ECE 220 Computer Systems & Programming

Lecture 22 – Polymorphism, Template and STL

ECE ILLINOIS

Adapted from Prof. Moon's slides

ILLINOIS

Polymorphism

- A call to a member function will cause a different function to be executed depending on the type of the object that invokes the function.
- Function overriding allows to have the same function in derived class which is already defined in its base class.

```
class Vehicle{
   public:
   void ShowData(){cout<<"<<Vehicle>> "<<endl;}
};
class Airplane : public Vehicle{
   public:
   void ShowData(){cout<<"<<Airplane>> "<<endl;}
};
class Train : public Vehicle{
   public:
   void ShowData(){cout<<"<<Train>> "<<endl;}
};
```

```
int main(){
   Airplane a(100,300,20);
   a.ShowData();
}
```

<<Airplane>>

Airplane::ShowData() overrides Vehicle::ShowData().

Declared Type vs. Actual Type

```
int main(){
   Airplane a(100,300,20);
   Train t(50,100,30);

   a.ShowData();                      <<Airplane>>
   t.ShowData();                      <<Train>>

   Vehicle *ptr;
   ptr = &a;
   ptr->ShowData();                   <<Vehicle>>

   ptr = &t;
   ptr->ShowData();                   <<Vehicle>>

}
```

- Base class pointer (or reference) can point its derived class.

- However, the base class does not have access to its derived class members.

Virtual Function

- Virtual functions are the member function in the base class that is expected to **be redefined in the derived class**.

```
class Vehicle{
   public:
   virtual void ShowData(){
      cout<<"<<Vehicle>> "<<endl;
   }
};
class Airplane : public Vehicle{
   public:
   void ShowData(){
      cout<<"<<Airplane>> "<<endl;
   }
};
class Train : public Vehicle{
   public:
   void ShowData(){
      cout<<"<<Train>> "<<endl;
   }
};
```

```
int main(){
   Airplane a(100,300,20);
   Train t(50,100,30);

   a.ShowData();
   t.ShowData();

   Vehicle *ptr;
   ptr = &a;
   ptr->ShowData();

   ptr = &t;
   ptr->ShowData();
}
```

<<Airplane>>

<<Train>>

static binding

<<Airplane>>

dynamic binding

<<Train>>

Virtual Function – Why?

```
class City{
   private:
      Vehicle *vlist[100];
      int index;
   public:
      City(){ index = 0;}
      void AddVehicle(Vehicle *v){
         vlist[index++] = v;
      }
      void ShowList(){
         for(int i=0;i<index;i++)
            vlist[i]->ShowData();
      }
};
```

```
int main(){
   City Champaign;

   Champaign.AddVehicle(new Airplane(30,100,5));
   Champaign.AddVehicle(new Train(100,300,10));
   Champaign.AddVehicle(new Train(130,300,15));

   Champaign.ShowList();
}
```

Virtual Function – Why?

```
class City{
  private:
    Vehicle *vlist[100];
    int index;
  public:
    City(){ index = 0;}
    void AddVehicle(Vehicle *v){
      vlist[index++] = v;
    }
    void ShowList(){
      for(int i=0;i<index;i++)
        vlist[i]->ShowData();
    }
};
```

```
int main(){
  City Champaign;

  Champaign.AddVehicle(new Airplane(30,100,5));
  Champaign.AddVehicle(new Train(100,300,10));
  Champaign.AddVehicle(new Train(130,300,15));

  Champaign.ShowList();
}
```

We want to print out the full information about Airplane or Train.

But, it will only print out Vehicle.

We want to manage *base class*, not *derived classes*.

→ Wish to resolve functions at run-time, a.k.a. dynamic binding.

Abstraction – Pure Virtual Function & Abstract Class

- ‘Vehicle’ class will never be instantiated as it is. Instead, it will be either ‘Airplane’ or ‘Train’ object.
- Abstract class cannot be instantiated (pointer is fine) and implemented with one or more “pure” virtual function

```
class Vehicle{            <=  abstract class (has a pure virtual function)
   public:
   virtual void ShowData() = 0;            <=  pure virtual function (has no body)
};
class Airplane : public Vehicle{
   public:
   void ShowData(){
      cout<<"<<Airplane>> "<<endl;
   }
};
class Train : public Vehicle{
   public:
   void ShowData(){
      cout<<"<<Train>> "<<endl;
   }
};
```

```
int main(){
   Vehicle *vptr;             // this is ok
   Vehicle v(100,10);         // compile error
}
```

*Derived class must define a body for the pure virtual function, otherwise it will also be considered an abstract base class.

