Advanced C++ Programming

Robert J. Oberg
Michael Saltzman

Student Guide

Revision 1.2

Object Innovations Course 157

Advanced C++ Programming
Rev. 1.2

This Student Guide consists of three modules:

Intermediate C++ Programming
Advanced C++ Topics
Fundamentals of STL

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

Printed in the United States of America.

Table of Contents (Overview)

Directory Structure

- **The course software installs to the root directory *OI\ImdCpp* for Module 1 and *OI\AdvCpp* 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.
- **Module 3 has the top-level directory *OI\FndStl*.**
  - Example programs for each chapter are in named subdirectories of chapter directories **Chap01**, **Chap03**, and so on.
  - Lab work is done in **Labs** directory with numbered subdirectories for each lab. Starter code is provided.
  - Solutions are in **Solutions** directory with corresponding numbered subdirectories.

Table of Contents (Detailed)

Module 1. Intermediate C++ Programming

Module 2. Advanced C++ Topics

Module 3. Fundamentals of STL

Module 1:

Intermediate C++ Programming

Top Level Directory: OI\ImdCpp

Chapter 1

Inheritance and Polymorphism

Part I. Introduction to Inheritance

Objectives

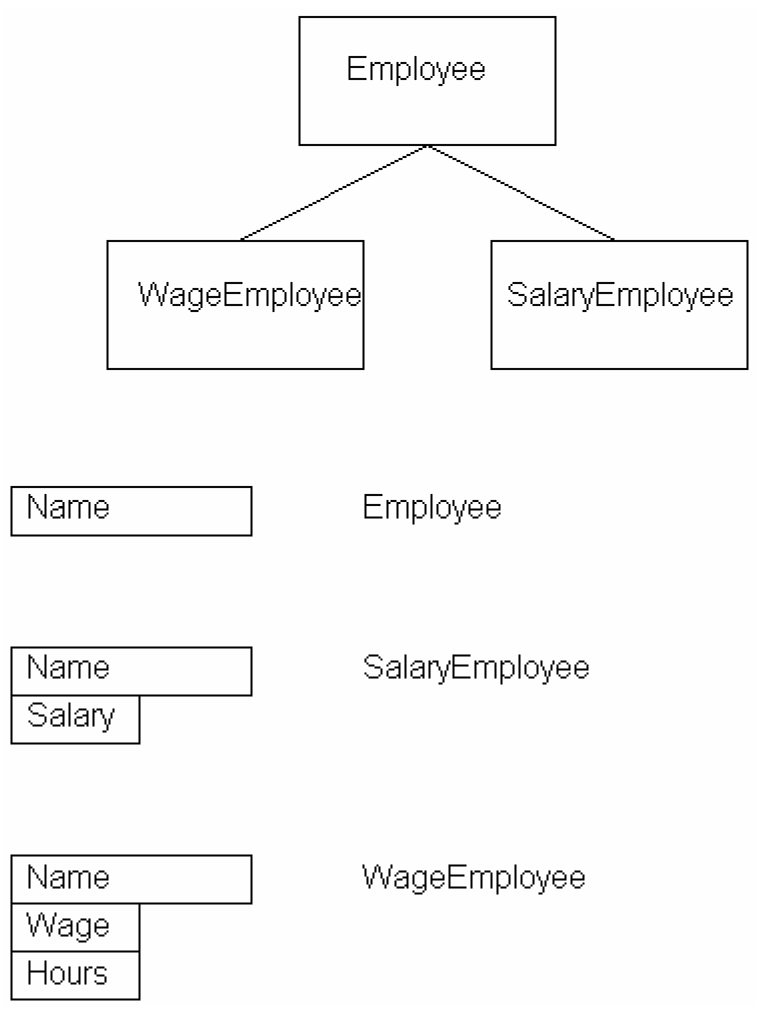
After completing this unit you will be able to:

- **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



Inheritance in C++

- **Inheritance is implemented in C++ by a mechanism known as class derivation.**

```
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 = "");
    void SetName(const char *name)
        {strcpy(m_name, name;}
    const char* GetName() const {return m_name;}
private:
    char m_name[MAXNAME];
};
class SalaryEmployee : public Employee
{
public:
    SalaryEmployee(const char *name = "",
                int salary = 0);
    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 = 0, int wage = 0);
    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;
};
```

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.**
  - If **m_name** were 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."**

```
class SalaryEmployee : public Employee
{
public:
    SalaryEmployee(const char *name = "",
                   int salary = 0);
    ...
};
```

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

```
SalaryEmployee::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.**

```
class Employee
{
public:
    Employee(const char *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.**

```
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.**

```
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.**

```
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**
  - 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 when you want the features of another class but not its interface.

Lab 1A

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 1A write-up at the end of the chapter.

Suggested time: 30 minutes

Summary – Inheritance

- **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.**

Part II. Polymorphism and Virtual Functions

Objectives

After completing this unit you will be able to:

- **Explain the features of virtual functions and dynamic binding.**
- **Describe pointer conversion in C++ inheritance and use pointers in connection with virtual functions.**
- **Use polymorphism in C++ to write better structured, more maintainable code.**
- **Provide virtual destructors for classes using virtual functions.**
- **Specify abstract classes using pure virtual functions.**

A Case For Polymorphism

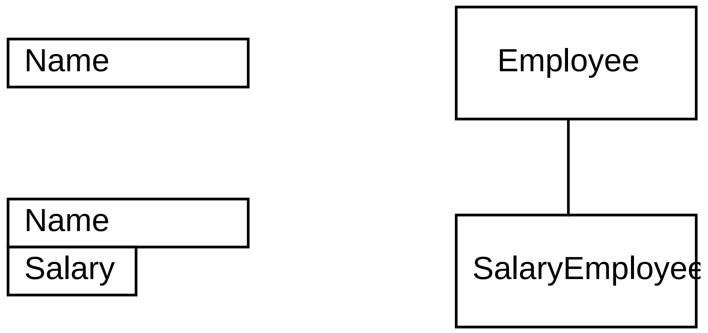
- **Consider the problem of generating a payroll for different types of employees.**
  - Wage and salary employees have pay calculated by different algorithms.
  - A traditional approach is to maintain a type field in an employee structure and to calculate pay in a switch statement, with cases for each type.
  - Such switch statement type code is error prone, and requires much maintenance when adding a new type (e.g. sales employee, where pay is based on commission).
- **An alternative is to localize the intelligence to calculate pay in each employee class, which will support its own *GetPay* function.**
  - Generic payroll code can then be written that will handle different types of employees, and will not have to be modified to support an additional employee type.
  - Provide a **GetPay** function in the base class, and an override of this function in each derived class
  - Call **GetPay** through a pointer to a general **Employee** object. Depending on the actual Employee class pointed to, the appropriate **GetPay** function will be called

Dynamic Binding

- **This use of "switch" statement must be replicated wherever Employee objects are manipulated.**
  - It is error prone.
  - Much duplicate code must be maintained.
- **Virtual functions and dynamic binding remove this problem.**
- **The member function *GetPay* in each class uses the appropriate algorithm for calculating pay.**
- **Declare *GetPay* as a virtual function.**
- **Then the compiler resolves the class type at runtime using the internal mechanism of dynamic binding.**
- **No need for type field in Employee class.**
- **No need for checking type of an Employee object.**

Pointer Conversion in Inheritance

- **Pointers can be converted up in an inheritance hierarchy but not down.**



- **Consider pointers to *Employee* and *SalaryEmployee*.**

