Part: 61

File I/O in C++: Read/Write in the Same Program & Closing Files

In this tutorial, we'll learn about creating a program that will read from a file and write to the file in the same program using a constructor.

Before jumping on to the main thing, we'll first give ourselves a quick revision of the things we had learned previously.

We had learned about the three most useful classes when we talk about File I/O, namely,

1. **`fstreambase`**
2. **`ifstream`**
3. **`ofstream.`**

All the above three classes can be used in a program by first including the header file, **`fstream.`**

Reading File Operation Output:

We learnt reading from a file using ifstream. Below snippet will help you recollect the same.

```
  string st;
// Opening files using constructor and reading it
  ifstream in("this.txt"); // Read operation
  in>>st;
```

Writing File Operation Output:

We learnt reading from a file using **`ofstream`**. Below snippet will help you recollect the same.

```
string st = "Harry bhai";
// Opening files using constructor and writing it
ofstream out("this.txt"); // Write operation
out<<st;
```

Let me make these codes functional in the same program for you to easily understand the workflow.

Suppose we have a file named sample60.txt in the same directory, we can easily call the file infinite number of times in the same program only by maintaining different connections for different purposes, using

```
<object_name>.close();
```

Now, let's move on to our systems. Open your editors as well. Don't forget to include the header file, **`<fstream>.`**

Follow these steps below to first write into the empty file:

1. Create a text file **"`sample60.txt`"** in the same directory as that of the program.
2. Create a string variable *name*.
3. Create an object *`hout`***`(name it whatever you wish)`** using **`ofstream`** passing the text file, sample60.txt into it. This establishes a connection between your program and the text file.
4. Take input from the user using **`cin`** into the name **`string. (You can write manually as well)`**
5. Pass this name string to the object *hout*. The string name gets written in the text file.
6. Disconnect the file with the program since we are done writing to it using *`hout.close()`*.

Since the file has been disconnected from the program, we can connect it again for any other purpose in the same program independently.

Follow these steps below to read from the file we just wrote into:

1. Create a string variable *content*.

2. Create an object *hin*(**name it whatever you wish**) using **ifstream** passing the text file, **sample60.txt** into it. This establishes a new connection between your program and the text file.
3. Fill in the string using the object *hin*. (**Use getline, which we talked about in the last video, to take into input the whole line from the text file.**)
4. Give output to the user, the string we filled in with the content in the text file.
5. Disconnect the file with the program since we are done reading from it using *hin.close()*.

```
#include<iostream>
#include<fstream>

using namespace std;

int main(){

              // connecting our file with hout stream
               ofstream hout("sample60.txt");

               // creating a name string variable and filling it with string entered by
the user
              string name;
               cout<<"Enter your name: ";
              cin>>name;

             // writing a string to the file
             hout<<name + " is my name";

            // disconnecting our file
            hout.close();
            // connecting our file with hin stream
           ifstream hin("sample60.txt");

          // creating a content string variable and filling it with string present
there in the text file
           string content;
            hin>>content;
           cout<<"The content of the file is: "<<content;

           // disconnecting our file
           hin.close();
           return 0;
}
```

Let's run the program we just created. The output will look like this:

```
Enter your name: Harry
The content of the file is: Harry
```

So, when we input a string **"Harry"** into the text file, it gets written there in the file as below, and when we read it from the file, it gives output as below. Since we used **hin** and not **getline**, it could read just the first word.

```
The content of the file is: Harry
```

Thank you, friends for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them. In the next tutorial, we'll be covering the use of **open()** and **eof()** functions, see you there, till then keep coding.

Part: 62

File I/O in C++: open() and eof() functions

In this tutorial, we are going to learn about the member functions open and eof of the objects we learnt about previously.

I remember teaching you all about the two methods to open a text file in our C++ program, first one using a constructor which we discussed in the last tutorial, and the second one, using the member function open, which is to be dealt with today.

Using the member function open:

The member function open is used to connect the text file to the C++ program when passed into it.

Understanding the snippet below:

1. Unlike what we did earlier passing the text file in the object while creating it, we'll first just declare an object out(any name you wish) of the type **`ofstream`** and use its open method to open the text file in the program.
2. We'll pass some string lines to the text file using the out operation.
3. We'll now close the file using the close function. Now closing is explicitly used to make the system know that we are done with the file. It is always good to use this.

