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++

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 the algorithm to search on arrays storing integer values
- Our implementation therefore would not work for arrays storing string values
- When we wanted the exact 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 to store integer values and would not be able to store double values
- If we want to store double values then we must re-implement the class 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?
- The reason is the fact that C++ programming language is strongly typed and we must adhere to the language specifications
- But wouldn't it be more efficient 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**?
- This is where generic programming play a role!

Generic Programming

- **Definition:-** A programming methodology whereby an algorithm or a class is designed and implemented without specifying a data type 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 specifying a data type so that it works with any data types
- **Definition:- Data abstraction** is a programming methodology whereby a **class** is designed and implemented without specifying a data type 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 all the search operations
- 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;
```

Don't forget...

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

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 C++ function template 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 function template

Algorithm Abstraction in C++

- As shown above any C++ function can be made generic; all we need is to make it a function template
- The actual code in a function template is almost identical to the code we would find in a standard 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 function template can be made by any of the parameter passing methods available in C++ language: **by value, by pointer or by 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 function template 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 function template, it is also allowed to use **template <typename T>**

Algorithm Abstraction in C++

- When an application calls a function template (with a specified data type), then every statement inside the function template must be well defined for the specified data type; otherwise syntax error will occur
- For example, there is the == operator in the sequential search function template
- 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 function template can be a template type as well
- For example, the following function template 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

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

Test Program (See the previous example)

```
//Compute the maximum elements of the arrays
int m1 = getMaxValue(A1, size);
double m2 = getMaxValue(A2, size);
string m3 = getMaxValue(A3, size);

cout << "Maximum element of A1 = " << m1 << endl;
cout << "Maximum element of A2 = " << m2 << endl;
cout << "Maximum element of A3 = " << m3 << endl;
```

Algorithm Abstraction in C++

- A function template can have more than one templates 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 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
- Friend functions may use same or different template name

Data Abstraction in C++

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

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

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

    //Destructor
    ~SmarterArray(); //perform memory clean up

    //Getters, Setters, operators and other functions
    int getSize() const; //return the number of elements
    T& operator[](const int &) const; //return element at index
    int findElement(const T &) const; //return index of search value
    void append(const T &); //insert at the end
    bool erase(const int &); //erase element at index

    //Friend functions
    template<class T1> //Some compilers require a different template name
    friend ostream& operator << (ostream &, const SmarterArray<T1> &);
};
```

Don't forget...

```
#include <iostream>
#include <cassert>
#include <string>
#include <ctime>
using namespace std;
```

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 findElement and append have both templated parameter
  - The output stream friend function has template prefix before the function declaration inside the class and a templated parameter
- Now the class name as a data type is no more ~~**SmarterArray**~~
- Rather it is **SmarterArray<T>** to reflect the fact that this is a class template
- The implementations of the member and friend functions will be as follows
- Note carefully the template prefix before each function definition

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

Data Abstraction in C++

```
template <class T>
T& SmarterArray<T>::operator[](const int &index) const
{
    assert(index >= 0 && index < this->size);
    return this->A[index];
}
template <class T>
int SmarterArray<T>::findElement(const T &value) const
{
    for (int i = 0; i < this->size; i++)
        if (this->A[i] == value)
            return i;
    return -1;
}
template <class T>
void SmarterArray<T>::append(const T &value)
{
    //First create a temporary array whose size is this->size+1
    int new_size = this->size + 1;
    T *temp = new T[new_size];

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

    //Copy the element to be appended to temp
    temp[this->size] = value;

    //Destruct the object
    this->~SmarterArray();

    //Make the array this->A to point to temp and adjust the size
    this->A = temp;
    this->size = new_size;
}
```

Data Abstraction in C++

```
template <class T>
bool SmarterArray<T>::erase(const int &index)
{
    if (index < 0 || index >= this->size)
        return false;
    else if (this->size == 1)//only one element in the *this object
        this->~SmarterArray();
    else
    {
        //First create a temporary array whose size is this->size-1
        int new_size = this->size - 1;
        T *temp = new T[new_size];
        //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->size; i++)
            temp[i-1] = this->A[i];
        //Destruct the object
        this->~SmarterArray();
        //Make the array this->A to point to temp and adjust the size
        this->A = temp;
        this->size = new_size;
    }
    return true;
}
template <class T1>
ostream& operator << (ostream &cout, const SmarterArray<T1> &L)
{
    cout << "[";
    for (int i = 0; i < L.size - 1; i++)
        cout << L[i] << ", ";
    if (L.size > 0)
        cout << L[L.size - 1];
    cout << "]";
    return cout;
}
```