```
Employee* pEmp = new Employee("John");
SalaryEmployee* pSalEmp = new SalaryEmployee(
                              "Bill", 1500);

pEmp = pSalEmp;  // legal
```

– A salary employee *is a* employee. It is safe to access the fields of **Employee** through **`pSalEmp`**, because the object pointed to by **`pSalEmp`** contains all the fields of **Employee** and possibly some additional fields.

```
pSalEmp = pEmp;  // illegal
```

– An employee is not necessarily a salary employee. If not, there will be an error in accessing the additional field **salary** through **`pEmp`**.

Polymorphism Using Dynamic Binding

- **A generic pointer to a base class can be changed at run time to point to an object belonging to a derived class.**
- **A virtual function in the base class is overridden in the derived class.**
- **The virtual function is called through a generic pointer to the base class. Which override of the function that gets invoked is determined at runtime by the class of object referenced by the pointer**
  - This runtime determination of which function is called is referred to as ***dynamic binding*.**
  - The ability for the same function call to result in different behavior depending on the object through which the function is invoked is referred to as ***polymorphism*.**
- **Polymorphic functions are declared as virtual by the programmer.**
- **Dynamic binding mechanism is carried out by the compiler. User need not know any details.**

Virtual Function Specification

- **A member function is declared as virtual within the class declaration as:**

```
class Employee
{
public:
    virtual int GetPay();
    ...
};
```

- **The function is overridden in derived classes.**

```
class SalaryEmployee : public Employee
{
public:
    int GetPay()  // keyword virtual not necessary
    { return m_salary; }
    ...
};

class WageEmployee : public Employee
{
public:
    int GetPay()
    { return m_hours * m_wage; }
    ...
};
```

- **Each redefinition of `GetPay` must exactly match the signature of the function in the base class.**

Invoking Virtual Functions

- **To use dynamic binding, a virtual function must be invoked through a pointer (or a reference).**

```
int pay;
Employee* pEmp;
WageEmployee Joe;
SalaryEmployee Mary;

pEmp = &Joe;
pay = pEmp->GetPay();     // wage version

pEmp = &Mary;
pay = pEmp->GetPay();     // salary version
```

- **The use of class scope operator disables dynamic binding**

```
pEmp->WageEmployee::GetPay(); // always resolves to
                              // WageEmployee::GetPay()
```

Virtual Functions Demo

- **C++ virtual functions will be demonstrated by the following simple program that you can build and run and then modify.**
  - Build and run it. Use **Demos\Virtdemo** directory for your work

```
// virtdemo.cpp

#include <iostream>

using namespace std;

class B
{
public:
    void f();
    void g();
private:
    long x;
};

class D : public B
{
public:
    void f();
    void g();
private:
    long y;
};

void B::f() {cout << "B::f" << endl;}
void B::g() {cout << "B::g" << endl;}
void D::f() {cout << "D::f" << endl;}
void D::g() {cout << "D::g" << endl;}
```

Virtual Functions Demo (Cont'd)

```
int main()
{
    B   b, *pb;
    D   d, *pd;
    pb = &b;
    pd = &d;
    b.f();
    d.f();
    pb->f();
    pd->f();
    pb = pd; // legal??
    pb->f();
    pd = pb; // legal??
    cout << "size B = " << sizeof(B) << endl;
    cout << "size D = " << sizeof(D) << endl;
    return 0;
}
```

- **Before building it, predict any compiler errors. Comment out any offending lines and build again**

- **Before running it, predict the output**

- **How can you change the definition of the base class to get the "expected" output from**

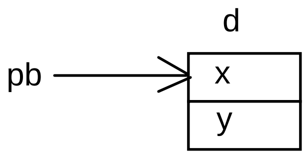
```
        pb->f()
```

after the pointer has been reassigned to point to a D object?

Virtual Functions Demo (Cont'd)

- **There is static binding, and *pb->f()* will always call the "B" version of the function. Output:**

```
B::f
D::f
B::f
D::f
B::f
size B = 4
size D = 8
```



- **Now declare the functions virtual in the base class and run again**

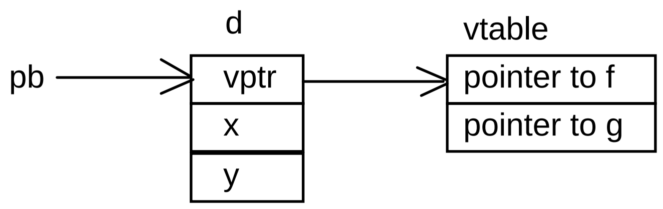
```
class B
{
public:
    virtual void f();
    virtual void g();
private:
    long x;
};
```

Virtual Functions Demo (Cont'd)

- **There is dyanamic binding, and *pb->f()* will call the "B" version of the function if the pointer has been assigned to point to a *D* object. Output:**

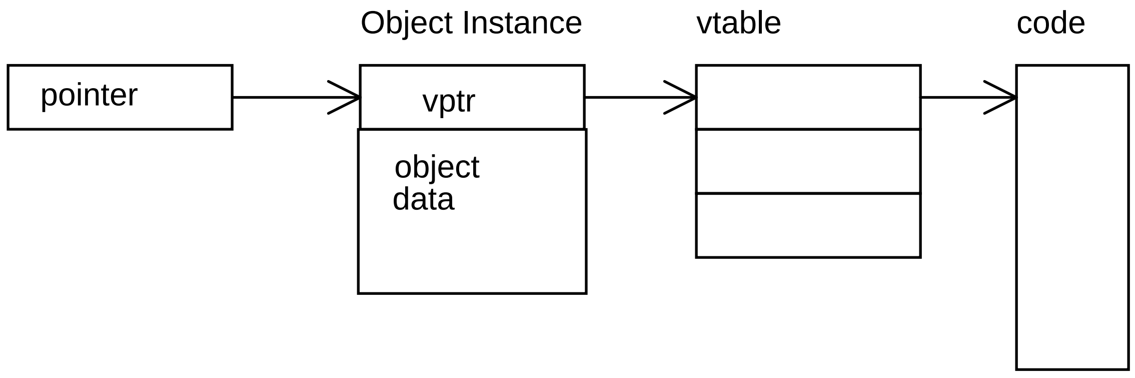
```
B::f
D::f
B::f
D::f
D::f
size B = 8
size D = 12
```

- **The size of the objects is increased by 4 bytes, because each object instance now holds a "vptr" (pointer to a vtable).**



VTable

- **Virtual functions are accessed through a pointer (or reference, which is implemented by a pointer).**
- **The pointer points to an area of memory of the object instance which contains a pointer to the object's *vtable***
  - The vtable contains an array of function pointers, which point to code implementing the member functions of the interface
  - The vtable is associated with the "class" corresponding to the object -- there is a single vtable for all object instances



Virtual Destructors

- **Suppose *Employee* classes have a destructor (e.g. private storage of name is changed to use heap).**

