CS217 OBJECT ORIENTED PROGRAMMING

Spring 2020

ABSTRACT CLASS

- Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation.
  - For example, let **Shape** be a base class. We cannot provide implementation of function **draw()** in Shape, but we know every derived class must have implementation of **draw().**
  - **Shape:circle; Shape:triangle; Shape:rectangle**

- An <u>abstract class is,</u> conceptually, a class that cannot be instantiated and is usually implemented as a class that has one or more pure virtual (abstract) functions.

- A pure virtual function is one which **must be overridden** by any <u>concrete</u> (i.e., non-abstract) derived class.

```
class AbstractClass {
public:
  //AbstractClass();
  virtual void AbstractMemberFunction() = 0;
  virtual void NonAbstractMemberFunction1();
  void NonAbstractMemberFunction2();
};
```

ABSTRACT CLASS

- In general an abstract class is <u>used to define an implementation</u> and is intended to be inherited by concrete classes.
- To use **an abstract class**, we must create a concrete **class** that extends the **abstract class** (inheritance) and provide implementations for all **abstract functions**.
- It's a way of forcing a contract between the class designer and the users of that class.
- If we wish to create a concrete class (a class that can be instantiated) from an abstract class we must declare and **define** a matching member function for each abstract member function of the base class.
- Sometimes we use the phrase "**pure abstract class**," meaning a class that exclusively has pure virtual functions (and no data).
- The concept of **interface** is mapped to pure abstract classes in C++.
- ***Interesting Fact****: A class is abstract if it has at least one pure virtual function.*

// pure virtual functions make a class abstract

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

class Test
{
   int x;
public:
    virtual void show() = 0;
    int getX() { return x; }
};

int main(void)
{
    Test t;
    return 0;
}
```

[Error] cannot declare variable 't' to be of abstract type 'Test'

//We can have pointers and references of abstract class type

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

class Base
{
public:
    virtual void show() = 0;
};

class Derived: public Base
{
public:
    void show() { cout << "In Derived \n"; }
};

int main(void)
{
   Base *bp;        // Base *bp = new Derived();
   Derived d;
   bp=&d;
    bp->show();
    return 0;
}
```

PURE ABSTRACT CLASS

- A Pure Abstract Class has only abstract member functions and no data or concrete member functions.
- In general, a pure abstract class is used to define an interface and is intended to be inherited by concrete classes.
- An object-oriented system might use an abstract base class to provide a common and standardized interface appropriate for all the external applications.
- Then, through inheritance from that abstract base class, derived classes are formed that operate similarly.

PURE ABSTRACT CLASS

- The capabilities (i.e., the public functions) offered by the external applications are provided as pure virtual functions in the abstract base class.

- The implementations of these pure virtual functions are provided in the derived classes that correspond to the specific types of the application

ABSTRACT CLASSES

- **Can there be private data members in an abstract class? If , YES, then WHY?**
-

CONSTRUCTOR OF ABSTRACT CLASS

- **Can there be Constructor in an abstract class? If , YES, then WHY?**

A (Pure Abstract): All abstract -> **B**(Abstract): some abstract, some virtual ->**C** (Abstract): some abstract, some virtual ->**D**(Concrete): non virtual + implementation of all inherited abstract functions

GENERICS IN C++

- **Generics** is the idea to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes and interfaces.
- For example, classes like an array, map, etc, which can be used using generics very efficiently. We can use them for any type.
- The method of Generic Programming is implemented to increase the efficiency of the code.
- Generic Programming enables the programmer to write a general algorithm which will work with all data types

GENERICS IN C++

- It eliminates the need to create different algorithms if the data type is an integer, string or a character.

- The advantages of Generic Programming are:
  - Code Reusability
  - Avoid Function Overloading
  - Once written it can be used for multiple times and cases.

- Generics can be implemented in C++ using Templates.

TEMPLATES

- **Template** is a simple and yet very powerful tool in C++.
- Writing Generic programs in C++ is called Templates.
- The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types.
- For example, a software company may need **sort()** for different data types. Rather than writing and maintaining the multiple codes, we can write one **sort()** and pass <u>data type as a parameter</u>.
- C++ adds two new keywords to support templates: **`template`** and **`typename/class`**.
  - C++ requires template sources to be added in their headers.