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

Data Abstraction in C++

```
int main()
{
    cout << "Testing constructors, appending, and output streaming operator" << endl;
    //Declare several SmarterArray objects
    SmarterArray<int> A1; //Default SmarterArray of integers
    double x[3] = {2.4, 1.2, 5.8};
    SmarterArray<double> A2(x, 3); //Non-default SmarterArray of doubles
    SmarterArray<string> A3; //Default SmarterArray of strings

    //Populate the SmarterArrayobjects
    srand(time(0));
    for (int i = 0; i < 10; i++)
    {
        if (rand() % 2 == 0)
            A1.append(rand() % 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;
```

Data Abstraction in C++

```
    //Test getSize, indexing operator, and searching for elements
    cout << "Testing getSize, indexing operator, and searching for elements..." << endl;
    cout << "A1 has " << A1.getSize() << " elements" << endl;
    int index = A2.findElement(-2.1);
    if (index == -1)
        cout << "-2.1 is not found in " << A2 << endl;
    else
        cout << "-2.1 is found in " << A2 << " at index " << index << endl;
    A2[1] = -2.1;
    cout << "A2 is now modified to " << A2 << endl;
    cout << "-2.1 is now found in " << A2 << " at index " << A2.findElement(-2.1) << endl;
    cout << endl;

    //Test copy constructor, assignment operator, and destructor
    cout << "Testing copy constructor, assignment operator, and destructor..." << endl;
    SmarterArray<string> A4(A3);
    A3.~SmarterArray();
    SmarterArray<double> A5;
    A5 = A2;
    A2.~SmarterArray();
    cout << "The SmarterArray object A1 is now " << A1 << endl;
    cout << "The SmarterArray object A2 is now " << A2 << endl;
    cout << "The SmarterArray object A3 is now " << A3 << endl;
    cout << "The SmarterArray object A4 is now " << A4 << endl;
    cout << "The SmarterArray object A5 is now " << A5 << endl;
    cout << endl;

    //Test erase element
    A1.erase(0);
    A1.erase(A1.getSize()-1);
    cout << "After erasing its first and last elements, A1 is now " << A1 << endl;

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

Data Abstraction in C++

- A class template can also have more than one templates
- We will demonstrate this with a class named **Map** that has two **SmarterArray** member variables known as the **Keys array** and the **Values array**
- The Map class represents a mapping from a set of Keys to a set of Values
- The Keys and Values arrays always have the same size by design
- We say a key at a given index of the Keys array is mapped to a value at the same index of the Values array
- This means whenever we append to a Map object we append a key-value pair together. Similarly, whenever we remove from a Map object then we remove a key-value pair

Data Abstraction in C++

Don't forget...

```
#include <iostream>
#include <cassert>
#include <string>
#include <ctime>
using namespace std;
```

```
template <class K, class V>
class Map
{
private:
    SmarterArray<K> A1; //The keys of the map
    SmarterArray<V> A2; //The values of the map

public:
    //Constructors
    Map(); //Default constructor
    Map(const Map<K, V>&); //Copy constructor. Deep copy.
    //Assignment operator
    Map<K, V>& operator = (const Map<K, V>&); //Assignment operator. Memory clean up and deep copy.
    //Destructor
    ~Map(); //Destructor.
    //Getters, Setters, operators and other functions
    int getSize() const; //Return the size of the map
    int findKey(const K&) const; //Return the index of the first element of the Keys array == the argument. Return -1 if not found.
    int findValue(const V&) const; //Return the index of the first element of the Values array == the argument. Return -1 if not found.
    K getKey(const V&) const; //Assert the argument is found in the Values array and then return the first key with the given value
    V getValue(const K&) const; //Assert the argument is found in the Keys array and then return the first value with the given key
    K getKeyAtIndex(const int&) const; //Assert the index argument and then return the key at the given index
    V getValueAtIndex(const int&) const; //Assert the index argument and then return the value at the given index
    void setKeyAtIndex(const int&, const K&); //Assert the index argument and then set the key at the given index
    void setValueAtIndex(const int&, const V&); //Assert the index argument and then set the value at the given index
    void append(const K&, const V&); //Append the key-value pair to the calling object
    template <class K1, class V1>
    friend ostream& operator << (ostream&, const Map<K1, V1>&); //Output streaming operator
};
```

Data Abstraction in C++

- Now we proceed with the member and friend functions definitions
- We start with the default constructor
- Since our member variables are SmarterArray objects, then they are already constructed as empty SmarterArray objects during their declaration in the Map class
- This means we don't need any code for the default constructor of the Map class
- Thus the default constructor member function for the Map class will look like as follows

```
template <class K, class V>
Map<K, V>::Map()
{
    //No code is needed here.
    //The member variables will automatically be initialized as default objects
}
```

Data Abstraction in C++

- Next we define the copy constructor, assignment operator, and destructor member functions. Once again we note that the SmarterArray class has all these member functions already defined and we can use them here as we wish. Thus these functions definitions will look like as follows

