Programming C++ Lecture 5

Howest, Fall 2013
Instructor: Dr. Jennifer B. Sartor
Jennifer.sartor@elis.ugent.be

Templates

- Function and class templates – you specify with a single code segment an entire range of related (overloaded) functions or classes (function or class-template specializations).
- Generic programming!
- Templates are stencils of pretty shapes

- Template specializations are tracings we make of stencils – same shape but maybe different colors.

Remember Function Templates

- We can do function overloading

```
int boxVolume(int side) {
    return side * side * side;
}
double boxVolume(double side) {
    return side * side * side;
}
```

- Why define 2 functions that look identical, but have different types?
- Overloading that is more compact and convenient = function templates. Only write it once!

Function Templates

- Template

  template <class T> //or template <typename T>

  ```
  T boxVolume(T side) {
      return side * side * side;
  }
  ```

- C++ compiler automatically generates separate function template specializations for each type the function is called with.
- T is placeholder for actual data type
- int result = boxVolume(3); double result = boxVolume(6.2);

Class Stack Template Example

```
//Stack.h
template< typename T >
class Stack {
public:
    Stack(int = 10);
    ~Stack() { delete [] stackPtr; }
    bool push( const T & ); //push element
    bool pop( T & ); //pop element
    bool isEmpty() const {
        return top == -1;
    }
    bool isFull() const {
        return top == (size -1);
    }
private:
    int size;
    int top;
    T *stackPtr;
};
```

```
template< typename T >
Stack< T >::Stack(int s)    //constructor
    : size( s > 0 ? s : 10 ),
      top( -1 ),
      stackPtr( new T[size] ) { }
template< typename T >
bool Stack< T >::push(const T &pushValue) {
    if (!isFull()) {
        stackPtr[++top] = pushValue;
        return true;
    } return false;
}
template< typename T >
bool Stack< T >::pop(T &popValue) {
    if (!isEmpty()) {
        popValue = stackPtr[top--];
        return true;
    } return false;
}
```

Test Stack

```
#include <iostream>
using namespace std;
#include “Stack.h”
int main() {
    Stack< double > doubleStack(5);
    double doubVal = 1.1;
    while (doubleStack.push(doubVal))
        doubVal += 1.1;
    while (doubleStack.pop(doubVal))
        cout << doubVal << ‘ ‘;

    Stack< int > intStack;  //default size
    int intVal = 1;
    while (intStack.push(intVal ))
        intVal ++;
    while (intStack.pop(intVal ))
        cout << intVal << ‘ ‘;
    return 0;
}
```

- Testing double stack vs. int stack is very similar pattern.
- You could create a template function to test your template class!

More Details

- Because a compiler compiles template classes on demand, it requires the definition (usual .cpp) to be in the same file as the declaration (usual .h).
- http://www.cplusplus.com/doc/tutorial/templates/
- Make sure operators used in template class are implemented if used with user-defined type!
  - Our Stack requires user-defined type to have default constructor and assignment operator.

Specifics of Templates

- You can have nontype template parameters too
  - template< typename T, int elements > //compile time constant
  - Stack< double, 100 > mostRecentSalesFigures;
  - .h could contain member: T stackHolder[elements];
- Type parameter can specify default type
  - template< typename T = string >
  - Stack<> jobDescriptions;
- Explicit specialization for a particular type
  - template<>
  - class Stack< Employee > { … };

Input to Main Function

```
int main(int argc, char* argv[]) {
    cout << "Number of arguments is " << argc << endl;
    for (int i = 0; i < argc; i++) {
        cout << "Argument " << i << " is " << argv[i] << endl;
    }
    ifstream inputFile(argv[1], ios::in);

}
```

Instantiating Objects Example

1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. cout << mPtr->getName(); //what does this print?
7. mPtr->print(); //and this?

Instantiating Objects Example

1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. cout << mPtr->getName(); //Jill
7. mPtr->print(); //Jill

Instantiating Objects Example

1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. Employee *ePtr = &e1;
7. cout << ePtr->getName() << ePtr->getSalary(); //result?
8. ePtr->print(); //what function does this call?

Instantiating Objects Example

1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. Employee *ePtr = &e1;
7. cout << ePtr->getName() << ePtr->getSalary(); //Jack 65000
8. ePtr->print(); //Employee.print which calls Member.print

Instantiating Objects Example

1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. Employee *ePtr = &e1;
7. mPtr = &e1; //is this ok? Base class pointer to derived class?

