OOP 3200 – Object Oriented Programming II

Week 6 – Generics

Week 6 Overview

- Generics
- In-class Exercise 5

Course Outline

Week	Date	Topic	Evaluation	Weight
1	Sep 09, 2020	- Course Orientation - Object-Oriented Programming overview - Partnering for labs		
2	Sep 16, 2020	REVIEW OF CLASSES & OBJECTS in C++ - Encapsulation - Object Attributes and Behaviours - Classes: The Blueprint for Objects - Relationship Between Class and Objects - Static Class Members - Friend Functions	In-Class Exercises 1: (2%) C++ Assignment 1: (6%)	8
3	Sep 23, 2020	CLASS OPERATORS AND DATA TYPE CONVERSIONS in C++: - Creating Class Operators - How Methods Are Shared - Data Type Conversions	In-Class Exercises 2: (2%) C++ Assignment 2: (6%) C++ Quiz 1: (5%)	13
4	Sep 30, 2020	INHERITANCE AND POLYMORPHISM in C++: - Class Inheritance - Polymorphism - Virtual functions - Interfaces / Abstract Classes	In-Class Exercises 3: (2%) C++ Assignment 3: (6%)	8
5	Oct 07, 2020	COLLECTIONS in C++: - Dynamic Object Creation and Deletion - Pointers As Class Members / Destructors - Copy Constructors / Copy Assignment Operators	In-Class Exercises 4: (2%) C++ Assignment 4: (6%)	8
6	Oct 14, 2020	GENERICS in C++ - Method templates - Class Templates	In-Class Exercises 5: (2%)	2
7	Oct 21, 2020	THE C++ STANDARD TEMPLATE LIBRARY (STL): - Vectors and Linked Lists - Stacks and Queues - Maps and Sets	In-Class Exercises 6: (2%) C++ Assignment 5: (6%) C++ Quiz 2: (5%)	13
READING WEEK				

Objectives

- This week, you will learn about:
  - Function Templates (Review)
  - The **swap** function template (example)
  - Using **Operators** in Function Templates
  - Function Templates with Multiple Types
  - **Overloading** with Function Templates
  - Class Templates
  - Class Templates and Inheritance

Function Templates

CONCEPT:

- A function template is a "generic" function that can work with different **data types**.
- The programmer writes the specifications of the function but substitutes parameters for data types.
- When the compiler encounters a call to the function, it **generates code** to handle the specific data type(s) used in the call.

Function Templates (continued)

- **Overloaded functions** make programming convenient because only one function name must be remembered for a set of functions that perform similar operations.

- Each of the functions, however, must still be written **individually** . For example, consider the following overloaded square functions.

```
int square(int number)
{
    return number* number;
}

double square (double number)
{
    return number* number;
}
```

- The only differences between these two functions are the **data types** of their **return values** and their **parameters**.

- In situations like this, it is more convenient to write a **function template** rather than an overloaded function.

Function Templates (continued)

- **Function templates** allow you to write a single function definition that works with many different data types, instead of having to write a separate function for each data type used.

- A function template is not an actual function, but a "mold" the compiler uses to **generate** one or more functions.

- When writing a function template, you **do not** have to specify actual types for the parameters, return value, or local variables.

- Instead, you use a **type parameter** to specify a generic data type.

- When the compiler encounters a call to the function, it examines the data types of its arguments and generates the function code that will work with those data types.

Function Templates (continued)

- Here is a **function template** for the square function:

```
template <class T>
T square( T number)
{
    return number* number;
}
```

- The beginning of a function template is marked by a **template prefix**, which begins with the key word **`template`**.

- Next is a set of **angled brackets < >** that contain one or more generic data types used in the template.

- A generic data type starts with the key word **`class`** , followed by a parameter name that stands for the data type.

- The example just given only uses one, which is named **T**. (If there were more, they would be separated by commas.) After this, the function definition is written as usual, except the type parameters **are substituted** for the actual data type names.

Function Templates (continued)

- In the example, the function header reads:

```
T square( T number)
```

- **T** is the **type parameter**, or **generic data type**. The header defines square as a function that returns a **value of type T** and uses a parameter, number, which is also of **type T**.

- As mentioned before, the compiler examines each call to square and fills in the appropriate data type for **T**.