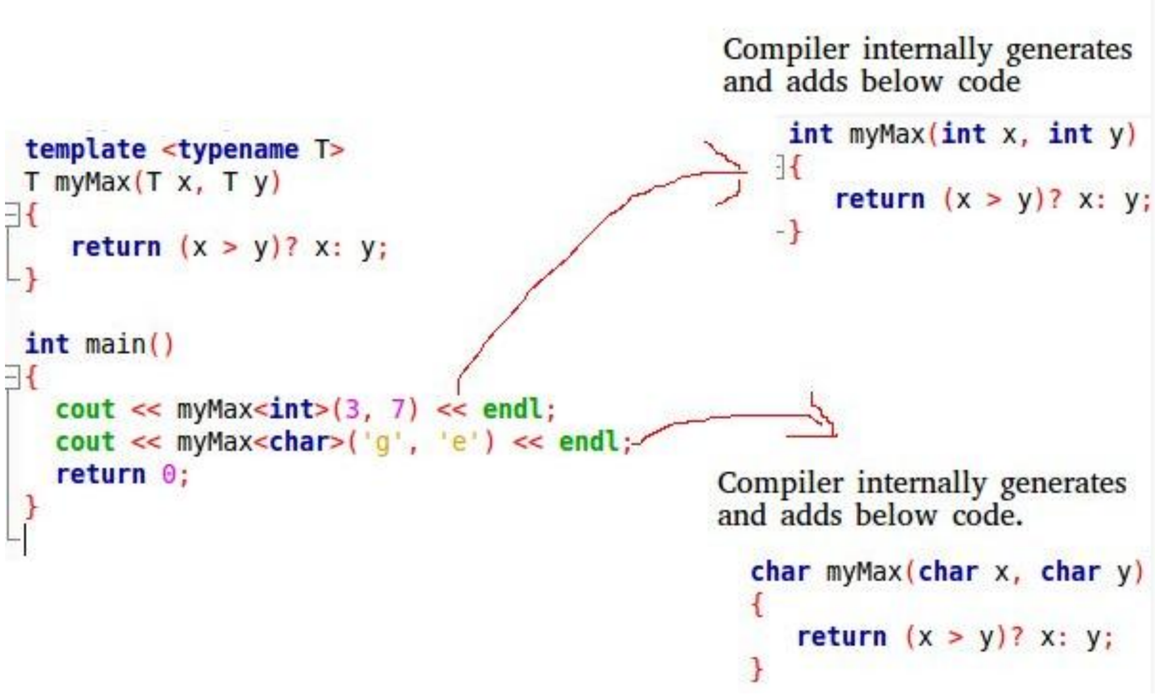
```cpp
template <typename T>
#include <iostream>
using namespace std;

// One function works for all data types. This would work
// even for user defined types if operator '>' is overloaded

T avg(T x, T y){return (x+y)/2}
template <typename T>
T myMax(T x, T y)
{
   return (x > y)? x: y;
}

int main()
{
   cout << myMax (7,3) << endl;  // 7         // Call myMax for int

   cout << myMax(3.1, 7.9) << endl; //7.9  // call myMax for double

    cout << myMax('f', 'e') << endl; //f    // call myMax for char

return 0;
}
```

Function Template

```
template <class T>              //function template
T abs(T n)
{
    return (n<0) ? -n : n;
}
```

TEMPLATES

- Templates are expanded at compiler time, compiler does type checking before template expansion.
- The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.



TEMPLATES

- **What does the compiler do when it sees the** template **keyword and the function definition that follows it?**
- Nothing right away.
  - The function template itself doesn't cause the compiler to generate any code.
  - It can't generate code because it doesn't know yet what data type the function will be working with.
  - It simply remembers the template for possible future use.
- Code generation doesn't take place <u>until the function is actually called</u> (invoked) by a statement within the program.

```
cout << abs(-17);
```

TEMPLATES

- When the compiler sees such a function call, it knows that the type to use is **int**, because that's the type of the argument **-17**. So it generates a specific version of the **abs()** function for type **int**

- This is called *instantiating* the function template, and each instantiated version of the function is called a *template function*.

- The compiler decides how to compile the function based entirely on the <u>data type used in the function call's argument</u> (or arguments). The function's return type doesn't enter into this decision. This is similar to the way the compiler decides which of several overloaded functions to call.

- Notice that the amount of RAM used by the program is the same whether we use the template approach or actually write separate functions.

FUNCTION TEMPLATES WITH MULTIPLE ARGUMENTS

```cpp
#include <iostream>
using namespace std;
//function returns index number of item, or -1 if not found
template <class atype>

int find(atype* array, atype value, int size)
{
    for(int j=0; j<size; j++)
        if(array[j]==value)
            return j;
    return -1;
}

char chrArr[] = {1, 3, 5, 9, 11, 13}; //array
char ch = 5; //value to find
int intArr[] = {1, 3, 5, 9, 11, 13};
int in = 6;
long lonArr[] = {1L, 3L, 5L, 9L, 11L, 13L};
long lo = 11L;
double dubArr[] = {1.0, 3.0, 5.0, 9.0, 11.0, 13.0};
double db = 4.0;

int main()
{
    cout << "\n 5 in chrArray: index= " << find(chrArr, ch, 6);
    cout << "\n 6 in intArray: index= " << find(intArr, in, 6);
    cout << "\n11 in lonArray: index= " << find(lonArr, lo, 6);
    cout << "\n 4 in dubArray: index= " << find(dubArr, db, 6);
    cout << endl;

return 0;
}
```

TEMPLATE ARGUMENTS MUST MATCH

- When a template function is invoked, all instances of the same template argument must be of the same type.