This was all about writing to a file. We'll now move to the eof function's vitality in File I/O.

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{

    // declaring an object of the type ofstream
    ofstream out;

    //connecting the object out to the text file using the member function open()
    out.open("sample60.txt");

    //writing to the file
    out <<"This is me\n";
    out <<"This is also me";
    //closing the file connection
    out.close();
    return 0;
}
```

Using the member function eof:

The member function **`eof`**(End-of-file) returns a **`boolean`** true if the file reaches the end of it and false if not.

Understanding the snippet below:

1. We'll first declare an object **`in`**(any name you wish) of the type **`ifstream`** and use its open method similar to what we did above, to open the text file in the program.
2. And now, we'll declare the string variable st to store the content we'll receive from the text file sample60.txt.

3. Now since we not only want the first or some two or three strings present in the text file, but the whole of it, and we have no idea of what the length of the file is, we'll use a while loop.
4. We'll run the while loop until the file reaches the end of it, and that gets checked by using **`eof()`** , which returns 1 or true if the file reaches the end. Till then a 0 or false.
5. We'll use **`getline`** to store the whole line in the string variable **`st`**. Don't forget to include the header file <string>.
6. This program now successfully prints the whole content of the text file.

Refer to the output below the snippet.

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main()
{
    // declaring an object of the type ifstream
    ifstream in;
    //declaring string variable st
    string st;
    //opening the text file into in
    in.open("sample60.txt");

    // giving output the string lines by storing in st until the file reaches the
end of it
    while (in.eof()==0)
    {
        // using getline to fill the whole line in st
        getline(in,st);
        cout<<st<<endl;
    }
    return 0;
}
```

Output of the above program:

This is me

This is also me

So, this was all about File I/O in C++. Learning to detect the eof and opening files in a C++ program in two ways, writing to it and reading from the same, was all a big deal, and you successfully completed them all. Cheers.

Thank you, friends, for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them. In the next tutorial, we'll be starting a topic, a must for competitive programmers, C++ templates, see you there, till then keep coding.

Part: 63

C++ Templates: Must for Competitive Programming

It has been quite a journey till here, and I feel grateful to have you all with me in the same. We have covered a lot in C++ and there is yet a great deal left. But we'll make everything ahead a cakewalk together.

Today we have in the box, the most important topic for all you enthusiastic programmers, C++ templates. We'll follow the below-mentioned roadmap:

1. What is a template in C++ programming?
2. Why templates?
3. Syntax

What is a template in C++ programming?

A template is believed to escalate the potential of C++ several fold by giving it the ability to define data types as parameters making it useful to reduce repetitions of the same declaration of classes for different data types. Declaring classes for every other data **`type(which if counted is way too much)`** in the very first place violates the DRY(Don't Repeat Yourself) rule of programming and on the other doesn't completely utilise the potential of C++.

It is very analogous to when we said classes are the templates for objects, here templates itself are the templates of the classes. That is, what classes are for objects, templates are for classes.

Why templates?

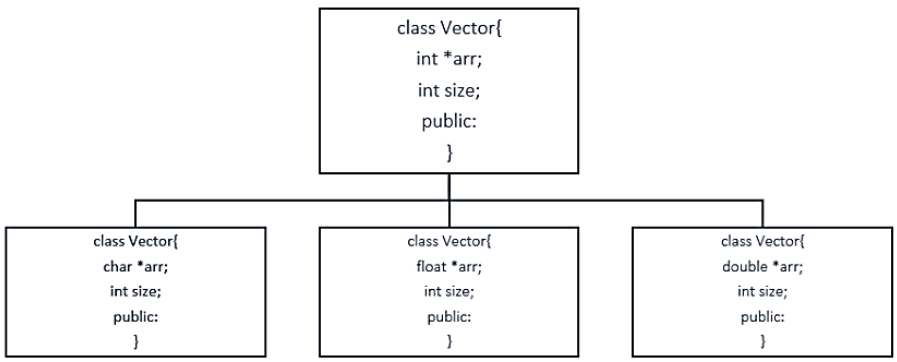
1. **DRY Rule:**

To understand the reason behind using templates, we will have to understand the effort behind declaring classes for different data types. Suppose we want to have a vector for each of the three (can be more) data types, int, float and char. Then we'll obviously write the whole thing again and again making it awfully difficult. This is where the saviour comes, the templates. It helps parametrizing the data type and declaring it once in the source code suffice. Very similar to what we do in functions. It is because of this, also called, '**parameterized classes**'.

