Generic Programming Algorithm and Data Abstraction

In this week

- Generic Programming: Motivation
- Generic Programming in C++: Templates
- Algorithm Abstraction in C++
- Data Abstraction in C++
- Deriving from C++ templated classes

Generic Programming

- So far we have seen several algorithms (most often than not implemented as functions) in order to solve different problems
- Similarly, we have seen several classes in order to design new data types representing real world objects
- Unfortunately all our functions and classes were designed with specific data types in mind and implemented for that specific data type

Generic Programming

- For example, consider the sequential search algorithm
- At first we implemented it for an integer data type arrays
- Our implementation therefore would not work properly for, say string data types
- When we wanted the exact the same sequential search algorithm for string data type; we had to re-implement it from scratch

Generic Programming

- Similarly, consider the **SmartArray** class we designed in order to have our own array data structure with several functionalities that made it much more flexible and useful than the standard C++ arrays
- Again the **SmartArray** class was designed and implemented for only integer data types and does not work properly, say for double data types
- If we want such an array with string data type elements then we must re-implement it from scratch for that specific data type

Generic Programming

- But why not use the same algorithm or class for different data types instead of re-implementing the same thing again and again?
- Well because by design C++ is strict when it comes to data types and we must adhere to the language requirements
- But wouldn't it be an amazing feature if we were able to **implement an algorithm or a class only once** and then **use the same algorithm or class with any data type of our choice**?

Generic Programming

- **Definition:-** A programming methodology whereby an algorithm or a class is designed and implemented without any specific data type in mind and whereby the actual data type is specified when an application makes use of the algorithm or the class is known as **generic programming**
- **Definition:- Algorithm abstraction** is a programming methodology whereby an **algorithm** is designed and implemented without any specific data type in mind so that it works with any data types
- **Definition:- Data abstraction** is a programming methodology whereby a **class** is designed and implemented without any specific data type in mind so that it works with any data types
- Abstracting either an algorithm or a class or both helps us achieve generic programming

Generic Programming in C++
C++ Templates

- In C++, generic programming is achieved with the help of **C++ templates**
- A **C++ template** is a **blue print** for a data type
- It tells C++ compiler that the blue print will be made concrete (it gets substituted by a specific data type) during program run time
- Some C++ authors use the term **templated programming** instead of **generic programming**

Algorithm Abstraction in C++

- Consider the following C++ program that creates several arrays with different data types and uses sequential search algorithm in order to search for a specified search values in the arrays
- Our aim is to design and implement **only one** sequential search algorithm (function) in order to perform the search
- We do not want to implement several functions and rely on function overloading

Algorithm Abstraction in C++

```
int main()
{
    //Construct several arrays of different data types
    const int size = 10;
    int *A1 = new int[size];
    double *A2 = new double[size];
    string *A3 = new string[size];

    srand(time(0));

    //Populate the arrays
    for (int i = 0; i < size; i++)
    {
        A1[i] = rand() % 21 + 5;
        A2[i] = ((1.0 * rand()) / RAND_MAX) * 15.0 + 5.0;
        int random = rand() % 5;
        A3[i] = (random == 0 ? "Paul" :
                (random == 1 ? "Jannet" :
                (random == 2 ? "Kevin" :
                (random == 3 ? "Sara" : "CMPT"))));
    }

    //Print the arrays
    cout << "Here are the arrays created..." << endl << endl;
    cout << "Array A1\tArray A2\tArray A3" << endl;
    cout << "========\t========\t========" << endl;
    for (int i = 0; i < size; i++)
        cout << A1[i] << "\t\t" << A2[i] << "\t\t" << A3[i] << endl;
    cout << endl;
```

Algorithm Abstraction in C++

```
    //Perform some searches using sequential search algorithm
    int s1 = rand() % 21 + 5;
    double s2 = A2[rand() % size];
    string s3 = "Sara";

    int ans1 = sequentialSearch(A1, size, s1);
    int ans2 = sequentialSearch(A2, size, s2);
    int ans3 = sequentialSearch(A3, size, s3);

    //Display search results
    if (ans1 == -1)
        cout << s1 << " is not found in the array A1" << endl;
    else
        cout << s1 << " is found in the array A1 at index " << ans1 << endl;

    if (ans2 == -1)
        cout << s2 << " is not found in the array A2" << endl;
    else
        cout << s2 << " is found in the array A2 at index " << ans2 << endl;

    if (ans3 == -1)
        cout << s3 << " is not found in the array A3" << endl;
    else
        cout << s3 << " is found in the array A3 at index " << ans3 << endl;

    system("Pause");
    return 0;
}
```