Constructor & Destructor

```
class Person{
    char name[20];
    int age;
public:
    Person(char const *_name, int _age){
        strcpy(name, _name);
        age = _age;
        cout<<"constructing name: "<<name<<endl;
    };
    ~Person(){
        cout<<"destroying name: "<<name<<endl;
    };
};

int main(){
    Person p1 = Person("Alice", 20);
    Person p2 = Person("Bob", 20);
}
```

```
constructing name: Alice
constructing name: Bob
destroying name: Bob
destroying name: Alice
```

Constructor & Destructor

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

class Person{
char *name;
 int age;

public:

 Person(char const *_name, int _age);
    Person(){};
 void ShowData(){cout<<name<<" "<<age<<endl;}
~Person();
};
Person::Person(char const *_name, int _age){
name = new char[strlen(_name)+1];
strcpy(name, _name);
age = _age;
}

Person::~Person(){
cout<<"destructor is working"<<" "<<name<<endl;
delete []name;
}

int main(){
// Person p = {"Alice", 20};
 Person p("Alice", 20);
 p.ShowData();
 Person P2("Bob", 30);
 P2.ShowData();
}
```

Alice 20
Bob 30
destructor is working Bob
destructor is working Alice

Copy Constructor

- Initialize an object using another object (member-by-member).
- If a copy constructor is not provided by the user,
  it will be automatically inserted (default copy constructor)

```
class Point{
private:
    int x,y;
public:
    Point(int _x, int _y){x = _x; y = _y;}
    Point(const Point &p){
        x = p.x;
        y = p.y;
        //p.x = 0; // Don’t want to allow this
    }
    void ShowData(){ cout<<"("<<x<<", "<<y<<")"<<endl;}
};
int main(){
    Point p1(10,20);
    Point p2(p1);

    p1.ShowData();
    p2.ShowData();
}
```

Use “const” to prevent modification on p

Shallow Copy

```
class Person{
    private:
    char *name;
    int age;
    public:
    Person(){};
    Person(const char *_name, int _age);
    void ShowData();
    ~Person();
};
Person::Person(const char *_name, int _age){
    name = new char[strlen(_name)+1];
    strcpy(name, _name);
    age = _age;
}
Person::~Person(){
    delete []name;
}
```

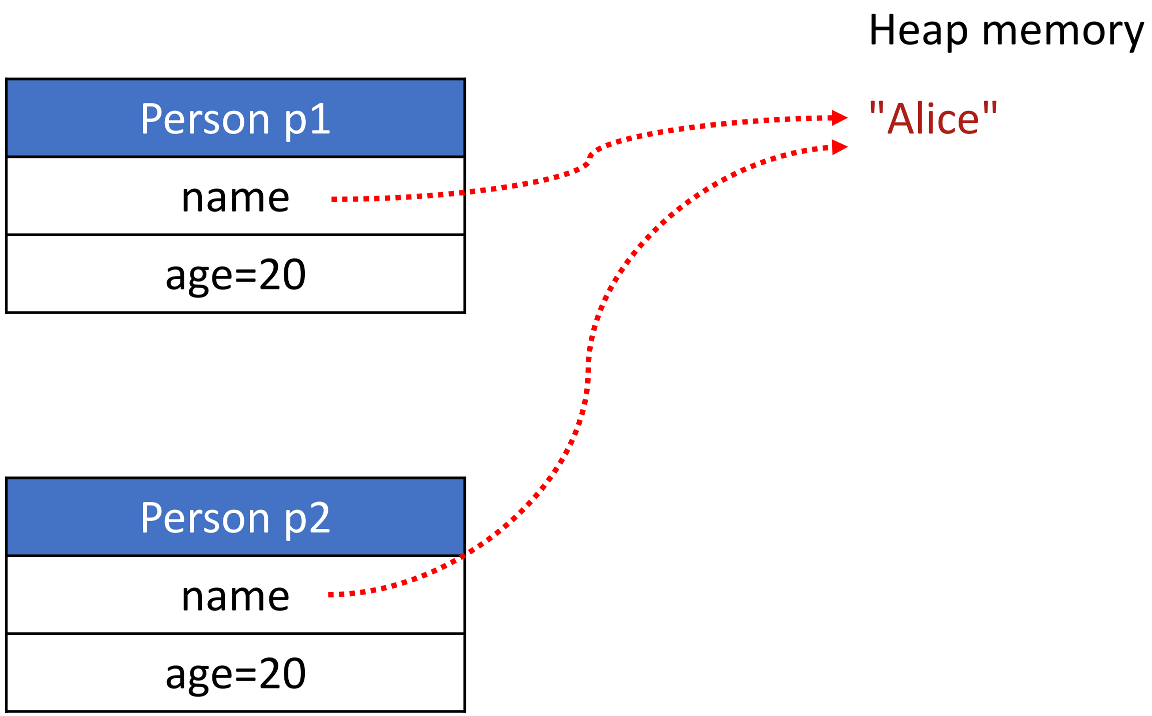
```
int main(){
    Person p1 = Person("Alice", 20);
    Person p2(p1);
    p1.ShowData();
    p2.ShowData();
}
```

Default copy constructor will be inserted.