1. **Generic Programming:**

It is called generic, because it is sufficient to declare a template once, it becomes general and it works all along for all the data types.

Refer to the schematic below:



We had to copy the same thing again and again for different data types, but a template solves it all. Refer to the syntax section for how.

Below is the template for a vector of int data type, and it goes similarly for float char double, etc.

```
class vector {
    int *arr;
    int size;
    public:
};
```

Syntax:

Understanding the syntax below:

1. First, we declare a template of class and pass a variable T as its parameter.
2. Define the class of vector and keep the data type of **`*arr`** as T only. Now, the array becomes of the type we supply in the template.

Now we can easily use this template to declare umpteen number of classes in our main scope. Be it int, float, or **`arr`** vector.

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

template <class T>
class vector {
    T *arr;
    int size;
    public:
        vector(T* arr)[
            //code
        ]
        //and many other methods

};

int main() {
    vector<int> myVec1();
    vector<float> myVec2();
    return 0;
}
```

Templates are believed to be very useful for people who pursue competitive programming. It makes their work several folds easier. It gives them an edge over others. It is a must because it saves you a lot of time while programming. And I believe you ain't want to miss this opportunity to learn, right?

So, get to the playlist as soon as you can. Save yourselves some time and get over your competitors.

Thank you, friends, for being with me throughout, hope you liked the tutorial. And If you haven't checked out the whole playlist yet, it's never too late, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them. Templates are an inevitable part of this process of learning C++. You just cannot afford to miss this. In the next tutorial, we'll be writing a program using templates for your better understanding, see you there, till then keep coding.

Part: 64

Writing our First C++ Template in VS Code

In the last tutorial, we learnt about what a template is, why a template is used in programming and what its syntax is. Let's give ourselves a quick revision of everything about templates.

Long story short, a template does the same thing to a class, what a class does to the objects. It parametrizes the data type hence making it easy for us to use different classes without having to write the whole thing again and again, violating the DRY rule. Templates furthermore give our program a generic view, where declaring one template suffices the task.

Today, we'll learn to make a program using templates to give you a better understanding about its uses. I'll make the process effortless for you to learn, so, you stay calm and keep learning.

Now suppose we have two integer vectors and we want to calculate their Dot Product. This part should not be troublesome since we have learnt pretty well the use of classes and constructors. We had learnt to write the code like the one mentioned below.

Understanding the code below to calculate the DotProduct of two integer vectors:

1. Here we declare a class vector, with an integer pointer arr.
2. We declared an integer variable to store the size.
3. We made the constructor for the integer vector. These things should be unchallenging for you by now as they have been already taught.
4. We then wrote a function which returns an integer value, to calculate the Dot Product and named it dotProduct which will take a vector as a parameter.
5. We traversed through the vectors multiplying their corresponding elements and adding it to the sum variable named d.
6. We finally returned it to the main.
7. And the output we received is this:

5

```
PS D:\MyData\Business\code playground\C++ course>
#include <iostream>
using namespace std;

class vector
{
    public:
        int *arr;
        int size;
        vector(int m)
        {
            size = m;
            arr = new int[size];
        }
    int dotProduct(vector &v){
        int d=0;
        for (int i = 0; i < size; i++)
        {
            d+=this->arr[i]*v.arr[i];
```

```
        }
        return d;
    }
};

int main()
{
    vector v1(3); //vector 1
    v1.arr[0] = 4;
    v1.arr[1] = 3;
    v1.arr[2] = 1;
    vector v2(3); //vector 2
    v2.arr[0]=1;
    v2.arr[1]=0;
    v2.arr[2]=1;
    int a = v1.dotProduct(v2);
    cout<<a<<endl;
    return 0;
}
```

So, this was all about creating a class and an embedded function to calculate the dot product of two integer vectors. But this program would obviously fail to calculate the dot products for some different data types. It would demand an entirely different class. But we'll save ourselves the effort and the time by declaring a template. Let's see how:

Understanding the changes, we made in the above program to generalise it for all data types:

1. First and foremost, we defined a template with class T where T acts as a variable data type.
2. We then changed the data type of arr to T, changed its constructor to T from int, changed everything except the size of the vector, to a variable T. The function then returned T. This has now changed the class from specific to general.
3. We then very easily added a parameter, while defining the vectors, of its data type. And the compiler itself transformed the class accordingly. Here we passed a float and the code handled it very efficiently.
4. The output we received was:


```
PS D:\MyData\Business\code playground\C++ course>
#include <iostream>
using namespace std;

template <class T>
class vector
{
    public:
        T *arr;
        int size;
        vector(int m)
        {
            size = m;
            arr = new T[size];
        }
    T dotProduct(vector &v){
        T d=0;
        for (int i = 0; i < size; i++)
        {
            d+=this->arr[i]*v.arr[i];
        }
```

```
            return d;
        }
    };

    int main()
    {
        vector<float> v1(3); //vector 1 with a float data type
        v1.arr[0] = 1.4;
        v1.arr[1] = 3.3;
        v1.arr[2] = 0.1;
        vector<float> v2(3); //vector 2 with a float data type
        v2.arr[0]=0.1;
        v2.arr[1]=1.90;
        v2.arr[2]=4.1;
        float a = v1.dotProduct(v2);
        cout<<a<<endl;
        return 0;
    }
```

Imagine how tough it would have been without these templates, you'd have made different classes for different data types handling them clumsily increasing your efforts and proportionally your chances of making errors. So, this is a life saviour.

And learning it will only benefit you. So why not.

Thank you, friends for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them. In the next tutorial, we'll be learning about further uses of a template and multiple parameters, see you there, till then keep coding.

Part: 65

C++ Templates: Templates with Multiple Parameters

In the last tutorial, we had ample understanding of a template and its uses. We had created a template which would calculate the Dot Product of two vectors of any data type just by declaring a simple template parameterizing the data type we usually hardcoded in the classes. This already made our task easier but here we are, with our next tutorial focusing on how to handle multiple parameters in a template.

To give you a short overview of how templates work with multiple parameters, you can think of it as a function where you have that power to pass different parameters of the same or different data types. A simple template with two parameters would look something like this. The only effort it demands is the declaration of parameters. We'll get through it thoroughly by making a real program, so, let's go.

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

/*
template<class T1, class T2>
class nameOfClass{
    //body
}
*/

int main(){
    //body of main
}
```

Code Snippet 1: Syntax of a template with multiple parameter

Suppose we have a class named myClass which has two data in it of data types int and char respectively, and the function embedded just displays the two. Fair enough, no big deal, we'll construct our class something like this. The problem arises when we wish to have both our data types anonymous and to be put from the main itself. You will be surprised to know that very subtle modifications in yesterday's code would do our task. Instead of declaring a single parameter T, we would declare two of them namely T1 And T2.

```
class myClass{
    public:
        int data1;
        char data2;
    void display(){
        cout<<this->data1<<" "<<this->data2;
    }
};
```

Code Snippet 2: Constructing a class

Refer to changes we have done below to parametrize both our data types using a single template:

1. We have declared data1 and data2 with data types T1 and T2 respectively.
2. We have applied the constructor filling the values we receive from the main into data1 and data2.
3. Finally, we have displayed both of them.

```
template<class T1, class T2>
class myClass{
    public:
```

```
        T1 data1;
        T2 data2;
        myClass(T1 a,T2 b){
            data1 = a;
            data2 = b;
        }
    void display(){
        cout<<this->data1<<" "<<this->data2;
    }
};
```

Code Snippet 2: Constructing a template with two parameters.

Let me now show you how this template works for different parameters. I'll pass different data types from the main and see if it's flexible enough.

Firstly, we put an integer and a char,

```
int main()
{
    myClass<int, char> obj(1, 'c');
    obj.display();
}
```

Code Snippet 3: Specifying the data types to be int and char.

And the output received was this, which is correct. Let's feed another one.

```
1   c
PS D:\MyData\Business\code playground\C++ course>
```

Figure 1: Output of code snippet 3.

Now we put an integer and a float,

```
int main()
{
    myClass<int, float> obj(1,1.8 );
    obj.display();
}
```

Code Snippet 4: Specifying the data types to be int and float.