```
Employee* pEmp;
SalaryEmployee* pSalEmp = new SalaryEmployee
                          ("John", 1500);
pEmp = pSalEmp;
delete pEmp;
```

- **Which destructors are involved? Destructor for *Employee*, *SalaryEmployee*, or both?**
- **Answer is only *Employee*, even though it was intended to destroy *pSalEmp*.**
- **To destroy *pSalEmp*, sequence should be destructor for *SalaryEmployee* and then for *Employee*.**
- **Specify *Employee* destructor as *virtual*.**

```
class Employee
{
public:
    virtual ~Employee();
    ...
};
```

- **In general it is a good idea to declare destructors of base classes as virtual.**

Abstract Class Using Pure Virtual Function

- **Often it is desirable to have a base class as a protocol for deriving implementations in the derived classes, without the base class having to implement all the specified functions itself.**
  - A virtual function specified but not implemented in the base class is referred to as a ***pure virtual function*.**
  - Notation for a pure virtual function is **= 0** after its prototype.

```
virtual int GetPay() = 0;
```

- **For a pure virtual function, only its signature is specified while its definition is deferred to derived classes.**
- **A class that contains at least one pure virtual function is referred to as an *abstract class*.**
  - An abstract class cannot be instantiated.
  - For any derived class to be non-abstract, it must define all inherited pure virtual functions.

Employee as an Abstract Class

- **The *GetPay* function is not meaningful in the Employee class—more information about an employee is need to calculate pay.**
  - Declare **GetPay** as a pure virtual function in **Employee**.
  - This make **Employee** into an abstract class.
  - Generic code can be written for employees.

```
class Employee
{
public:
    virtual int GetPay() = 0;
    ...
};

class SalaryEmployee : public Employee
{
public:
    int GetPay() { return m_salary; }
    ...
};

class WageEmployee : public Employee
{
public:
    int GetPay()
    { return m_hours * m_wage; }
    ...
};
```

Heterogeneous Collections

- **A heterogeneous collection can be constructed using pointers to a base class**
  - A pointer to a base class is generic, and at run time can be assigned to point to different derived classes
- **Consider an array of Employee pointers**

```
Employee* pEmp[10];       // up to 10 employees of
                          // different types
int nNumEmp;          // number of employees

WageEmployee Joe("Joe",40,15);
SalaryEmployee Mary("Mary",1500);

pEmp[0] = &Joe;
pEmp[1] = &Mary;
nNumEmp = 2;
```

Polymorphic Code Example

- **We can now write generic, polymorphic code to calculate pay for a group of Employees.**
  - This code is general and won't change, even if new classes of employees are defined, provided each derived employee class implements **GetPay** function.

```
int payroll[10];

for (int i = 0; i < nNumEmp; ++i)
    payroll[i] = pEmp[i]->GetPay();
```

Lab 1B

Polymorphism in Employee Class Hierarchy

In these exercises you will study an enhanced Employee class hierarchy in which **Employee** is now an abstract base class and **GetPay** is a pure virtual function. There is a heterogeneous array of employee pointers and a generic function **PayReport** that polymorphically prepares a report, delegating to each employee's **GetPay** function to calculate pay appropriately. You will look at issues of deleting objects from the collection. Finally you will add a new employee class and observe how easy it is to maintain a program having this kind of structure.

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

Suggested time: 45 minutes

Summary – Polymorphism

- **Virtual functions and dynamic binding support the concept of polymorphism.**
- **Polymorphic code is cleaner and easier to maintain, eliminating forests of switch statements.**
- **Dynamic binding rests on accessing functions through pointers, determining the function invoked at runtime by the class of the object pointed to.**
- **In C++ inheritance hierarchies it is safe to cast a pointer to a class higher in the hierarchy but not to a class that is lower.**
- **Virtual destructors are essential for using delete on a class with virtual functions.**
- **Abstract classes are defined using pure virtual functions and cannot be instantiated.**
- **Heterogeneous collections can be constructed in C++ by using pointers to a base class.**

Lab 1A

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.

Suggested time: **30 minutes**

Root Directory: **OI\ImdCpp**

Directory:	**Labs\Lab1A\Employee**	(do your work here)
	Chap01\Employee\Step1	(backup copy of starter files)
	Chap01\Employee\Step2	(answer)

Files to Modify :

demoemp.cpp
employee.h

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 **demoemp.exe** 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 section!)

Lab 1B

Polymorphism in Employee Class Hierarchy

In these exercises you will study an enhanced Employee class hierarchy in which **Employee** is now an abstract base class and **GetPay** is a pure virtual function. There is a heterogeneous array of employee pointers and a generic function **PayReport** that polymorphically prepares a report, delegating to each employee's **GetPay** function to calculate pay appropriately. You will look at issues of deleting objects from the collection. Finally you will add a new employee class and observe how easy it is to maintain a program having this kind of structure.

Suggested time: **45 minutes**

Root Directory: OI\ImdCpp

Directory:	**Labs\Lab1B\EmployeePoly**	(do your work here)
	Chap01\Employee\Start3	(backup copy of starter files)
	Chap01\Employee\Step3	(answer)

Files to Modify :

demopoly.cpp
employee.h
employee.cpp

Instructions:

1. The file **employee.h** in the work directory specifies an abstract class of employees. The pure virtual function **GetPay** specifies a function to determine the pay of an employee. There are two concrete derived classes **WageEmployee** and **SalaryEmployee**. A wage employee has an hourly rate of pay and a number of hours worked. Pay is
```
rate * hours.
```
A salary employee has a salary which is equal to the pay. The code file **employee.cpp** contains an implementation of **GetPay** for the two concrete classes. The code file **demopoly.cpp** demonstrates polymorphism and a heterogeneous collection. An array of pointers to **Employee** is declared, and two employee objects are created on the heap. A function **PayReport** takes as an argument an array of pointers to **Employee** and prints a report showing name and pay of each employee, the proper pay being calculated based on the type of employee. Study this code, making sure you understand how this polymorphic behavior is implemented through virtual functions. Then predict what the output will be, including what constructors and destructors are called. Build and run the program to verify your understanding.

2. Why are **Employee** objects not being destroyed? Add code to **demopoly.cpp** to cause the **Employee** objects to be destroyed. Again predict the output and build and run the program to verify your understanding.

3. Why are the **SalaryEmployee** and **WageEmployee** destructors not being called? In what file do you need to make a change to fix this problem? Implement the fix. Build and run the program to verify that it is now behaving as it should. Is this last problem actually an error in this particular case? Under what circumstances would it be an error?

4. Add another concrete class **SalesEmployee** derived from **Employee**. A sales employee has a commission rate and a volume of sales. Pay is
   ```
         commission * sales.
   ```
   Implement changes in both the header file and code file of the employee classes, and modify the demo program **demopoly.cpp** to test the additional class. Does any modification need to be made to the function **PayReport** that prints out a report of pay of all employee stored in an array?

Lab 1A 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:

```
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()
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)");
      }
      ~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;
};

#endif


//  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());

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

Lab 1B Answers

1. See the starting files **employee.h**, **employee.cpp** and **demopoly.cpp** in the **Employee\Start3** directory. The following is the output when running the program.

```
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)
Sally   500
Wally   400
```

2. **Employee** objects are not being destroyed because **delete** is not called for the objects created by **new**. See **demopoly.cpp** in the **Employee\Step3** directory and in the hardcopy listings. When we run the program we get the following output:

```
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)
Sally   500
Wally   400
Employee::~Employee()
String::~String()
Employee::~Employee()
String::~String()
```

3. The **SalaryEmployee** and **WageEmployee** destructors are not being called because the destructor in **Employee** was not virtual. Changing the destructor to virtual (in file **employee.h**) resolves the problem (see **Employee\Step3** directory). The problem is not an error here, because the destructor does nothing (other than print a trace message). It would be an error in a case where the destructor did something substantive, such as deallocate dynamic memory allocated in the constructor.

4. See the **Employee\Step3** directory for code adding class **SalesEmployee**. No modification need to be made to the function **PayReport**.

Module 2:

Advanced C++ Topics

Top Level Directory: OI\AdvCpp

Chapter 1

Advanced Polymorphism and Inheritance

Advanced Polymorphism and Inheritance

Objectives

After completing this unit you will be able to:

- **Design and implement C++ classes that can be used in the same manner as the built-in language types.**
- **Describe the difference between public, protected, and private inheritance.**
- **Design and implement C++ classes that use inheritance, composition, and templates when appropriate.**
- **Use interface inheritance to hide from client programs the details of class definitions.**

Good Class Design

- **Elements of a well designed class:**
  - Can be used where any built
  - Provide for good abstract data type (ADT) or interface design that can be the basis of inheritance hierarchy
- **Use the Orthodox Canonical Form (OCF) (see Coplien's Advanced C++):**
  - Default Constructor
  - Copy Constructor
  - Assignment Operator
  - Destructor
- **Classes that use the OCF support assignment of objects, function returns, default initialization, and call by value parameters.**
- **Other operators, for conversion or manipulation may have to be added depending on how the class is used.**
- **Building a good inheritance hierarchy through abstraction and encapsulation reduces the complexity of the system. It may or may not promote software reuse in subsequent projects.**

String Class

```
class String
{
public:
    String(const char *str = "");
    String(const String& s);
    ~String();
    String operator=(const char *str);
    String& operator=(const String& s);
private:
    char *m_str;
    int m_length;

};

String::String(const String& s)
{
    m_length = s.m_length;
    m_str = new char[m_length + 1];
    strcpy(m_str, s.m_str);
}
String String::operator=(const char *str)
{
    delete [] m_str;
    m_length = strlen(str);
    m_str = new char[m_length + 1];
    strcpy(m_str, str);
    return *this;
}
String& String::operator=(const String& s)
{
    if (this == &s)      // special case s = s
        return *this;
    m_length = s.m_length;
    delete [] m_str;
    m_str = new char[m_length + 1];
    strcpy(m_str, s.m_str);
    return *this;
}
```

Lab 1A

Orthodox Canonical Form

The goal of this simple exercise is to reinforce in your mind how a class written in the orthodox canonical form can be used in the same way as a built-in type. In the C++ program you will write notice that while the String class is an improvement over the use of a char* pointer, its use naturally fits into C++.

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

Suggested time: 30 minutes

Public Inheritance

- **Inheritance determines the encapsulation level that the access control specifiers provide for member functions and data.**
- **Public Inheritance is of the form *class B: public A*.**
  - With public access control, member functions and data are available to any function in the program.
  - With private access control, member functions and data are only available to member functions in the class, but not any derived *classes*.
  - With protected access control, member functions and data are available to derived *classes*.
  - Derived *classes* access data through public or protected member functions of the *classes*.

Public Inheritance Example

- **To speed up development, the class IntList inherits from the class IntStack.**

```
class IntList : public IntStack
{
public:
    IntList():IntStack(100){};
    IntList(int size):IntStack(size){};
    ~IntList(){};
    void Add(int value) {Push(value);}
    int Remove() {int ret; ret = Pop();return ret;}
};

class IntStack
{
public:
    IntStack(long size = STACKSIZE);
    ~IntStack();
    void Push(int x);
    int Pop();
    void Print();
    int IsEmpty();
    int IsFull();
private:
    long stacksize;         // size of stack
    int *stack;             // stack data
    long top;               // top of stack
};
```

Public Inheritance Problems

- **IntStack's Push(), Pop() methods are visible to all users of IntList.**
  - This compromises the encapsulation because IntList cannot be reimplemented if Push() and Pop() are used.
  - Push() and Pop() cannot be made protected.
- **The fundamental problem is that public inheritance is *type* inheritance.**
  - *class IntList : public IntStack* means that IntList "is a" type of IntStack which is not really the case.
  - If you have an argument that takes a reference or a pointer to IntStack will take a pointer or reference to IntList.
  - As a type Push() and Pop() methods have to be public.
- **Examples of proper "type" or "is a" inheritance:**
  - A cat is a type of animal – a reference to an animal can be a cat.
  - A BMW is a type of car – a reference to a car can be a BMW.