```
Person::Person(const Person &p){
    name = p.name;
    age = p.age;
}
```

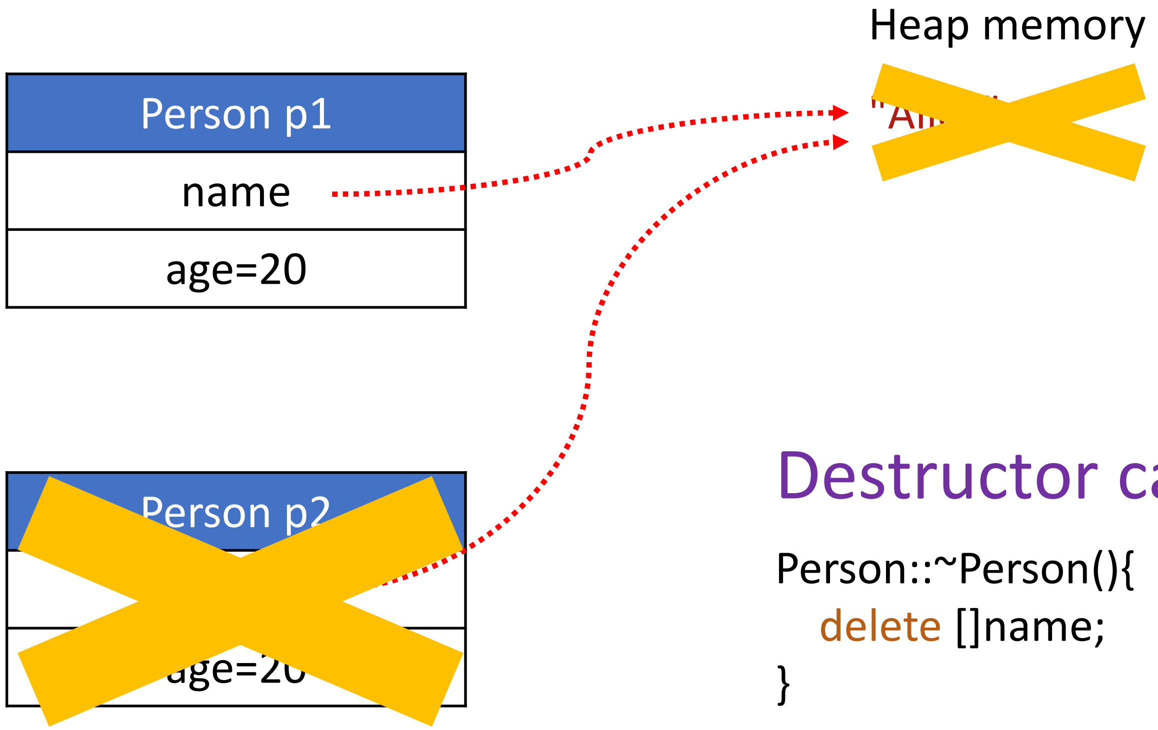
Run-time error!

Shallow Copy