Function Templates (continued)

❖ For example, the following call uses an **`int`** argument:

```
int y, X = 4;
y = square(x);
```

❖ This code will cause the compiler to **generate** the function:

```
int square(int number)
{
    return number* number;
}
```

❖ while the statements:

```
double y, d = 6.2
y = square(d);
```

❖ will result in the **generation** of the function:

```
double square(double number)
{
    return number* number;
}
```

The swap Function Template

- In many applications, there is a need for **swapping** the contents of two variables of the same type. For example, while sorting an array of **integers**, there would be a need for the function:

```
void swap(int &a, int &b)
{
    int temp = a;
    a = b;
    b = temp;
)
```

- whereas while sorting an **array string objects**, there would be a need for the function:

```
void swap(string &a, string &b)
{
    string temp = a;
    a = b;
    b = temp;
)
```

The swap Function Template (continued)

- Because the only difference in the coding of these two functions is the **type of the variables** being swapped, the logic of both functions and all others like them can be captured with **one template function**:

```
template<class T>
void swap(T &a, T &b)
{
    T temp = a;
    a = b;
    b = temp;
}
```

- Such a template function is available in the libraries that come with standard C++ compilers. The function is declared in the **`algorithm`** header file.

Using Operators in Function Templates

- The square function template shown earlier in this section applies the operator ***** to its parameter. The square template will work correctly if the type of the parameter passed to it supports the ***** operator.

- For example, it works for numeric types such as **`int`**, **`long`**, and **`double`** because all these types have a multiplication operator *****.

- In addition, the square template will work with any **user-defined** class type that overloads the operator *****.

- Errors will result if square is used with types that **do not support** the operator *****.

- Always remember that templates will only work with types that support the operations used by the template.

Using Operators in Function Templates (continued)

- For example, a **class** can only be used with a template that applies the relational operators such as **<**, **<=**, and **!=** to its type parameter if the class **overloads** those operators .

- For example, because the **`string`** class overloads all the relational operators, it can be used with template functions that compute the minimum of an array of items.

```
// This program illustrates the use of function templates.
#include <string>
#include <iostream>
using namespace std;

// Template for minimum of an array
template <class T>
T minimum(T arr[ ], int size)
{
   T smallest = arr[0];
   for (int k = 1; k < size; k++)
   {
      if (arr[k] < smallest)
         smallest = arr[k];
   }
   return smallest;
}
```

Function Templates with Multiple Types

❖ More than **one generic type** may be used in a function template.

❖ The following program is an example of a function that takes as parameters a **list of three values** of any printable type, prints out the list in order, and then print s out the list in reverse.

❖ The type parameters for the template function are represented using the identifiers T1, T2, and T3.

```
// Template function
template <class T1, class T2, class T3>
void echoAndReverse(T1 a1, T2 a2, T3 a3)
{
   cout << "Original order is: "
        << a1 << "  " << a2 << "  " << a3 << endl;
   cout << "Reversed order is: "
        << a3 << "  " << a2 << "  " << a1 << endl;
}
```

Overloading with Function Templates

- Function templates may be **overloaded**. As with regular functions, function templates are overloaded by having different parameter lists.

- For example, there are two overloaded versions of the **sum** function. The first version accepts **two arguments**, and the second version accepts **three**.

```
template <class T>
T sum(T val1, T val2)
{
   return val1 + val2;
}

template <class T>
T sum(T val1, T val2, T val3)
{
   return val1 + val2 + val3;
}
```

- There are other ways to perform overloading with function templates as well.

- For example, a program might contain a **regular** (non-template) version of a function as well as a **template version** . As long as each has a different parameter list, they can coexist as **overloaded functions** .

Defining Template Functions

- In defining template functions, it may be helpful to start by writing a **non-template version** of the function and then **converting** it to a template after it has been tested.

- The conversion is then achieved by **prefixing** the function definition with an appropriate template header, say:

```
template <class T>
```

- and then systematically replacing the relevant type with the generic type **T**.

NOTE: Beginning with C++ 11, you may use the key word `typename` in place of `class` in the template prefix. So the template prefix can be written as `template <typename T>`.

Class Templates

CONCEPT:

❖ Templates may also be used to create **generic classes** and **abstract data types**.

❖ **Function templates** are used whenever we need several different functions that have the same problem-solving logic but differ only in the types of the parameters they work with.

❖ **Class templates** can be used whenever we need **several classes** that only differ in the **types of some of their data members** or in the types of the parameters of their member functions.

Class Templates (continued)

- **Declaring** a class template is similar to declaring a function template:

- You write the class using identifiers such as **T**, **T1**, **T2** (or whatever other identifier you choose) as generic types, and then you **prefix** the class declaration with an appropriately written template header.

Class Templates (continued)

❖ For example, suppose that we wish to define a class that represents an array of a **generic type** and adds an overloaded operator **[ ]** that performs bounds checking.

❖ Calling our class **`SimpleVector`** and putting in the appropriate data members and constructors, we arrive at the template:

```
template <class T>
class SimpleVector
{
    unique_ptr<T []> aptr;
    int arraySize;
public:
    SimpleVector(int);                    // Constructor
    SimpleVector(const SimpleVector &);   // Copy constructor
    int size() const { return arraySize; }
    T &operator[](int);                   // Overloaded [] operator
    void print() const;                   // Outputs the array elements
};
```

Class Templates (continued)

- This above class template will store elements of **type T** in a dynamically generated array.