And the output received was this,

```
1   1.8
PS D:\MyData\Business\code playground\C++ course>
```

Figure 1: Output of code snippet 4.

So yes, this is functioning all good.

And this was all about templates with multiple parameters, just don't miss out the commas while defining the parameters in a template. And you can have 2, 3 or more of them according to your needs. Could you believe how luxurious it has become to work with customized data types? It is now you, who'll decide what the data type of some variable in a class should be. It is no longer pre-specified. It has given you some unimaginable power which, if you realise, can save you a lot of energy and time.

Thank you, friends for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them. In the next tutorial, we'll be learning about having a default parameter in a template, see you there, till then keep coding.

Part: 66

C++ Templates: Class Templates with Default Parameters

So far, we have already covered the C++ templates with single parameters. In the last tutorial, we learnt about templates with multiple parameters, when it comes to handling different data types of two or more containers. Today, we'll be learning a very easy yet powerful attribute of templates, its ability to have default parameters. Its ability to have default specifications about the data type, when it receives no arguments from the main.

So, let's start by making a program manifesting the use of default parameters in a C++ template. **Refer to the code snippet below and follow the steps:**

1. We'll start by constructing a class named Harry.
2. We'll then define a template with any number of arguments, let three, T1, T2, and T3. If you remember, we had this feature of specifying default arguments for functions, similarly we'll mention the default parameters, let, int, float and char for T1, T2 and T3 respectively.
3. This ensures that if the user doesn't put any data type in main, default ones get considered.
4. In public, we'll define variables a, b and c of the variable data types T1, T2 and T3. And build their constructors.
5. The constructor accepts the values featured by the main, and assigns them to our class variables a, b and c. If the user specifies the data types along with the values, the compiler assigns them to T1 , T2 and T3, otherwise gives them the default ones, as specified while declaring the template itself.
6. We'll then create a void function display, just to print the values the user inputs.

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

template <class T1=int, class T2=float, class T3=char>
class Harry{
    public:
        T1 a;
        T2 b;
        T3 c;
        Harry(T1 x, T2 y, T3 z) {
            a = x;
            b = y;
            c = z;
        }
        void display(){
            cout<<"The value of a is "<<a<<endl;
            cout<<"The value of b is "<<b<<endl;
            cout<<"The value of c is "<<c<<endl;
        }
};
```

Since we are done defining the templates and class, we can very easily move to the main where we'll see how these work. **Understanding code snippet 2:**

1. Firstly, we'll create an object, let's name it h, of the class Harry. And we'll pass into it three values, an int, a float and a char, suppose 4, 6.4 and c respectively. Now since we have not specified the data types of the values we have just entered, the default data types, int, float and char would be considered.
2. We'll then display the values, which you'll be seeing when we run the same.

3. And then we'll create another object g, of the class Harry but this time, with the data types of our choice. Let's specify them to be float, char and char.
4. We can then pass some values into it, suppose 1.6, o, and c and call the display function again.
5. These objects are sufficient to give us the main concept behind using a default parameter and the variety of classes we could make via this one template.

```
int main()
{
    Harry<> h(4, 6.4, 'c');
    h.display();
    cout << endl;
    Harry<float, char, char> g(1.6, 'o', 'c');
    g.display();
    return 0;
}
```

We'll now refer to the output the above codes combinedly gave. As you can see below, it worked all fine. Had we not specified the default parameters; the above program would have thrown an error. Thanks to this feature of C++ templates.

```
The value of a is 4
The value of b is 6.4
The value of c is c

The value of a is 1.6
The value of b is o
The value of c is c
PS D:\MyData\Business\code playground\C++ course>
```

Thank you, friends for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them all. So far, we were learning about the class templates. See you all in the next tutorial where we'll look after the **function templates**. Till then keep coding.

Part: 67

C++ Function Templates & Function Templates with Parameters

In this tutorial, we are wishing to learn how a function template works. Prior to this video, we have only talked about a class template and its functionalities. In class template we used to have template parameters which we, very often, addressed as a variable for our data types. We have also declared a class template similar to what shown here below:

```
template <class T1 = int, class T2 = float>
```

Today, we'll be interested in knowing what a function template does. So. let's get ourselves on our editors. Suppose we want to have a function which calculates the average of two integers. So, this must be very easy for you to formulate. Look for the snippet below.

1. We have declared a float function named **`funcAverage`** which will have two integers as its parameters, a and b.
2. We stored it average in a float variable **`avg`** and returned the same to the main.
3. Later we called this function by value, and stored the returned float in a float variable a and printed the same.
4. So, this was the small effort we had to make to get a function which calculates the average of two integers.

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

float funcAverage(int a, int b){
    float avg= (a+b)/2.0;
    return avg;
}
int main(){
    float a;
    a = funcAverage(5,2);
    printf("The average of these numbers is %f",a);
    return 0;
}
```

The output of the above program is :

```
The average of these numbers is 3.500000
PS D:\MyData\Business\code playground\C++ course>
```

But the effort we made here defining a single function for two integers increases several folds when we demand for a similar function for two floats, or one float and one integer or many more data type combinations. We just cannot repeat the procedure and violate our DRY rule. We'll use function templates very similar to what we did when we had to avoid defining more classes.

See what are the subtle changes we had to make, to make this function generic.

We'll first declare a template with two data type parameters T1 and T2. And replace the data types we mentioned in the function with them. And that's it. Our function has become general for all sorts of data types. Refer to the snippet below.

```
template<class T1, class T2>
float funcAverage(T1 a, T2 b){
    float avg= (a+b)/2.0;
    return avg;
}
```

Let's call this function by passing into it two sorts of data types combination, first, two integers and then one integer and one float. And see if the outputs are correct.

```
int main(){
    float a;
    a = funcAverage(5,2);
    printf("The average of these numbers is %f",a);
    return 0;
}
```

Code snippet: Calling the function by passing two integers

```
The average of these numbers is 3.500000
PS D:\MyData\Business\code playground\C++ course>
```

```
int main(){
    float a;
    a = funcAverage(5,2.8);
    printf("The average of these numbers is %f",a);
    return 0;
}
```

Code snippet: Calling the function by passing one integer and one float

```
The average of these numbers is 3.900000
PS D:\MyData\Business\code playground\C++ course>
```

And a general swap function named swapp for those variety of data types we have, would look something like the one below:

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

So, this is how we utilize this powerful tool to avoid writing such overloaded codes. And this was all about function templates with single or multiple parameters. We covered them all in this tutorial.

Thank you, for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them all. We are now done with both class and function templates. See you all in the next tutorial where we'll see if a function template can be overloaded. Till then keep coding.

Part: 68

Member Function Templates & Overloading Template Functions in C++

So, since we have finished learning about the two template categories, we can now swiftly dive deep into if it's possible for a template function to get overloaded, and if yes, then how.

Before starting to know what an overloaded template function is, we'll learn how to declare a template function outside a using the scope resolution operator, **'::'**.

First, we'll revise how to write a function inside the class by just following the snippet given below.

1. We'll declare a template, then a class named Harry.
2. We'll then define a variable *data* inside that class with variable data type T.
3. We then make a constructor feeding the value received from the main to data.
4. And then, we'll write the function, *display* and write its code.

This was an unchallenging task. But when we need the function to be declared outside the class, we follow the code snippet 2.

```
template <class T>
class Harry
{
public:
    T data;
    Harry(T a)
    {
        data = a;
    }
    void display()
    {
        cout << data;
    }
};
```

Code Snippet 1: Writing function inside the class

Here, we first write the function declaration in the class itself. Then move to the outside and use the scope resolution operator before the function and after the name of the class Harry along with the data type T. We must specify the function data type, which is void here. And it must be preceded by the template declaration for class T.

And write the display code inside the function and this will behave as expected. See the output below the snippet.

```
template <class T>
class Harry
{
public:
    T data;
    Harry(T a)
    {
        data = a;
    }
    void display();
};

template <class T>
void Harry<T> :: display(){
    cout<<data;
}
```

Code Snippet 2: Writing function outside the class

So, to check if it's working all fine, we'll call this function from the main.

```
int main()
{
    Harry<int> h(5.7);
    cout << h.data << endl;
    h.display();
    return 0;
}
```

Code Snippet 3: Calling the function from the main

And the output is:

```
5
5
PS D:\MyData\Business\code playground\C++ course>
```

Now, we'll move to the **overloading of a function template**. Overloading a function simply means assigning two or more functions with the same name, the same job, but with different parameters. For that, we'll declare a void function named func. And a template function with the same name. Follow the snippet below to do the same:

1. We made two void functions, one specified and one generic using a template.
2. The first one receives an integer and prints the integer with a different prefix.
3. The generic one receives the value as well as the data type and prints the value with a different prefix.
4. Now, we'll wish to see the output of the following functions, by calling them from the main. Refer to the main program below the snippet below.