```
Person::Person(const Person &p){
   name = p.name;
   age = p.age;
}
```

Shallow Copy



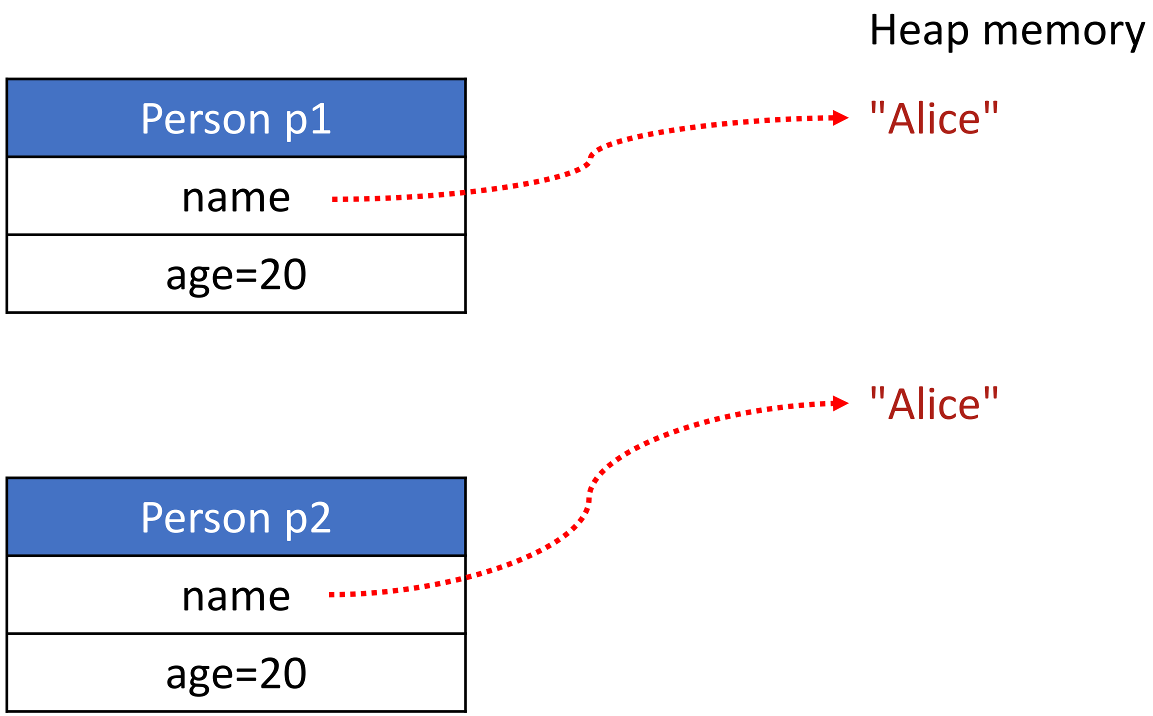
Destructor called!