Algorithm Abstraction in C++

- The generic sequential search algorithm implemented as a templated C++ function and that can search on any of the arrays shown in the application program will therefore look like as follows

```
template <class T>
int sequentialSearch(const T* A, const int size, const T& searchValue)
{
    for (int i = 0; i < size; i++)
    {
        if (A[i] == searchValue)
            return i;
        else
            continue;
    }
    return -1;
}
```

- The **template <class T>** prefix above the function header tells the C++ compiler that the function just below that line of code is a templated function

Algorithm Abstraction in C++

- As shown above any C++ function can be made generic; all we need is to make it a templated function
- The actual code in a templated C++ function is almost identical to the code we would find in a non-templated function; all that changes is any specific data type that we would like to be templated is replaced by the template name
- Parameter passing to a templated C++ function can be made by any of the parameter passing methods available in C++ language: value, pointer or reference
- If a function is templated then the function must make use of the template data type name in its parameter(s); otherwise a syntax error occurs when we try to call the function
- The choice of name for the template (**T** in the sequential search templated function above) is arbitrary and we can use any valid C++ identifier; although **T** is very commonly used by C++ programmers
- Although we often use **template <class T>** when designing a templated function, it is also allowed to use **template <typename T>**

Algorithm Abstraction in C++

- When an application calls a templated C++ function (with a specified data type), then every statement inside the templated function must be well defined for the specified data type; otherwise syntax error will occur
- For example, there is the == operator in the templated sequential search function
- Therefore the == operator must be well defined for the actual data types of the arguments passed to the function

Algorithm Abstraction in C++

- The return data type of a templated C++ function can be a templated type
- For example, the following templated function returns the maximum value of its array argument irrespective of the data type of the array; so long as the > operator is well defined for the data type of the array
- Write an application to test the function. In your test application, create several arrays of different data types and call the same function to compute the maximum elements of each of the arrays

```
template <typename WeirdDataTypeName>
WeirdDataTypeName getMaximumElement(const WeirdDataTypeName *A, const int size)
{
    WeirdDataTypeName m = A[0];
    for (int i = 1; i < size; i++)
    {
        if (A[i] > m)
            m = A[i];
    }
    return m;
}
```

Algorithm Abstraction in C++

- A templated C++ function can have more than one templated types for different data types in its parameters
- See the following example that takes two arguments (same or different data types) and returns the sum of the two arguments. Analyze and determine the output
- Of course for this function to be valid, both the **static_cast** and the **+ operator** shown in the function body must be well defined for the data types specified

```
template <typename T1, class T2>
T1 sum_up(const T1 x, const T2 y)
{
    T1 result = x + static_cast<T1>(y);
    return result;
}

int main()
{
    int a = 5;
    float b = 2.7;

    cout << "sum_up(" << a << ", " << b << ") = " << sum_up(a, b) << endl;
    cout << "sum_up(" << b << ", " << a << ") = " << sum_up(b, a) << endl;

    system("Pause");
    return 0;
}
```

Data Abstraction in C++

- In order to demonstrate data abstration with C++ templates, let us reconsider the SmartArray class that was discussed in the past and redesign it now so that it becomes an array of any data type
- We will rename the class name **SmarterArray** to emphasize the change of design
- When a class is templated, it requires the template prefix just above the class declaration
- All its **member functions declarations inside the class** declaration **do not need** the template prefix
- However **implementations of member functions outside the class declaration** will **require the template prefix** for each of the member functions independently
- Importantly, **any friend function** that is **declared inside the class requires** the template prefix; as well as in its implementation outside
- The identifier name chosen in the template prefix for the friend function(s) declarations or implementations of functions outside the class can be the same identifier as the identifier chosen for the class template prefix or different; it doesn't matter. See below…