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

void func(int a){
    cout<<"I am first func() "<<a<<endl;
}

template<class T>
void func(T a){
    cout<<"I am templatised func() "<<a<<endl;
}
```

Code Snippet 4: Overloading the template function

And now when we call the function func, we'll be interested to know which one among the two it calls. So here since we've entered a value with an integer parameter, it finds its exact match in the overloading and calls that itself. That is, it gives its exact match the highest priority. Refer to the output below the snippet:

```
int main()
{
    func(4);  //Exact match takes the highest priority
    return 0;
}
```

Code Snippet 5: Calling function func from the main

And the output is,

```
I am first func() 4
PS D:\MyData\Business\code playground\C++ course>
```

If we hadn't created the first function with int data type, the call would have gone to the templatised func only because a template function is an exact match for every kind of data type. So this was enough preparation for the next topic, STL(Standard Template Library). It might have sounded boring to you for quite a few days, but the results would fascinate you once we enter STL, which is a must for all the competitive programmers out there. Thank you, for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them all. See you all in the next tutorial where we'll start the STL. Till then keep coding.

Part: 69

The C++ Standard Template Library (STL)

We have been waiting so long to start this, but creating a base is as important as any other phase. So, today we'll be starting the most awaited topic, the STL(Standard Template Library).

There is a reason why I've been saying that this topic is a must for all the competitive programmers out there, so let's deal with that first.

Why is this important for competitive programmers?

1. Competitive programming is a part of various environments, be it job interviews, coding contests and all, and if you're in one of those environments, you'll be given limited time to code your program.
2. So, suppose you want in your program, a resizable array, or sort an array or any other data structure. or search for some element in your container.
3. You will always try to code a function which will execute the above mentioned things, and end up losing a great amount of time. But here is when you will use STL.

An STL is a library of generic functions and classes which saves you time and energy which you would have spent constructing for your use. This helps you reuse these well tested classes and functions umpteenth number of times according to your own convenience.

To put this simply, STL is used because it is not a good idea to reinvent something which is already built and can be used to innovate things further. Suppose you go to a company who builds cars, they will not ask you to start from scratch, but to start from where it is left. This is the basic idea behind using STL.

COMPONENTS OF STL:

We have three components in STL:

1. Containers
2. Algorithm
3. Iterators

Let's deal with them individually;

Containers:

Container is an object which stores data. We have different containers having their own benefits. These are the implemented template classes for our use, which can be used just by including this library. You can even customise these template classes.

Algorithms:

Algorithms are a set of instructions which manipulates the input data to arrive at some desired result. In STL, we have already written algorithms, for example, to sort some data structure, or search some element in an array. These algorithms use template functions.

Iterators:

Iterators are objects which refer to an element in a container. And we handle them very much similarly to a pointer. Their basic job is to connect algorithms to the container and play a vital role in manipulation of the data.

I'll give you a quick illustration of how they work combinedly.

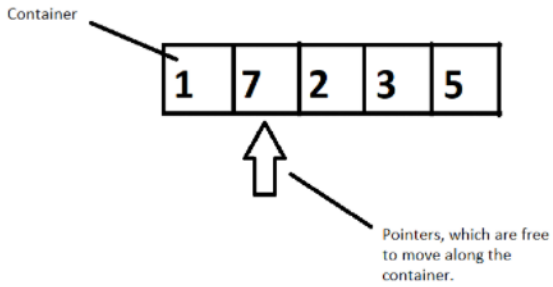


Figure 1: Illustration of how these three components work together