```
Person::~Person(){
   delete []name;
}
```

When p1 calls its destructor,
the heap memory pointed by “name” is already deallocated.

🚫 double free!

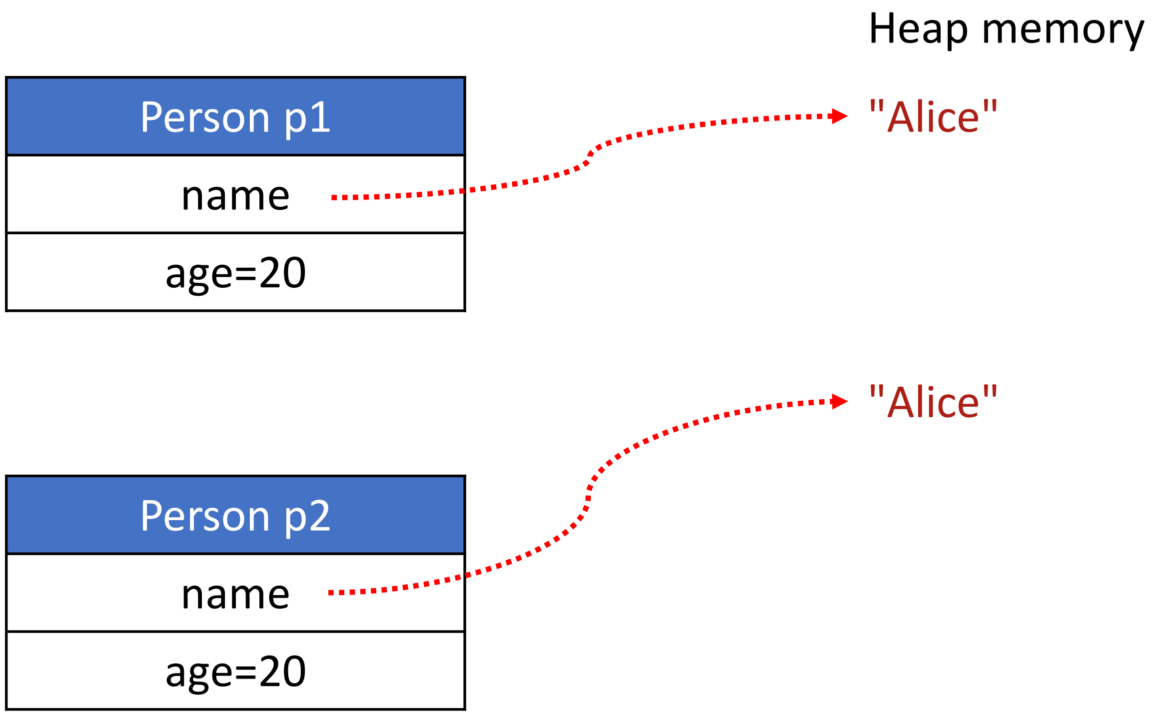
Deep Copy




```cpp
Person::Person(const Person &p){
    name = p.name;
    age = p.age;
}
```

```cpp
Person::Person(const Person &p){
    name = new char[strlen(p.name)+1];
    strcpy(name, p.name);
    age = p.age;
}
```

Deep Copy




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

class Person{
char *name;
 int age;

public:

 Person(char const *_name, int _age);
    Person(){};
    Person(const Person &p);
 void ShowData(){cout<<name<<" "<<age<<endl;}

~Person();
};
Person::Person(char const *_name, int _age){
name = new char[strlen(_name)+1];
strcpy(name, _name);
age = _age;
}

 Person::Person(const Person &p){
 name = new char[strlen(p.name)+1];
strcpy(name, p.name);
age = p.age;
}
```

```cpp
Person::~Person(){
cout<<"destructor is working"<<" "<<name<<endl;
delete []name;
}

int main(){
// Person p = {"Alice", 20};
 Person p("Alice", 20);
 p.ShowData();
 Person P2("tom", 30);
 P2.ShowData();
 Person P3(P2);
 P3.ShowData();
}
```

Alice 20
tom 30
tom 30
destructor is working tom
destructor is working tom
destructor is working Alice

this Pointer

- The this pointer holds the address of the current object.

```
class AAA{
  public:
    AAA *getAddress(){
      return this;
    }
};

int main(){
  AAA *a1 = new AAA();
  cout<<"pointer a1: "<<a1<<endl;
  cout<<"this of a1: "<<a1->getAddress()<<endl;
```

```
pointer a1: 0xddb010
this of a1: 0xddb010
```

Copy Constructor vs Copy Assignment Operator in C++

Copy constructor and Copy Assignment operator are similar as they are both used to initialize one object using another object. However,

Copy constructor is called when a new object is created from an existing object, as a copy of the existing object

Copy assignment operator is called when an already initialized object is assigned a new value from another existing object.