Data Abstraction in C++

```
template <class T>
class SmarterArray
{
private:
    T *A;
    int size;

public:
    //Constructors
    SmarterArray();
    SmarterArray(const T *A, const int &size);
    SmarterArray(const SmarterArray<T> &L); //Copy constructor

    //Assignment operator
    SmarterArray<T>& operator = (const SmarterArray<T> &L);

    //Destructor
    ~SmarterArray();

    //Getters, Setters, operators and other functions
    int getSize() const;
    T& operator[](const int &index) const;
    int find(const T &e) const;
    void append(const T &e);
    bool remove(const T &e);
    SmarterArray<T> operator - (const SmarterArray<T> &L) const;

    //Friend functions need to be explicitely shown as templated even inside the class declaration
    template<class T>
    friend ostream& operator << (ostream &outputStream, const SmarterArray<T> &L);
};
```

Data Abstraction in C++

- As can be seen here, the **SmarterArray** class declaration is almost identical with the **SmartArray** class declaration except
  - There is template prefix before the class declaration
  - The pointer member variable is templated
  - The non-default constructor pointer parameter is templated
  - The copy constructor parameter is templated
  - The return data type and the parameter of the assignment operator are templated
  - The indexing operator return data type is templated
  - The member functions find, append, remove, and **-** operator have templated parameter
  - The output stream friend function has template blue print before the function declaration inside the class.
- Now the class name is no more ~~**SmarterArray**~~
- Rather it is **SmarterArray<T>** to reflect the fact that this is a templated class
- The implementations of the member and friend functions will be as follows
- Note carefully the template prefix before each function

Data Abstraction in C++

```
template <class T>
SmarterArray<T>::SmarterArray()
{
    this->size = 0;
}

template <class T>
SmarterArray<T>::SmarterArray(const T *A, const int &size)
{
    this->size = size;
    if (this->getSize() > 0)
    {
        this->A = new T[this->getSize()];
        for (int i = 0; i < this->getSize(); i++)
            this->A[i] = A[i];
    }
}

template <class T>
SmarterArray<T>::SmarterArray(const SmarterArray<T> &L) //Copy constructor
{
    this->size = L.getSize();
    if (this->getSize() > 0)
    {
        this->A = new T[this->getSize()];
        for (int i = 0; i < this->getSize(); i++)
            this->A[i] = L[i];
    }
}
```

Data Abstraction in C++

```
template <class T>
SmarterArray<T>& SmarterArray<T> :: operator = (const SmarterArray<T> &L)
{
    //Check for self asssignment. If so, do  nothing.
    if (this == &L)
        return *this;
    //Delete the left hand side object's memory
    this->~SmarterArray();
    //Now copy the right hand side to the left
    this->size = L.getSize();
    if (this->getSize() > 0)
    {
        this->A = new T[this->getSize()];
        for (int i = 0; i < this->getSize(); i++)
            this->A[i] = L[i];
    }
    return *this;
}

template <class T>
SmarterArray<T>::~SmarterArray()
{
    if (this->getSize() > 0)
    {
        delete[] this->A;
        this->size= 0;
    }
}
```

Data Abstraction in C++

```
template <class T>
int SmarterArray<T>::getSize() const
{
    return this->size;
}

template <class T>
T& SmarterArray<T>::operator[](const int &index) const
{
    if (index < 0 && index >= this->getSize())
    {
        cout << "ERROR! Index out of bounds." << endl;
        abort();
    }
    return this->A[index];
}

template <class T>
int SmarterArray<T>::find(const T &e) const
{
    for (int i = 0; i < this->getSize(); i++)
        if (this->A[i] == e)
            return i;   //return index if found
    return -1;  //return -1 to mean not found
}
```

Data Abstraction in C++

```
template <class T>
void SmarterArray<T>::append(const T &e)
{
    //First create a temporary array whose size is this->size+1
    T *temp = new T[this->getSize() + 1];

    //Copy the elements of this->A to temp
    for (int i = 0; i < this->getSize(); i++)
        temp[i] = this->A[i];

    //Copy the element to be appended to temp
    temp[this->getSize()] = e;

    //Delete the existing array this->A
    if (this->getSize() > 0)
        delete[] this->A;

    //Make the array this->A to point to temp and increment the size
    this->A = temp;
    this->size += 1;
}
```