Inheritance and Semantics

- **Developing classes requires you to partition the classes properly:**

```
class Fish {
public:
    virtual void LayEggs()
    ...
};
class Guppy : public Fish
```

  – But Guppies give birth to live young!

- **We need:**

```
class Fish
class ViviparousFish : public Fish {
public:
    virtual void GiveBirth();
    ...
};
class OviviparousFish : public Fish {
    public:
    virtual void LayEggs();
}
class Guppy : public ViviparousFish
```

- **Now the compiler will prevent:**

```
Guppy myGuppy;
MyGuppy.LayEggs();
```

- **If the abstraction of your problem does not require reproductive functions then you can ignore the issue.**

Private Inheritance

- **Private inheritance is *implementation* inheritance.**
  - Implementation is reused, the base class interface is hidden.

```
class IntList : private IntStack
{
public:
    IntList():IntStack(100){};
    IntList(int size):IntStack(size){};
    ~IntList(){};
    void Add(int value) {Push(value);}
    int Remove() {int ret; ret = Pop();return ret;}
};
```

- **Private Inheritance is of the form *class B: private A*.**
  - With public access control, member functions and data are private in the derived *class*.
  - With private access control, member functions and data are hidden in the derived *class*.
  - With protected access control, member functions and data are private in the derived *classes*.
  - The derived *class* accesses data through public or protected member functions of the base class.
- **Only the class that immediately derives from the base class can use the base class.**
- **Only the implementation is reused, none of the methods so the "is a" relationship is not required.**

Lab 1B

Public, Protected, and Private Inheritance and Access Scoping

The best way to understand the access scope for the public, protected, and private inheritance is to try a few simple examples and try to predict how the compiler will treat each case.

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

Suggested time: 30 minutes

Composition

- **Composition is an alternative to private inheritance.**
- **The composing class contains an instance of the composed class as a member:**

```
class IntList : public IntStack
{
public:
    IntList():stk(100){};
    IntList(int size):stk(size){};
    ~IntList(){};
    void Add(int value) {stk.Push(value);}
    int Remove() {return stk.Pop();}
private:
    IntStack stk;
};
```

- **Composition is a "has a" or "contains a" or "implemented using a" relationship.**
  - Person has a social security number.
  - A car contains an engine and four wheels.

Composition vs. Private Inheritance

- **If the relationship is "contains" or "has" composition should be used. But what about the "implemented using" relationship?**
- **If you need to redefine virtual functions or use protected members you need to use private inheritance.**
  - Protected constructors or assignment operators.
- **If there is no clear choice use whichever one is simplest to use.**
- **Both private inheritance and composition are *reuse* technologies.**

Templates vs. Inheritance

- **Templates are another reuse technology.**
- **Templates are used whenever the behavior does not depend on the type of the object.**

```
template<class T> class Array {
public:
    Array(int size = 10);
    ~Array();
    void SetAt(int i, const T& x);
    T GetAt(int i);
    void Print();
private:
    int m_size;
    T* m_array;};
```

- **The behavior of the array operations *SetAt* and *GetAt* do not depend on the type of object they contain.**
- **Inheritance would not be work for an Array base class container. What types would you use as parameters for virtual functions *SetAt* and *GetAt* in the base class?**
  - If you use void* pointers as parameters you will not be able to *SetAt* and *GetAt* objects such as cars or doubles, only pointers to them.
- **If the behavior depends on the type of object you must use virtual functions. You could use templates to create the LayEggs() or GiveBirth() functions for the fish class. You must use inheritance.**

Protected Inheritance

- **Protected Inheritance is of the form *class B: protected A*.**
  - With public access control, member functions and data are protected in the derived *class*.
  - With private access control, member functions and data are hidden in the derived *class*.
  - With protected access control, member functions and data are protected in the derived *classes*.
- **Infrequently used, allows public members to be protected in the derived class.**
- **Semantically unclear, can just use public inheritance and protected access control.**

Implementation Encapsulation

- **Part of good design is the separation of class definition from the implementation.**
- **The compiler, however, needs to know the size of a class to instantiate it. This often leads to "include file hell":**

```
#include "stringclass.h"
#include "zipcode.h"
#include "addressclass.h"
#include "dateclass.h"
#include "ssnclass.h"

class BankAccount
{
private:
  String  PrimaryAccountHolder;
  Address PrimaryAccountAddress;
  Address MailingAddress;
  Date    AccountOpenDate;
  SSN     TaxpayerId;
public:
  BankAccount();
  Virtual ~BankAccount();
  const char*   getAccountHolderName();
  const char*   getAccountAddress();
  const char*   getMailingAddress();
  const char*   getTaxpayerId();
};
```

- **You change the implementation, you have to recompile the client. For large systems this can be a major problem and inhibits the release of software revisions.**

Interface Inheritance

- **To avoid this define an abstract base class.**

```
class BankAccount {
public:
    virtual ~BankAccount();
    virtual const char* getAccountHolderName() const=0;
    virtual const char*   getAccountAddress()const = 0;
    virtual const char* getMailingAddress() const = 0;
    virtual const char* getTaxpayerId() const; = 0
};
```

- **Clients program with pointers to this BankAccount class. These pointers actually reference a concrete instance of this abstract base class.**

```
class ActualBankAccount : public BankAccount
{
public:
     ActualBankAccount(…) {…};
     virtual ~ActualBankAccount();
     const char*   getAccountHolderName();
     const char*   getAccountAddress();
     const char*   getMailingAddress();
     const char*   getTaxpayerId();
private:
     // actual implementation
```

- **The client obtains a pointer to an actual instance from some other function.**

Lab 1C

Investigation of Class Inheritance

This lab will build the IntList class using the IntStack class with various implementation techniques. Composition, private inheritance, public inheritance, and interface inheritance will be used to demonstrate implementing C++ classes.