Instantiating Objects Example

```
1. #include "Member.h"
2. #include "Employee.h"
3. Member m1("Jill");
4. Employee e1("Jack", 65000);
5. Member *mPtr = &m1;
6. Employee *ePtr = &e1;
7. mPtr = &e1;    //Yes, valid;  all Employees are Members
8. ePtr = &m1;    //this valid? Derived class pointer to base class?
```

Instantiating Objects Example

1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. Employee *ePtr = &e1;
7. mPtr = &e1;   //Yes, valid; all Employees are Members
8. ePtr = &m1;   //No, not all Members are Employees;
   //compiler error

Instantiating Objects Example

1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. mPtr = &e1; //yes, this is valid; all Employees are Members
7. cout << mPtr->getName(); //what does this print?
8. cout << mPtr->getSalary(); //this ok?
9. mPtr->print(); //what function does this call?

Instantiating Objects Example

```
1. #include "Member.h"
2. #include "Employee.h"
3. Member m1("Jill");
4. Employee e1("Jack", 65000);
5. Member *mPtr = &m1;
6. mPtr = &e1;
7. cout << mPtr->getName();          //Jack
8. cout << mPtr->getSalary();        //compiler error
9. mPtr->print();                    //calls Member's print: Jack
```

Introducing Polymorphism

- Member *mPtr = &e1;   mPtr->print();
- By default, method that is called depends on the type of the handle, not the type of the object
- Polymorphism enables the compiler to call the more specific method, i.e. call based on the type of object dynamically.
- Because all derived class objects ARE base class objects, 1 base class pointer can enable calls to any number of derived class methods.
  - Program "in the general" rather than "in the specific"

Polymorphism!

- Member *mPtr = &e1;   mPtr->print();
- To get the Employee print function to be called, the method has to be declared **virtual** (in the .h)
- For virtual functions, the type of the object being pointed to determines function call, not type of handle.
  - At execution time we determine what function to call (not compile time), so it is done dynamically.
  - This is called **dynamic binding**

Polymorphism!

- Member *mPtr = &e1;   mPtr->print();
- Dynamic binding with virtual functions only works with pointer and reference handles (you need a level of indirection).
  - Member m1("Jill");
  - m1.print(); resolved at compile time => static binding!
- Base class declares functions as virtual, and implicitly for all derived classes that function is virtual (whether declared thus or not – virtualness is inherited).
- Derived class function can override/redefine base class regular or virtual function, or takes on base class's implementation if not defined

Base Class Example

```
class Member {
public:
    Member(string name);
    Member( Member const &);
    Member& operator= (Member const &);
    ~Member();

    string getName() const;
    void setName(string name);
    virtual void print() const;
private:
    string myName;
};
```

Derived Class Example

```
#include "Member.h"
class Employee : public Member {
public:
    Employee(string name, double money);
    Employee( Employee const &);
    Employee& operator= (Employee const &);
    ~Employee ();

    double getSalary() const;
    void setSalary(double money);
    virtual void print() const;  //keywork here unnecessary, but good practice.
private:
    double salary;
};
```

Instantiating Objects Example

1. #include “Member.h”
2. #include “Employee.h”
3. Member m1(“Jill”);
4. Employee e1(“Jack”, 65000);
5. Member *mPtr = &m1;
6. mPtr = &e1;
7. cout << mPtr->getName(); //Jack
8. mPtr->print(); //calls Employee’s print: Jack 65000

Kinds of Assignments

- Base class pointer -> base class object = FINE
  - Invokes base class functionality
- Derived class pointer -> derived class object = FINE
  - Invokes derived class functionality
- Base class pointer to derived class object = FINE
  - Will invoke base class functionality unless functions declared virtual, then will invoke derived class functionality
- Derived class pointer to base class object = COMPILER ERROR (unless explicit cast)

Base class is a Derived class?