```
template <class K, class V>
Map<K, V>::Map(const Map<K, V>& M)
{
    A1 = M.A1;
    A2 = M.A2;
}
template <class K, class V>
Map<K, V>& Map<K, V>::operator = (const Map<K, V>& M)
{
    A1 = M.A1;
    A2 = M.A2;
    return *this;
}
template <class K, class V>
Map<K, V>::~Map()
{
    A1.~SmarterArray();
    A2.~SmarterArray();
}
```

Data Abstraction in C++

- We now present the remaining member and friend functions definitions and a test program as shown below

```
template <class K, class V>
int Map<K, V>::getSize() const
{
    return A1.getSize();
}
template <class K, class V>
int Map<K, V>::findKey(const K& key) const
{
    return A1.findElement(key);
}
template <class K, class V>
int Map<K, V>::findValue(const V& value) const
{
    return A2.findElement(value);
}
template <class K, class V>
K Map<K, V>::getKey(const V& value) const
{
    int index = A2.findElement(value);
    assert(index != -1);
    return A1[index];
}
template <class K, class V>
V Map<K, V>::getValue(const K& key) const
{
    int index = A1.findElement(key);
    assert(index != -1);
    return A2[index];
}
```

```
template <class K, class V>
K Map<K, V>::getKeyAtIndex(const int& index) const
{
    assert(index >= 0 && index < A1.getSize());
    return A1[index];
}
template <class K, class V>
V Map<K, V>::getValueAtIndex(const int& index) const
{
    assert(index >= 0 && index < A2.getSize());
    return A2[index];
}
template <class K, class V>
void Map<K, V>::setKeyAtIndex(const int& index, const K& key)
{
    assert(index >= 0 && index < A1.getSize());
    A1[index] = key;
}
template <class K, class V>
void Map<K, V>::setValueAtIndex(const int& index, const V& value)
{
    assert(index >= 0 && index < A2.getSize());
    A2[index] = value;
}
template <class K, class V>
void Map<K, V>::append(const K& key, const V& value)
{
    A1.append(key);
    A2.append(value);
}
```

Data Abstraction in C++

```
template <class K1, class V1>
ostream& operator << (ostream& cout, const Map<K1, V1>& m)
{
    if (m.getSize() == 0)
        cout << "[Empty Map]" << endl;
    else
    {
        cout << endl;
        cout << "Keys \t\t Values" << endl;
        cout << "==== \t\t ======" << endl;
        for (int i = 0; i < m.getSize(); i++)
            cout << m.A1[i] << "\t\t" << m.A2[i] << endl;
    }
    return cout;
}
int main()
{
    //Test default constructor
    Map<string, double> m1;
    cout << "m1 is " << m1 << endl;

    //Test append and getSize member functions
    const int size = 5;
    string city[size] = {"Burnaby", "Surrey", "New West", "Delta", "Coquitlam"};
    double distance[size] = {10.8, 20.5, 15.4, 21.8, 18.1};
    for (int i = 0; i < size; i++)
        m1.append(city[i], distance[i]);
    cout << "Now m1 is " << m1 << endl;
    cout << "m1 has " << m1.getSize() << " key-value pair elements in it." << endl;

    //Test copy constructor
    Map<string, double> m2(m1);
    cout << "m2 is " << m2 << endl;
```

Data Abstraction in C++

```
    //Test destructor
    m1.~Map();
    cout << "Now m1 is " << m1 << endl;
    cout << "m2 is still " << m2 << endl;

    //Test assignment operator
    m1 = m2;
    cout << "Now m1 is " << m1 << endl;
    m2.~Map();
    cout << "m2 is destructed. m2 is now " << m2 << endl;

    //Test findKey, findValue, getKey, getValue, getKeyAtIndex, and getValueAtIndex member functions
    cout << "The city Delta is found in the keys array at index " << m1.findKey("Delta") << endl;
    cout << "A city whose distance from Vancouver is 18.1 is found at index " << m1.findValue(18.1) << endl;
    cout << "A city whose distance from Vancouver is 18.1 is " << m1.getKey(18.1) << endl;
    cout << "The distance of New West from Vancouver is " << m1.getValue("New West") << endl;
    cout << "The city at index 2 of the Keys array is " << m1.getKeyAtIndex(2) << endl;
    cout << "The distance at index 4 in the Values array at index " << m1.getValueAtIndex(4) << endl;

    //Test the setKeyAtIndex and setValueAtIndex member functions
    m1.setKeyAtIndex(0, "North Vancouver");
    cout << "The city at index 0 of the Keys array is " << m1.getKeyAtIndex(0) << endl;
    m1.setValueAtIndex(0, 17.7);
    cout << "The distance at index 0 in the Values array at index " << m1.getValueAtIndex(0) << endl;
    cout << "At last m1 is " << m1 << endl;

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

This completes the Map class declaration, definition, and a test program