Copy Constructor Vs
Copy Assignment Operator

```
[ubhowmik@linux-ssh-08 CPlus]$ g++ copy_assn_constructor.cpp
[ubhowmik@linux-ssh-08 CPlus]$ ./a.out
Assignment operator called

Copy constructor called
```

Ref:
https://www.geeksforgeeks.org/copy-constructor-vs-assignment-operator-in-c/

```cpp
// CPP Program to demonstrate the use of copy constructor
// and assignment operator
#include <iostream>
#include <stdio.h>
using namespace std;

class Test {
public:
        Test() {}

        Test(const Test& t)
        {
                cout << "Copy constructor called " << endl;
        }

        Test& operator=(const Test& t)
        {
                cout << "Assignment operator called " << endl;
                return *this;
        }

};

// Driver code
int main()
{
        Test t1, t2;
        t2 = t1;
        getchar();
        Test t3 = t1;
        getchar();
        return 0;
}
```

Default things added by compiler, if user doesn’t provide

- constructor
- destructor
- copy constructor
- copy assignment

Templates

- A template is a blueprint for creating a generic function or a class.

- Template ruler:
  Shapes are pre-defined, but colors are not.

- Template in cpp:
  Functionalities are pre-defined, but types are not.

1. Function templates 2. Class templates

Function Templates

```
int Add(int a, int b){
    return a+b;
}
double Add(double a, double b){
    return a+b;
}
```

template parameter

```
template <typename T>
T Add(T a, T b){
    return a+b;
}
```

```
int main(){
    cout<<Add(1, 3)<<endl;
    cout<<Add(1.2, 2.5)<<endl;
}
```

Function Templates

- Multiple template parameters are possible.

```
template <typename T1, typename T2>
void Print(T1 a, T2 b){
    cout<<a<<endl;
    cout<<b<<endl;
}

int main(){
    Print(2,'b');
}
```

Example1:

```
template <typename T>
T Mul(T a, T b){
   return a*b;
}

class Point{
   private:
      int x,y;
   public:
      Point(int _x=0, int _y=0){x=_x; y=_y;}
      void ShowPosition(){cout<<x<<", "<<y<<endl;}
};


int main(){
   Point p1(1,2);
   Point p2(3,1);
   Point p3;
   p3 = Mul(p1,p2);
```

What should we add in Point class?

Example1:

```
template <typename T>
T Mul(T a, T b){
    return a*b;
}

class Point{
    private:
        int x,y;
    public:
        Point(int _x=0, int _y=0){x=_x; y=_y;}
        void ShowPosition(){cout<<x<<", "<<y<<endl;}
};


int main(){
    Point p1(1,2);
    Point p2(3,1);
    Point p3;
    p3 = Mul(p1,p2);
```

What should we add in Point class?

```
Point operator*(const Point& p){
    Point temp(x*p.x, y*p.y);
    return temp;
}
```

Class Templates

- Just like function templates, class templates allow classes to have members that use template parameters as types.

```
class Data{
   int data;
public:
   Data(int d){
      data = d;
   }
   void ShowData(){
      cout<<data<<endl;
   }
};
int main(){
   Data d1(10);
   d1.ShowData();
```

```
template <class T>
class Data{
   T data;
public:
   Data(T d){
      data = d;
   }
   void ShowData(){
      cout<<data<<endl;
   }
};
int main(){
   Data<int> d1(10);
   d1.ShowData();

   Data<char> d2('a');
   d2.ShowData();
```

Require an explicit template parameter for class templates.

Example: Implement Stack using Class Template

```
class MyStack{
  private:
    int TOS; // index for top of the stack
    int size; // size of the stack
    char* data; // pointer for dynamic array
  public:
    MyStack();
    ~ MyStack();
    void push(const char &value);
    char top(); // return the top
    void pop(); // remove the top
    bool empty();
};
```

```
int main(){
  MyStack s(10);
  s.push('A');
  s.push('B');
  s.push('C');

  while(!s.empty()){
    cout<<s.top()<<endl;
    s.pop();
  }
```

Design Stack class for character type.

Example: Implement Stack using Class Template

```
class MyStack{
  private:
    int TOS; // index for top of the stack
    char* data; // pointer for dynamic array
  public:
    MyStack(int size = 5){
      TOS = -1; // initialize TOS
      data = ___________________________;
    }
    ~ MyStack(){ ________________________;}
    void push(const char &value){
      TOS++;
      data[TOS] = value;
    }
    char top(){ return data[TOS];}
    void pop(){ TOS--;}
    bool empty(){
      if(____________) return true;
      else return false;
    }
};
```

```
int main(){
  MyStack s(10);
  s.push('A');
  s.push('B');
  s.push('C');

  while(!s.empty()){
    cout<<s.top()<<endl;
    s.pop();
  }
```

Design Stack class for character type.

