }
    else
    {
        cout << "legal commands are:\n";
        ...
    }
}
```
Benefits of Exception Handling

• As we have just seen, exceptions can be handled in the context most convenient to the program logic

• In many cases a number of operations that might cause an exception can be taken inside a single guarded section of code, without having to check each individual operation
  – Our array example did not need individual checks on the calls SetAt and GetAt

• In contrast to status returns, exceptions cannot be ignored
  – How many programmers both to check the return code of printf?
  – We could have designed the execute function in “array” example to have a return code, but then we would both have had the burden of programming the return, and even if we did, the user of the function could have ignored it!

• The stack unwinding process automatically cleans up variables, including calls to appropriate destructors
Unhandled Exceptions

- If no handler at any levels catches an exception, the exception is said to be “unhandled” or “uncaught”.
  - An uncaught exception can also occur if a new exception is thrown before an existing exception reaches its handler.

- When an uncaught exception occurs, the special function `terminate` is called.
  - The default behavior of `terminate` is to print an error message and call `abort`.
  - Although this “uncaught” behavior is not graceful, it is better than having no error message printed and unpredictable results occur.

- You can customize the treatment of an uncaught exception by calling the `set_terminate` function.
Clean Up

• As part of unwinding the stack, C++ takes care of popping local variables, which causes the destructors to be called for class objects.
  - But objects on the heap are *not* automatically deleted.

• For example, consider *Array\Step4*.
  - At beginning of `execute` function two `String` objects are created, one on heap and one on stack.
  - Trace statements in `String` class have been augmented to show the contents of the string on construction and destruction.
  - Run the program, with and without an exception. In both cases stack object gets destroyed, but heap object is *not* destroyed in case of exception – program needs to be made smarter.

```cpp
void execute(const char* cmd)
{
    String str1 ("I am on stack");
    String* pstr2 = new String("I am on heap");

    ... // exception may be thrown here

    delete pstr2;
}
```
Multiple Catch Handlers

• An exception object in C++ is typed.
  – If you have several catch handlers, the first one that matches the thrown object will be invoked.
  – Standard automatic type conversions by constructors and cast operators are not performed when an exception is thrown.

• As an example consider Demos\Multicatch.
  – See program on next page.
  – Run first with throwing an integer. As expected the int catch handler gets called.
  – Now rebuild and run with throwing a String. Although the String class has a conversion to const char *, the only match is with the String catch handler.
// multiple.cpp

#include <iostream.h>
#include "trace.h"
#include "strn.h"

int main()
{
    try
    {
        String s("I am a string");
        int n = 17;
        // throw s;
        throw n;
    }
    catch (int n)
    {
        Trace("int caught: ", n);
    }
    catch(const char *str)
    {
        Trace("char * caught: ", str);
    }
    catch(String s)
    {
        Trace("String caught: ", (const char*) s);
    }
    return 0;
}
Lab 5

Using Exceptions

In this exercise you will modify the template version of a stack class and its test program to use exception handling. The starter version prints an error message and aborts when error occurs. The exception handler version will print an error message and then prompt for a new command.

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

Suggested time: 20 minutes
Summary

- The C++ exception mechanism provides a robust means of dealing with exceptional program behavior without need of tracking function return codes.

- You “throw” an exception. An operation which may throw an exception should be performed in a “try” block, and the exception is handled in a “catch” block.

- Exceptions not handled at the current context are propagated to higher contexts.

- An “uncaught exception” is ultimately handled by the `terminate` function, which will abort the program. Thus exceptions cannot be ignored.

- The stack unwinding process automatically cleans up local variables, but not objects on the heap.

- If there are several catch handlers, the first one that matches the thrown exception will be called.
Lab 5

Using Exceptions

In this exercise you will modify the template version of a stack class and its test program to use exception handling. The starter version prints an error message and aborts when error occurs. The exception handler version will print an error message and then prompt for a new command.

Suggested time: 20 minutes

Root Directory: OIC\AdvCpp

Directories: Labs\Lab3\StackException (do your work here)
             Chap05\Stack\Step3 (backup copy of starter files)
             Chap05\Stack\Step4 (answer)

Files to Modify:
    stack.h
    tststack.cpp

Instructions:

1. In the working directory is the template version of a stack class, with a test program that exercises an integer stack. If an error occurs in the stack functions, the program prints an error message and aborts. Verify this functionality.

2. Modify both the stack class and the test program to make use of exception handling. The stack class should throw an exception (of type char*) on a stack error. The test program should use a try block. On detecting an error the program should print an error message (which is passed as an argument of the exception) and then prompt for a new command.