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

Suggested time: 40 minutes

Summary

- **The Orthodox Canonical Form provides a format for the design of classes that can be used in the same manner as built-in types.**
- **Public Inheritance is “type” inheritance and implies an “is a” relationship.**
- **Public Inheritance hierarchies should reflect the system design abstraction. It may or may not promote code reuse.**
- **Private Inheritance is an implementation inheritance and implies an “implemented using” relationship.**
- **Protected Inheritance is semantically vague. It is not often used.**
- **Composition uses classes to implement functionality. It implies a “contains” or “has a” relationship.**
- **Templates implement reusable behavior that does not depend on the type of the object.**
- **Abstract base classes can be used to hide implementation details from a client and decouple the client from the details of the object’s implementation.**

Lab 1A

Orthodox Canonical Form

Introduction

The goal of this simple exercise is to reinforce in your mind how a class written in the orthodox canonical form can be used in the same way as a built-in type. In the C++ program you will write notice that while the String class is an improvement over the use of a char* pointer, its use naturally fits into C++.

Suggested time: **30 minutes**

Root Directory: **OI\AdvCpp**

Directory:	**Labs\Lab1A\OCF**	(working directory)
	Chap01\OCF\Step0	(backup copy of starter files)
	Chap01\OCF\Step1	(answer)

Files to Modify :

democonv.cpp
strn.h
strn.cpp

Instructions:

1. In the working directory there is a version of a **String** class in which there is implemented a second overloaded assignment operator for assigning a character pointer to a **String**. There is a test program **democonv.cpp** for testing assignments and conversions. Before running the program try to predict the output, including all constructors, destructors, assignments, and overloaded cast operations. Build and run the program. Redirect the output to a file so that you can save and study it.
   (a) How many copy constructors are invoked?
   (b) What conversions are involved in executing **`PrintString("Goodbye")`** ?
   (c) What conversions are involved in executing **`PrintCharPtr(a)`** ?

2. Replace the two lines

```
a = "Hello";
b = a;
```

by the single line

```
b = a = "Hello"
```

3. Build the program. What is wrong? Fix the problem.

Lab 1A

Answers

1. The output is

```
String::String(const char *str)
String::String(const char *str)
operator const char* () const
1st = Alpha
operator const char* () const
2nd = Beta
operator=(const char *str)
operator=(const String& s)
operator const char* () const
1st = Hello
operator const char* () const
2nd = Hello
String::String(const char *str)
operator const char* () const
String = Goodbye
String::~String()
operator const char* () const
char ptr = Hello
String::~String()
String::~String()
```

(a) There are no copy constructors invoked, because there is no explicit initialization of an object by another object, and no passing of a **String** object by value.
(b) First there is a conversion from **const char *** to **String** (by means of a constructor) and then a conversion from **String** to **const char *** (by overloaded cast operator).
(c) There is a conversion from **String** to **const char *** (by overloaded cast operator).

2. When attempting to do the assignments on one line the compile fails, because the overloaded **operator=(const char *)** returns a **void**. To fix the problem, make it return a **String**, following the pattern of **operator=(const String&)**. See the answer directory for a complete solution.

```
String String::operator=(const char *str)
{
    Trace("operator=(const char *str)");
    delete [] m_str;
    m_length = strlen(str);
    m_str = new char[m_length + 1];
    strcpy(m_str, str);
    return *this;
}
```

Lab 1B

Public, Protected, and Private Inheritance and Access Scoping

Introduction

The best way to understand the access scope for public, protected and private inheritance is to try a few simple examples and try to predict how the compiler will treat each case.

Suggested Time: 30 minutes

Root Directory: **OI\AdvCpp**

Directories:	**Labs\Lab1B\Access**	(do your work here)
	Chap01\Access\Step0	(backup copy of starter files)
	Chap01\Access\Step1	(answer to first part)
	Chap01\Access\Step2	(answer to second part)
	Chap01\Access\Step3	(answer to third part)
	Chap01\Access\Step4	(answer to fourth part)

Instructions

1. Before you build the starter project examine the code in the main routine and predict which lines of code the compiler will flag as illegal.
2. Comment out the illegal lines and build the project. Before you run the project predict what the output will be. Run the project and check your answer.
3. Change the inheritance derivation for class B to protected. Before you build the project try to predict what code the compiler will now flag as illegal.
4. Build the project to check your prediction. Run the project and make sure you understand the results.
5. Change the inheritance derivation for class B to private. Before you build the project try to predict what code the compiler will now flag as illegal.
6. Build the project to check your prediction. Run the project.
7. Change the inheritance derivation for class B to public. Suppose I wanted to prevent a call to the foo() member function in class A. How could I do that? Insert a call to b.foo() in the program. Compile the code to make sure your change compiles. Now define a protected member function foo() in class B. Recompile. What happens?
8. Remove the protected member function foo() from B. Define a public member function foo(int z) in B. Compile. Fix the compilation error. Now put in a call to b.foo() in the program. What compilation error do you get now? There is no polymorphism without virtual functions. If class B has to reimplement foo() then class B does not fulfill the "is a" relationship for public inheritance.

Lab 1C

Investigation of Class Inheritance

Introduction

This lab will build the IntList class using the IntStack class with various implementation techniques. Composition, private inheritance, public inheritance, and interface inheritance will be used to demonstrate implementing C++ classes.

Suggested Time: 40 minutes

Root Directory: **OI\AdvCpp**

Directories:		
	Labs\Lab1C\List	(do your work here)
	Examples\List\Step0	(backup copy of starter files)
	Examples\List\Step1	(answer to the first part)
	Examples\List\Step2	(answer to the second part)
	Examples\List\Step3	(answer to the third part)
	Examples\List\Step4	(answer to the fourth part)

Instructions

1. Verify that the starter code compiles.
2. Create a class IntList that uses public inheritance from IntStack to help implement is functionality. IntList will have a default constructor, a constructor that takes a list size, Add() and Remove() methods that use the Push() and Pop() methods from IntStack.
3. Inside the main routine create an instance of IntList. Invoke the Add() method three times with different values. Invoke the Push method once. Invoke the Remove() method four times. After each Remove() print out the value that was removed from the list. Why can we use the Push() method from the base class?
4. Change the derivation on the IntList class from public to private. Make the appropriate changes to get the program to work. What is the difference between public and private inheritance? Which is the proper one to use in this implementation?
5. Reimplement the IntList class using composition instead of inheritance. Which do you prefer composition or private inheritance. When would you have to use composition? When would you have to use private inheritance?
6. Define an abstract base class that defines the essential functions for the IntList. Reimplement the IntList class to derive from the abstract base class. Use composition to implement the IntList class. Rewrite the main program to use a pointer to the abstract base class instead of the IntList class. How does the client program create an instance of the IntList class? Hint: encapsulate the creation inside another function.