Practice: Implement Stack using Class Template

```
class MyStack{
  private:
    int TOS; // index for top of the stack
    char* data; // pointer for dynamic array
  public:
    MyStack(int size = 5){
      TOS = -1; // initialize TOS
      data = new char[size];
    }
    ~ MyStack(){ delete []data;}
    void push(const char &value){
      TOS++;
      data[TOS] = value;
    }
    char top(){ return data[TOS];}
    void pop(){ TOS--;}
    bool empty(){
      if(TOS==-1) return true;
      else return false;
    }
};
```

```
int main(){
  MyStack s(10);
  s.push('A');
  s.push('B');
  s.push('C’);

  while(!s.empty()){
    cout<<s.top()<<endl;
    s.pop();
  }
```

Modify Stack class to adapt any type.

Example: Implement Stack using Class Template

Modify Stack class to adapt any type.

```
int main(){
    MyStack<char> s1(10);
    s1.push('A');
    s1.push('B');
    s1.push('C');
    while(!s1.empty()){
        cout<<s1.top()<<endl;
        s1.pop();
    }

    MyStack<int> s2(10);
    s2.push(1);
    s2.push(2);
    s2.push(3);
    while(!s2.empty()){
        cout<<s2.top()<<endl;
        s2.pop();
    }
```

```
template <class T>
class MyStack{
    private:
        int TOS;
        T* data;
    public:
        MyStack(int size = 5){
            TOS = -1;
            data = new T[size];
        }
        ~ MyStack(){ delete []data;}
        void push(const T &value){
            TOS++;
            data[TOS] = value;
        }
        T top(){ return data[TOS];}
        void pop(){ TOS--;}
        bool empty(){
            if(TOS==-1) return true;
            else return false;
        }
};
```

```
int main(){
    MyStack<char> s1(10);
    s1.push('A');
    s1.push('B');
    s1.push('C');
    while(!s1.empty()){
        cout<<s1.top()<<endl;
        s1.pop();
    }

    MyStack<int> s2(10);
    s2.push(1);
    s2.push(2);
    s2.push(3);
    while(!s2.empty()){
        cout<<s2.top()<<endl;
        s2.pop();
    }
```

Standard Template Library (STL)

- STL is a set of C++ template classes to provide common data structures and functions.
  - Algorithms
  - Functors
  - **Containers – stores objects and data**
    - **vector**
    - **list**
    - **stack**
    - **queue**
  - **Iterators – object that points to an element inside the container**

Container example - stack

```cpp
#include <iostream>
#include <stack>

int main(){
    stack<char> s;
    s.push('a');
    s.push('b');
    s.push('c');

    while(!s.empty()){
        cout<<s.top()<<endl;
        s.pop();
    }
}
```

fx Member functions

(constructor)	Construct stack (public member function)
empty	Test whether container is empty (public member function)
size	Return size (public member function)
top	Access next element (public member function)
push	Insert element (public member function)
emplace C++11	Construct and insert element (public member function)
pop	Remove top element (public member function)
swap C++11	Swap contents (public member function)

Container example : Vector (dynamic array)

vector::size	Return size (public member function)
vector::push_back	Add element at the end (public member function)

```
#include <vector>
using namespace std; //for vector and cout

   vector<int> a;                    ← default constructor (size = 0)
   vector<int> b(4, 3);              ← parameterized constructor (size, initial values)

   cout<<"size of a: "<< a.size()<<endl;
   for(int i=0; i < a.size(); i++)
      cout<<a[i]<<" ";
   cout<<endl;


   cout<<"size of b: "<< b.size()<<endl;
   for(int i=0; i < b.size(); i++)
      cout<<b[i]<<" ";
   cout<<endl;

   for(int i=0; i < 2; i++)
      a.push_back(i);


   cout<<"size of a: "<< a.size()<<endl;
   for(int i=0; i < a.size(); i++)
      cout<<a[i]<<" ";
   cout<<endl;
```