Data Abstraction in C++

```
template <class T>
bool SmarterArray<T>::remove(const T &e)
{
    int index = this->find(e);
    if (index == -1)
        return false;
    else
    {
        //First create a temporary array whose size is this->size-1
        T *temp = new T[this->getSize() - 1];

        //Copy the elements of this->A to temp except the element at index
        for (int i = 0; i < index; i++)
            temp[i] = this->A[i];
        for (int i = index+1; i < this->getSize(); i++)
            temp[i-1] = this->A[i];

        //Delete the existing array this->A
        if (this->getSize() > 0)
            delete[] this->A;

        //Make the array this->A to point to temp and decrement the size
        this->A = temp;
        this->size -= 1;
        return true;
    }
}
```

Data Abstraction in C++

```
template <class T>
SmarterArray<T> SmarterArray<T>::operator - (const SmarterArray<T> &L) const
{
    //Returns the elements of this that are not found in L
    SmarterArray<T> A;
    for (int i = 0; i < this->getSize(); i++)
    {
        int index = L.find(this->A[i]);
        if (index == -1)
            A.append(this->A[i]);
    }
    return A;
}

template <class T>
ostream& operator << (ostream &outputStream, const SmarterArray<T> &L)
{
    outputStream << "[";
    for (int i = 0; i < L.getSize() - 1; i++)
        outputStream << L[i] << ", ";
    if (L.getSize() > 0)
        outputStream << L[L.getSize() - 1];
    outputStream << "]";
    return outputStream;
}
```

Data Abstraction in C++

- Now we can create several **SmarterArray** objects of different data types as we please; all we need to remember is to specify the underlying data type of the array during object instantiation
- For example, in the following test application we instantiate three **SmarterArray** objects with int, double and string underlying data types and work with them seamlessly
- **With one exception!!!** If we separate the header file of the class, the implementation of the class and the application on to different files; then we must include our class implementation file in our application file using the include directive (see below)

Data Abstraction in C++

```
#include "SmarterArray.h"
#include "SmarterArray.cpp"

int main()
{
    //Declare several SmarterArray objects
    SmarterArray<int> A1;               //Default array of integers
    double x[3] = {2.4, 1.2, 5.8};
    SmarterArray<double> A2(x, 3);      //Non-default array of doubles
    SmarterArray<string> A3;            //Default array of strings

    //Populate the SmarterArrayobjects
    srand(time(0));
    for (int i = 0; i < 10; i++)
    {
        if (rand() % 2 == 0)
            A1.append(rand() % 21 + 5);
        else
        {
            int random = rand() % 5;
            A3.append(random == 0 ? "Paul" :
                    (random == 1 ? "Jannet" :
                    (random == 2 ? "Kevin" :
                    (random == 3 ? "Sara" : "CMPT"))));
        }
    }

    //Print the objects
    cout << "The SmarterArray object A1 is " << A1 << endl;
    cout << "The SmarterArray object A2 is " << A2 << endl;
    cout << "The SmarterArray object A3 is " << A3 << endl;

    system("Pause");
    return 0;
}
```

Data Abstraction in C++

- A templated class can also have more than one templated data types…
- See for example the following class. Implement the class and test your class using the program given below and then compare your output with the output given below to see the correctness of your work

```
template <class K, class V>
class Map   //A mapping from one array to another array
{
private:
    K  *arr1;   //The first array
    V  *arr2;   //The second array
    int size;   //The size of the arrays (they have equal size)

public:
    Map();  //Default constructor. size = 0, pointers not initialized
    Map(const K *A, const V *B, const int &size);   //Non-default constructor
                                                    //Deep copy A to arr1 and B to arr2
    Map(const Map<K, V> &m);    //Copy constructor
    Map<K, V>& operator = (const Map<K, V> &m); //Assignment operator
    ~Map(); //Destructor. Delete both arrays and set size = 0
    V& operator [](const K &key) const;     //Find an index value s.t. arr1[index] == key.
                                            //If such index exists then return arr2[index]. Otherwise system("exit")
    K& operator [](const V &value) const;   //Find an index value s.t. arr2[index] == value.
                                            //If such index exists then return arr1[index]. Otherwise system("exit")
    void append(const K &key, const V &value);  //Append key to arr1 and value to arr2

    //Friend functions
    template <class K, class V>
    friend ostream& operator << (ostream &out, const Map<K, V> &m); //Print the mapping vertically nicely
};
```