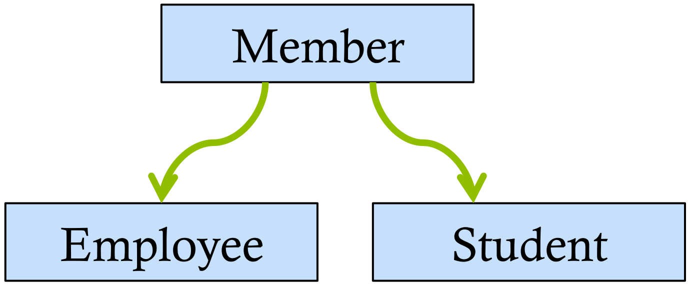
- Derived class pointer -> base class object
  - Could downcast?
  - DANGEROUS!

```
Member *mPtr;

…

Employee *ePtr = static_cast< Employee* > (mPtr);

ePtr->getSalary();
```

- We will see a safe way to do this – with dynamic cast.

Derived Class Example2

```
#include "Member.h"
class Student: public Member {
public:
    Student(string name, int id);
    Student(Student const &);
    Student& operator= (Student const &);
    ~Student ();

    int getUniqueID( ) const;
    void setUniqueID (int id);
    virtual void print( ) const;  //keywork here unnecessary, but good practice.
private:
    int uniqueID;
};
```



Example of Polymorphism

```
vector < Member* > members(4);

members[0] = new Employee("Alice", 60000); //name & salary

members[1] = new Student("Bob", 987654);  //name & uniqueID

for (size_t i = 0; i < members.size(); i++) {
    members[i]->print();  //polymorphic behavior here

}
```

Example of Polymorphism

```
vector < Member* > members(4);

members[0] = new Employee("Alice", 60000); //name & salary

members[1] = new Student("Bob", 987654);  //name & uniqueID

for (size_t i = 0; i < members.size(); i++) {
    members[i]->print();
    //what if we want to change salary here – give everyone a raise?

}
```

Example of Polymorphism

```
vector < Member* > members(4);
members[0] = new Employee("Alice", 60000);
members[1] = new Student("Bob", 987654);  //name & uniqueID
for (size_t i = 0; i < members.size(); i++) {
    Employee *ePtr = dynamic_cast < Employee* > (members[i]);
    if (ePtr != 0) { //if downcast succeeded, we have Employee*
        ePtr->setSalary((ePtr->getSalary()) * 1.1);
    }
    members[i]->print();
}
```

Memory Management

```
vector < Member* > members(4);
members[0] = new Employee(“Alice”, 60000);
members[1] = new Student(“Bob”, 987654);  //name & uniqueID
for (size_t i = 0; i < members.size(); i++) {
    Employee *ePtr = dynamic_cast < Employee* > (members[i]);
    if (ePtr != 0) { //if downcast succeeded
        ePtr->setSalary((ePtr->getSalary()) * 1.1);
    }
    members[i]->print();
}
for (size_t i = 0; i < members.size(); i++) {
    delete members[i];
}
```

Destructors

- What happens if we call delete on a base class pointer to a derived class object?
  - Call base class destructor?
  - Derived class destructor?
  - Error?

Destructors

- What happens if we call delete on a base class pointer to a derived class object?
  - Call base class destructor?
  - Derived class destructor?
  - Error?
- This is undefined and can cause compiler warnings. BAD

Virtual Destructors

- When virtual methods exist, declare destructor **virtual** in base class.
- All derived classes destructors are then by default virtual as well (even though they have different names).
- Enables proper destruction of derived classes from base class pointers (behavior undefined if destructor not virtual)
- Constructors CANNOT be virtual.

Base Class Example

```
class Member {
public:
    Member(string name);
    Member( Member const &);
    Member& operator= (Member const &);
    virtual ~Member();

    string getName() const;
    void setName(string name);
    virtual void print() const;
private:
    string myName;
};
```

Abstract Classes

- An abstract class provides a common public interface for its class hierarchy. It is usually the base class.
- Class is made abstract by declaring 1 or more of its virtual functions to be "pure" in .h, no implementation in .cpp
  - virtual void print() const = 0;
  - Abstract classes are never instantiated (lack implementation)
- Abstract classes provide a framework but are incomplete. Derived classes must define missing pieces.

Pure Virtual

- Every concrete derived class *must* override all base-class pure virtual functions with concrete implementations.
  - If not overridden, derived class is abstract (can't be instantiated).
- A virtual-only function in base class has an implementation and gives derived class an option to override (as with regular functions).

Abstract Classes

- Abstract class can have data members and concrete functions (constructors/destructors) which go by normal inheritance rules.
- Can use pointers to abstract classes to use polymorphic functionality on all concrete derived classes.
  - Useful with container classes (vector of abstract base class)
  - Can use iterator to iterate over items in container class