Module 3:

Fundamentals of STL

Top Level Directory: OI\FndStl

FndStl

Chapter 1

An Overview of Templates

An Overview of Templates

Objectives

After completing this unit you will be able to:

- **Define a function template.**
- **Employ the rules for disambiguation under specialization for template functions.**
- **Define a class template.**
- **Define templates with multiple type parameters.**

Templates

- ***Templates* provide a way of implementing a piece of code in a type independent way.**
  - Functions such as **sort**, **average**, **min** and **max** should be written as template functions since the code for each of them is independent of their type.
- **A *template function* is similar to a macro except with templates: there is no tedium about backslashes or parentheses, there are better diagnostics, and type checking is performed.**
  - One also need not worry about double expansions in expressions such as x++.
- **Templates may be classes as well as functions.**
  - Most data structures such as **Array**, **Stack**, **List**, and **Set** (to name a few) make good candidates for templates.

Overloading Functions

- **Note the following overloaded non template functions:**

```
double minimum(const double & a, const double & b)
{
    if ( a < b)
        return a;
    return b;
}
int minimum(const int & a, const int & b)
{
    if ( a < b)
        return a;
    return b;
}
string minimum(const string & a, const string & b)
{
    if ( a < b )
        return a;
    return b;
}
```

- **One could imagine other *minimum* functions.**
- **What about many *maximum* or *sort* functions?**

Template Functions

- **The previous functions differ only by their type.**
  - If the type could be made into a parameter, then the programmer could code one such function and instantiate it on any type.
  - In C++, parameterized types are achieved by using the **template** keyword.

Template Functions – Example

```
// Min.cpp

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

// template function
template < class T >
T minimum(const T & a,const T & b)
{
   if ( a < b)
      return a;
   return b;
}

int main( )
{
   int a = 5, b = 10;
   double x = 30.45, y = 57.35;
   string r("mike"), s("sue");
   cout << minimum(a,b) << endl; // instantiation
   cout << minimum(x,y) << endl; // "
   cout << minimum(r,s) << endl; // "
   return 0;
}
```

Specializing a Template Function

- **Special care must be taken in implementing template functions.**
  - One must be sure that operators have been correctly defined for any type instantiated for the function.
  - In some cases this means that a special (overloaded) function must be written.

Specializing a Template Function

- **The code below would get translated as shown:**

```
char *line = "there", *text = "hello";
cout << minimum(line, text) << endl;

char * minimum(const char * a, const char * b)
{
    if ( a  <  b)    // compares addresses
        return a;
    return b;
}
```

- **What is needed is a function specifically coded for the two *char* * types! (i.e. a non-template function!).**

```
char * minimum(const char *a, const char *b)
{
    if ( strcmp(a,b) < 0 )    // compares data
        return a;
    return b;
}
```

Disambiguation under Specialization

- **In the special case of a template function overloaded with a non template function, the following rules are used to disambiguate function calls.**

1. Examine all non-template calls!

```
if == 1,     done
if  > 1,     ambiguous
if == 0,     next step
```

2. Examine all template instances.

```
if == 1, done
if  > 1, ambiguous
if == 0, next step
```

3. Reexamine all non template instances, attempting to resolve the call as a non-template function.

Disambiguation under Specialization

- **Examples:**

```
char line[100] = "hello";
char word[100] = "there";
int a,b;
double x,y;

// match on non template call
cout << minimum(line,word);

// match on template <class int>
cout << minimum (a,b);

// match on template <class double>
cout  << minimum (x,y);

// NO MATCH
// cout << minimum (line,5);
```

Template Classes

- **In addition to functions, classes can also be templatized.**
  - For example, one could imagine a **Stack** class for integers, Strings, and Windows. Similarly, one could imagine a **List** class for Integers, Records, or some other type.
- **Observe the *IntStack* class:**

Template Classes

```
class IntStack
{
public:
   IntStack(int number = 10);
   ~IntStack( );
   void push(int value);
   int pop( );
private:
   int *data;
   int howmany;
   int top_of_stack;
};
IntStack::IntStack(int number)
 {
   data = new int[howmany = number];
   top_of_stack = 0;
}
IntStack::~IntStack()
{
   delete [ ] data;
}
void IntStack::push(int value)
{
   if ( top_of_stack >= howmany )
      throw overflow(top_of_stack, howmany);
   else
      data[top_of_stack++] = value;
}
int IntStack::pop( )
{
   if( top_of_stack < 0)
      throw underflow(top_of_stack);
   else
      return(data[--top_of_stack]);
}
```

Template Classes

- **Next, observe the DoubleStack class:**
  - Notice the differences between the two classes – **IntStack** and **DoubleStack**. They differ only by their type. The code is identical except for the type.

Template Classes

```
class DoubleStack
{
public:
   DoubleStack(int number = 10);
   ~DoubleStack( );
   void push(double value);
   double pop( );
private:
   double *data;
   int howmany;
   int top_of_stack;
};
DoubleStack::DoubleStack(int number)
 {
   data = new double[howmany = number];
   top_of_stack = 0;
}
DoubleStack::~DoubleStack()
{
   delete [ ] data;
}
void DoubleStack::push(double value)
{
   if ( top_of_stack >= howmany )
      throw overflow(top_of_stack, howmany);
   else
      data[top_of_stack++] = value;
}
double DoubleStack::pop( )
{
   if( top_of_stack < 0)
      throw underflow(top_of_stack);
   else
      return(data[--top_of_stack]);
}
```

Template Classes

- **The two classes above represent the same abstraction, a stack.**
  - They have the same functionality and the same implementation. They differ only by their type.
- **It's easy to imagine other classes such as the ones above.**
  - For example, **StringStack** or **ComplexStack**.
- **Classes which have identical interfaces and implementations except for their types should be implemented in C++ as a *template class* – a blueprint for a family of classes.**
- **An *array* provides a good abstraction for a template.**

An Array Template Class

```
template < class T > // T is the type parameter
class Array
{
    public:
        Array(int);
        Array(const Array & a);
        ~Array( );
        T & operator[](int index);
        int dimension( ) const;
        int operator==(const Array & a);
        Array<T> & operator=(const Array & a);
    private:
        T *elements;
        int size;
};
```

An Array Template Class

- **Each member function of a template class is itself a template function and must be coded as such. The notation:**

```
template < class T>
Array<T>::Array(int number)
```

defines this function as a template function and specifies it is from a template class.

```
template <class T>
Array<T>::Array(int number)
{
   elements = new T[size = number];
}
```

or

```
template <class T> Array<T>::Array(int number)
{
   elements = new T[size = number];
}
```

Instantiating a Template Class Object

- **Objects of template classes first need to be instantiated and then employed.**
  - When the compiler sees a template instantiation, it builds the class in accordance with the **template definition** by substituting the instantiated type in place of the **template parameter**.
- **Then, using a template class object is the same as using a non-template class object.**