Data Abstraction in C++

- Here is a program to test the templated Map class

```
int main()
{
    //Let us seed the random number generator to a fixed integer so that we all get the same output in Visual Studio
    srand(5);
    //The following array lists some cities around Vancouver
    int citiesSize = 20;
    string citiesArray[] = {"Victoria", "Nanaimo", "Calgary", "Edmonton", "Kamloops", "Prince George", "Kelowna",
                            "Seattle", "Belingham", "Whistlet", "Squamish", "Chilwak", "Abbotsford", "Penticton",
                            "Fort McMurray", "Prairie", "Yellow Knife", "Horseshoe Bay", "Langley", "White Rock"};
    //Let us create a map m1 listing these cities and their distances in Km from Vancouver
    Map<string, double> m1;
    //Print the map m1
    cout << "The initial map m1 is " << m1 << endl;
    //Append the cities and their distances from Vancouver to m1
    for (int i = 0; i < citiesSize; i++)
    {
        string city = citiesArray[i];
        int distance = rand() % 40 + 20;
        m1.append(city, distance);
    }
    //Let us print our map m1
    cout << "The map created m1 is now... " << m1<< endl;
    //Print the distance of Nanaimo from Vancouver
    cout << "Nanaimo is " << m1["Nanaimo"] << "km away from Vancouver." << endl;
    //Print one city that is 50km away from Vancouver. If no such city exists, the program will crash
    cout << "A city that is 50km away from Vancouver is " << m1[50] << endl;
    //Create a copy of the map m1
```

Data Abstraction in C++

```
    //Create a copy of the map m1
    Map<string, double> m2 = m1;
    //Delete the map m1
    m1.~Map();
    //Now print the map m1 after deleting it
    cout << "After deleting, the map m1 is now " << m1 << endl;
    //Print the map m2
    cout << "The map m2 that was copied from m1 still is..." << m2 << endl;
    //Assign m2 to m1
    m1 = m2;
    //Print the map m1
    cout << "After assigning m2 to m1, the map m1 is now " << m1 << endl;
    system("Pause");
    return 0;
}
```

- The output corresponding to this test program is provided below

```
The initial map m1 is Empty map

The map created m1 is now...
Victoria                34
Nanaimo                 33
Calgary                 35
Edmonton                29
Kamloops                40
Prince George           50
Kelowna                 27
Seattle                 29
Belingham               52
Whistlet                20
Squamish                54
Chilwak                 21
Abbotsford              33
Penticton               56
Fort McMurray           46
Prairie                 47
Yellow Knife            23
Horseshoe Bay           47
Langley                 50
White Rock              51

Nanaimo is 33km away from Vancouver.
A city that is 50km away from Vancouver is Prince George
After deleting, the map m1 is now Empty map

The map m2 that was copied from m1 still is...
Victoria                34
Nanaimo                 33
Calgary                 35
Edmonton                29
Kamloops                40
Prince George           50
Kelowna                 27
Seattle                 29
Belingham               52
Whistlet                20
Squamish                54
Chilwak                 21
Abbotsford              33
Penticton               56
Fort McMurray           46
Prairie                 47
Yellow Knife            23
Horseshoe Bay           47
Langley                 50
White Rock              51
```

```
After assigning m2 to m1, the map m1 is now
Victoria                34
Nanaimo                 33
Calgary                 35
Edmonton                29
Kamloops                40
Prince George           50
Kelowna                 27
Seattle                 29
Belingham               52
Whistlet                20
Squamish                54
Chilwak                 21
Abbotsford              33
Penticton               56
Fort McMurray           46
Prairie                 47
Yellow Knife            23
Horseshoe Bay           47
Langley                 50
White Rock              51

Press any key to continue . . .
```