- This explains why the pointer **`aptr`**, which will point to the base of this array, is declared to be of type **`T []`** .

- We have used a `unique_ptr` for the type of **`aptr`** because a **`SimpleVector`** object will not share the dynamically allocated array with any other part of the program.

- Likewise, the **overloaded array subscription operator** returns a value of type **T**.

- Notice, however, that the value returned by the **`size`** member function and the member **`arraySize`** are both of type **`int`**. This makes sense because the number of elements in an array is always an **integer**, regardless of the type of element the array stores.

Class Templates (continued)

- You can think of the **`SimpleVector`** template as a **generic pattern** that can be specialized to create classes of **`SimpleVector`** that hold **`double`**, **`long`**, **`string`**, or any other type that you can define.

- The rule is that you form the name of such an actual class by appending a list of the actual types, enclosed in **angled brackets** , to the name of the class template:
  - **`SimpleVector<double>`** is the name of a class that stores arrays of **`double`**.
  - **`SimpleVector<string>`** is the name of a class that stores arrays of **`string`**.
  - **`SimpleVector<char>`** is the name of a class that stores arrays of **`char`**.

Class Templates (continued)

- Here is an example of defining a **`SimpleVector`** object by using the convert constructor to create an array of **`10`** elements of type **`double`**:

```
SimpleVector<double> dTable (10);
```

- Defining a member function of a template class inside the class is straightforward: an example is furnished by the definition of **`size()`** in the **`SimpleVector`** class.

- To define a member function **outside the class**, you must prefix the definition of the member function with a template header that specifies the list of type parameters, and then within the definition, use the **name of the class** template followed by a list of the type parameters in angled brackets whenever you need the name of the class.

Class Templates (continued)

❖ Let us use the **operator [ ]** function to illustrate the definition of a member function outside the class.

```
template <class T>
T &SimpleVector<T>::operator[](int sub)
{
   if (sub < 0 || sub >= arraySize)
      throw IndexOutOfRangeException(sub)
   return aptr[sub];
}
```

❖ In this definition, the **name of the class** is needed just before the scope resolution operator, so we have **`SimpleVector<T>`** at that place. As another example, consider the definition of the convert constructor:

```
template <class T>
SimpleVector<T>::SimpleVector(int s)
{
   arraySize = s;
   aptr = make_unique<T[]>(s);
   for (int count = 0; count < arraySize; count++)
       aptr[count] = T();
}
```

Class Templates (continued)

- There is an **exception** to the rule of attaching the **list of type parameters** to the name of the template class.

- The **list**, and the **angled brackets** that enclose it, can be omitted whenever the name of the class is **within the scope of the template class**.

- Thus the list can be omitted when the name of a class is being used anywhere within the class itself, or within the local scope of a member function that is being defined outside of the class.

Class Templates (continued)

❖ For example, the copy constructor:

```
template <class T>
SimpleVector<T>::SimpleVector(const SimpleVector &obj)
{
    arraySize = obj.arraySize;
    aptr = make_unique<T[]>(arraySize);
    for (int count = 0; count < arraySize; count++)
        aptr[count] = obj[count];
}
```

❖ does not need to append the **`<T>`** to the **`SimpleVector`** that denotes the type of its argument.

Class Templates and Inheritance

❖ Inheritance can be applied to class templates. For example, in the following template, **`SearchableVector`** is derived from the **`SimpleVector`** class .

Contents of SearchVect.h

```
#include "SimpleVector.h"

template <class T>
class SearchableVector : public SimpleVector<T>
{
public:
   // Constructor.
   SearchableVector(int s) : SimpleVector<T>(s)
   { }
   // Copy constructor.
   SearchableVector(const SearchableVector &);
   // Additional constructor.
   SearchableVector(const SimpleVector<T> &obj) :
       SimpleVector<T>(obj) { }
   int findItem(T);
};

//*******************************************
// Definition of the copy constructor.      *
//*******************************************
template <class T>
SearchableVector<T>::
```

Class Templates and Inheritance

```
SearchableVector(const SearchableVector &obj) :
    SimpleVector<T>(obj)
{
}

//*******************************************
// findItem takes a parameter of type T     *
// and searches for it within the array.    *
//*******************************************
template <class T>
int SearchableVector<T>::findItem(T item)
{
    for (int count = 0; count < this->size(); count++)
    {
        if (this->operator[](count) == item)
            return count;
    }
    return -1;
}
```

- The **SearchableVector** class demonstrates that a class template maybe derived from another class template.

- In addition, class templates may be derived from ordinary classes, and ordinary classes may be derived from class templates.