Instantiating a Template Class Object

```
// An array of 10 integers
// <int>    is the parameterized type

   Array <int> IArray(10);

// An array of 10 doubles
// <double>    is the parameterized type

   Array <double> DArray(10);


   for(int i=0; i < IArray.dimension(); i++)
      cout << IArray[i] << endl;
```

A Non-member Function with a Template Argument

- **Suppose we wish to write a non member function to sum the elements of a template Array.**
- **This can be done in two ways.**
  - Generalize the function – make the function a template function.

```
template <class T>
T sum1(const Array<T> & x)
{
   T total  = 0;
      ...
   return total;
}
```

  - Make it type specific – do not make it a template function.

```
int  sum2(const Array<int> & x)
{
   int total  = 0;
   ..
   return total;
}
```

Friends of Template Classes

- **Template classes can have friend functions. Since friends are non-member functions, either of the above approaches could be used:**
  - Generalize the function – make the function a template function.
  - Make it type specific – do not make it a template function.

Friends of Template Classes

```
//    type specific
//
friend
ostream& operator<<(ostream & s,const Array<int>&);

      Corresponding To The Function

ostream& operator<<(ostream & s,const Array<int>&a)
{
   for (int i = 0; i < a.size; i++)
      s << a.elements[i];
   return s;
}

//    or template
//
friend
ostream& operator<<(ostream & s, const Array<T> &);

      Corresponding To The Function

template < class T>
ostream& operator<<(ostream & os,const Array<T>& a)
{
   for (int i = 0; i < a.size; i++)
      os << a.elements[i];
   return os;
}
```

Templates with Multiple Type Parameters

- **A template class represents a family of classes whose exact type is parameterized.**
- **The parameterization is not limited to one type.**
  - There can be more than one parameterized type.
  - Each parameter can be of any type – the same type or a different type.

Templates with Multiple Type Parameters

```
template <class L, class R>
class Pair {
public:
   Pair(const L & Left,const R & Right)
   : left_d(Left), right_d(Right) { }
   Pair() { }
   L getleft() const { return left_d; }
   void setleft(const L & clr){left_d = clr;}
   R getright() const { return right_d; }
   void setright(const R & crr){right_d=crr;}
private:
   L left_d;
   R right_d;
};
```

Templates with Multiple Type Parameters

- **Partial program using *Pair*:**

```
#include "Fraction.h"
#include <string>
#include <iostream>

int main( )
{
   using namespace std;
   string Name1("MERCK"), Name2("NETSCAPE");
   Fraction MRKPE(80,4), NETPE(35,2);  //PE's

   Pair <string, Fraction> pfolio[2];
   pfolio[0].setleft(Name1);
   pfolio[0].setright(MRKPE);
   pfolio[1].setleft(Name2);
   pfolio[1].setright(NETPE);
   for ( int i = 0; i < 2; i++)
   {
      cout << pfolio[i].getleft( ) << endl;
      cout << pfolio[i].getright( ) << endl;
   }
/* if(Lookup(pfolio, 2, Name1) == -1)
      cout << Name1 << " not found\n";
   else
      cout << Name1 << " found\n";
*/
   return 0;
}
```

Non Class-type Parameters for Template Classes

- **Thus far, each template has had parameterized types.**
- **Template classes may take expression parameters, i.e. non class-type parameters.**

```
template < class T, int howmany = 20>
class FArray   {
public:
   FArray( );
   ..
private:
   T elements[howmany];
};


const SIZE = 512;
FArray < int, 10> fsia;
FArray < String, SIZE > sa;
```

Comments Regarding Templates

- **There exist strict rules regarding templates:**

1. Template classes expect proper constructors.
2. Template classes expect proper functions.
3. If a template function expects a reference, then a reference (not a value) must be passed!
4. A Derived object cannot be passed in place of a Base object.
5. There is no way to specify that all template instantiations are friends.
6. Template classes can be type arguments to a template class.

Comments Regarding Templates

```
// (a) proper constructors
template<class T>Array<T>::Array(int size)
{
   data = new T[size];  // define
      ...            // proper
      ...            // constructor
}

// (b) proper functions
template <class T> T min(T a, T b)
{
   return a < b ? a : b;
}

// (e)
friend class Array<int>;               // ok
friend template <class T> class Array; // error


//  (f) template class as a type argument
Matrix <  Array<int>  > M; // watch spaces
```

M is a template object whose elements are integer arrays.

Templates and Inheritance

- **A template class represents a family of classes.**
  - Each of the classes in the family is used as if it is a non-template class.
  - Thus, you can derive classes from a template class in the same way that you can derive from a non-template class.
  - Some of the notation may be a little strange at first. Ultimately one gets used to the syntax.

```
// ListInternal.cpp

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

template <class T>
class ListWithString : public list<T>
{
public:
   ListWithString(string n, T *beg, T *end)
   : list<T>(beg, end), name(n)
   {}
   string getName() { return name; }
private:
   string name;
};

...
```

- **In the code above, the functions are defined inside the class definition.**

Templates and Inheritance

- **If the function is defined outside the class definition the syntax looks like this:**

```
// ListExernal.cpp

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

template <class T>
class ListWithString : public list<T>
{
public:
   ListWithString(string n, T *beg, T *end);
   string getName();
private:
   string name;
};

template <class T>
ListWithString<T>::
ListWithString(string n, T *beg, T *end)
: list<T>(beg, end), name(n) {}

template <class T>
string ListWithString<T>::getName()
{
   return name;
}

...
```

Templates and Inheritance

- **Here is the test program, that works for either version of the *ListWithString* class.**
  - We derived the new class from the standard **list** class.

```
int main()
{
   int x[10] = { 1,2,3,4,5,6,7,8,9,10 };
   ListWithString<int> L("Mylist", x, x + 10);

   cout << L.getName() << endl;

   list<int>::iterator begin = L.begin();
   list<int>::iterator end = L.end();

   while(begin != end)
      cout << *begin++ <<  " " << endl;
   return 0;
}
```

Exercises for Chapter 1

- **The exercises for this module have a somewhat different format than those for the first two modules.**
  - Find start code in numbered subdirectories of **FndStl\Labs**. Note that the starter code does not necessarily compile.
  - Find solution code in numbered subdirectories of **FndStl\Solutions**.

1. Write the **Lookup** function referenced on page 263. Lookup is a non member function which takes a **Pair** array, an **int**, and a **String**.

2. Write a template **Stack** class and then implement the following:

```
Array < Stack<int, 10> >  Array_of_Stacks(5);
```