Suppose we have a container of integers, and we want to sort them in ascending order. We will have pointers which will help moving elements to places by pointing to it, following a well-constructed algorithm. So, here a container gets sorted by following an algorithm by the use of pointers. This is how they work in accordance with each other.

So, this was the basics of STL and the motivation behind using it in your programs. I hope I was able to introduce it to you.
Thank you, for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them all. See you all in the next tutorial where we'll dive deep in the containers and its different types. Till then keep coding.

Part: 70

Containers in C++ STL

In the last tutorial, we had briefed about the three components of STL, namely,

Containers, objects which store data, **Algorithms,** set of procedures to process data, and **Iterators,** objects which point to some element in a container. Today, in this tutorial, we will be interested in discussing more about containers.

Containers are themselves of three types:

1. Sequence Containers
2. Associative Containers
3. Derived Containers

When we talked about containers, we said containers are objects which store data, but what are its three types all about? We'll discuss that too.

- **Sequence Containers**

A **sequence container** stores that data in a linear fashion. Refer to the illustration below to understand what storing something in a linear fashion means.

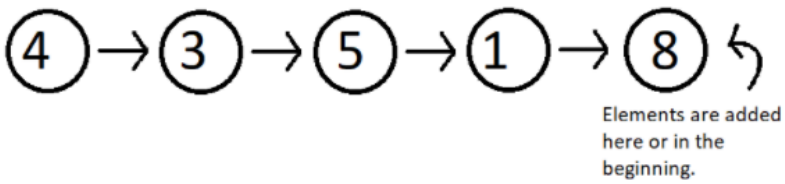


Figure 1: Elements stored in a linear fashion

Sequence containers include **Vector, List, Dequeue etc.** These are some of the most used sequence containers.

- **Associative Containers**

An **associative container** is designed in such a way that enhances the accessing of some element in that container. It is very much used when the user wants to fastly reach some element. Some of these containers are, **Set, Multiset, Map, Multimap etc.** They store their data in a tree-like structure.

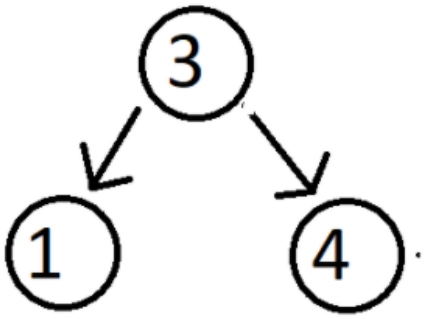


Figure 2: A tree-like structure

- **Derived Containers**

As the name suggests, these containers are derived from either the sequence or the associative containers. They often provide you with some better methods to deal with your data. They deal with real life modelling.

Some examples of derived containers are **Stack, Queue, Priority Queue, etc.** The following illustration give you the idea of how a stack works.

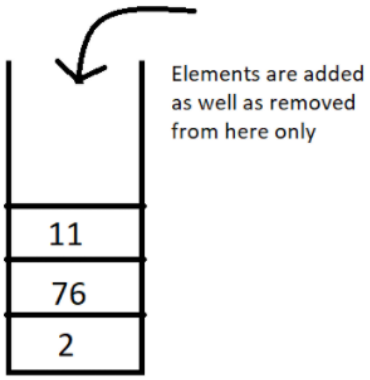


Figure 3: A stack, works on the first in first out [FIFO] method

Now since we have got the basic idea of all the three types of containers, a question which might arise is **when to use which.** So, let's deal with that,

In sequence containers, we have **Vectors**, which has following properties:

1. Faster random access to elements in comparison to array
2. Slower insertion and deletion at some random position, except at the end.
3. Faster insertion at the end.

In **Lists,** we have,

1. Random accessing elements is too slow, because every element is traversed using pointers.
2. Insertion and deletion at any position is relatively faster, because they only use pointers, which can easily be manipulated.

In associative containers, every operation except random access is faster in comparison to any other containers, be it inserting or deleting any element.

In associative containers, we cannot specifically tell which operation is faster or slower, we'll have to inspect every data structure separately, and to get a clearer picture of all of these, you can access my Data Structure course : Data Structures and Algorithms Course in Hindi

For now, I'd like to hold on to our topic STL, and get you a strong hold on this too. In the coming videos, we'll deal with our vectors, list, dequeues, set, multiset, maps, stack and much more. Just bear with me.

Thank you, for being with me throughout, hope you liked the tutorial. If you haven't checked out the whole playlist yet, move on to codewithharry.com or my YouTube channel to access it. I hope you enjoy them all.

See you all in the next tutorial where we'll talk about Vectors in C++ STL in detail. Till then keep coding.