Result:

size of a: 0

size of b: 4
3 3 3 3
size of a: 2
0 1

Other STL Containers

List

- Sequence container that encapsulates **a doubly linked list**
- Use when you need to insert and delete elements without moving existing elements
- Elements are individually allocated

Iterators

- Abstract way to represent traversing a collection
- Can be used for reading or writing (depending on iterator type)
- To use them, collection has begin() and end() class methods

Iterator : Why do you need this?

```
vector<int> data;
data.push_back(1);
data.push_back(2);
data.push_back(3);

for(int i=0;i<data.size();i++)
    cout<<data[i]<<endl;
```

```
list<int> data;
data.push_back(1);
data.push_back(2);
data.push_back(3);

for(int i=0;i<data.size();i++)
    cout<<data[i]<<endl;
```

compile error (operator[] not defined in list)

```
vector<int> data;
data.push_back(1);
data.push_back(2);
data.push_back(3);

vector<int>::iterator it;
for(it = data.begin(); it != data.end(); it++)
    cout<< *it << endl;
```

```
list<int> data;
data.push_back(1);
data.push_back(2);
data.push_back(3);

list<int>::iterator it;
for(it = data.begin(); it != data.end(); it++)
    cout<< *it << endl;
```

List & Iterators Example

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

int main(){
    //Create a list containing ints
    list<int> l {2, 3, 2, 3, 5, 5};
    //Remove all elements of list with value '3'
    l.remove(3);
    //Remove adjacent duplicate elements of list
    l.unique();
    for(auto it=l.begin();it!=l.end();it++){
            cout<<*it<<endl;
    }
    return 0;
}
```

STL Example: Vector (dynamic array)

```
class City{
  private:
    Vehicle *vlist[100];
     int index;
  public:
     City(){ index = 0;}
     void AddVehicle(Vehicle *v){
        vlist[index++] = v;
     }
     void ShowList(){
        for(int i=0; i < index;i++)
           vlist[i]->ShowData();
     }
};
```

```
#include <vector>
class City{
   private:
      vector<Vehicle*> vlist;
   public:
      void AddVehicle(Vehicle *v){
         vlist.push_back(v);
      }
      void ShowList(){
         for(int i=0; i < vlist.size(); i++)
            vlist[i]->ShowData();
      }
};
```

friend functions

- Private and protected members cannot be accessed from outside of the class.
- friend keyword allows to access the private and protected members.

```
class AAA{
    private:
        int x;
    public:
        AAA(){ x=0;}
        void Print(){ cout<<x<<endl;}
        friend void setX(AAA &a, int x);
};

void setX(AAA &a, int x){
    a.x= x;
}
```

```
int main(){
    AAA a;
    a.Print();

    setX(a, 10);
    a.Print();
}
```

friend classes

```
class AAA{
  private:
    int x;
  public:
    AAA(){ x=0;}
    void Print(){ cout<<x<<endl;}
    friend class BBB;
    void setY(BBB &b, int y);
};
class BBB{
  private:
    int y;
  public:
    BBB(){y=0;}
    void Print(){ cout<<y<<endl;}
    void setX(AAA &a, int x){
      a.x = x;
    }
    //friend class AAA;
};
```

```
int main(){
  AAA a;
  BBB b;
  b.setX(a, 20);
  a.Print();

  a.setY(b, 40);  //compile error
  b.Print();
```

friend allows class BBB
to access AAA's private members.
However, class BBB will NOT allow
access by class AAA.

string class

#include <string>

```
string str1("ABC");
string str2 = "123";
string str3 = str1 + str2;

cout<<"str1: "<<str1<<endl;
cout<<"str2: "<<str2<<endl;
cout<<"str3: "<< str3<<endl;

str1 = str2;
cout<<"str1: "<<str1<<endl;
cout<<"str2: "<<str2<<endl;

cout<<( str1==str2)<<endl;
cout<<( str1==str3)<<endl;
```

Result

```
str1: ABC
str2: 123
str3: ABC123
str1: 123
str2: 123
1
0
```

*Warning ⚠️
≠ String in C (array of characters)