- For example, in **find()**, if the array name is of type **int**, the value to search for must also be of type **int**. You can't say

```
int intarray[] = {1, 3, 5, 7};        //int array
float f1 = 5.0;                       //float value
int value = find(intarray, f1, 4);    //uh, oh
```

because the compiler expects all instances of **atype** to be the same type.

- It can generate a function **find(int*, int, int)**; but it can't generate find(int*, float, int); because the first and second arguments must be the same type

MORE THAN ONE TEMPLATE ARGUMENT

- You can use more than one template argument in a function template.
  - For example, suppose you like the idea of the **find()** function template, but you aren't sure how large an array it might be applied to.
  - If the array is too large then type **long** would be necessary for the array size, instead of type **int**.
  - On the other hand, you don't want to use type **long** if you don't need to.
  - You want to select the type of the array size, as well as the type of data stored, when you call the function.

- To make this possible, you could make the array size into a template argument as well. We'll call it **btype**:

```
template <class atype, class btype>
btype find(atype* array, atype value, btype size)
{
    for(btype j=0; j<size; j++) //note use of btype
        if(array[j]==value)
            return j;

    return static_cast<btype>(-1);
}

find(longArray, longVar, intVar)
find(intArray, floatVar, intVar)
find(intArray, intVar, intVar)
find (charArray, longVar, longVar)
find(floatArray, floatVar, floatVar)
```

CLASS TEMPLATES

- Class templates are generally used for data storage (container) classes. Consider the following class

```
class Stack
{
private:
     int st[MAX];       //array of ints
     int top;           //index number of top of stack

public:
     Stack();           //constructor
     void push(int var); //takes int as argument
     int pop();         //returns int value
};
```

CLASS TEMPLATES

- If we wanted to store data of type long in a stack, we would need to define a completely new class:

```
class LongStack
{
private:
    long st[MAX];           //array of longs
    int top;                //index number of top of stack
public:
    LongStack();            //constructor
    void push(long var);    //takes long as argument
    long pop();             //returns long value
};
```

- Similarly, we would need to create a new stack class for every data type we wanted to store.

CLASS TEMPLATES

- It would be nice to be able to write a single class specification that would work for variables of all types, instead of a single basic type.

- As you may have guessed, **class templates** allow us to do this.

```cpp
#include <iostream>
using namespace std;
const int MAX = 100;    //size of array

template <class Type>
class Stack{

     Type st[MAX];      //stack: array of any type
     int top;           //number of top of stack

public:
     Stack()            //constructor
     {top = -1; }

     void push(Type var) //put number on stack
     { st[++top] = var; }

     Type pop()         //take number off stack
     { return st[top--]; }
};
```

```cpp
int main(){

     Stack<float> s1;   //s1 is object of class Stack<float>
     s1.push(1111.1F); //push 3 floats, pop 3 floats
     s1.push(2222.2F);
     s1.push(3333.3F);
     cout << "1: " << s1.pop() << endl;
     cout << "2: " << s1.pop() << endl;
     cout << "3: " << s1.pop() << endl;

     Stack<long> s2;       //s2 is object of class Stack<long>
     s2.push(123123123L); //push 3 longs, pop 3 longs
     s2.push(234234234L);
     s2.push(345345345L);
     cout << "1: " << s2.pop() << endl;
     cout << "2: " << s2.pop() << endl;
     cout << "3: " << s2.pop() << endl;

     Stack<int> s3;     //s3 is object of class Stack<int>
     s2.push(123123);  //push 3 ints, pop 3 ints
     s2.push(234234);
     s2.push(345345);
     cout << "1: " << s2.pop() << endl;
     cout << "2: " << s2.pop() << endl;
     cout << "3: " << s2.pop() << endl;

return 0;
}
```

CLASS TEMPLATES

- Here the class **Stack** is presented as a template class.

- The approach is similar to that used in function templates. The `template` keyword and class `Stack` signal that the entire class will be a template.

- A template argument, named `Type` in this example, is then used (instead of a fixed data type such as int) everyplace in the class specification where there is a reference to the type of the array `st`.
  - There are three such places: the definition of `st`, the argument type of the `push()` function, and the return type of the `pop()` function

-

CLASS TEMPLATES

- Class templates differ from function templates in the way they are instantiated,
- Classes, however, are instantiated by defining an object using the template argument.

```
Stack <float> s1;
```

- This creates an object, **s1**, a stack that stores numbers of type **float**.
- The compiler provides space in memory for this object's data, using type **float** wherever the template argument **Type** appears in the class specification.
- It also provides space for the member functions. These member functions also operate exclusively on type **float**.

CLASS TEMPLATES

- Similarly, Creating a Stack object that stores objects of a different type, as in **Stack<long> s2**; creates not only a different space for data, but also a new set of member functions that operate on type **long**.

- Note that the name of the type of **s1** consists of the class name Stack *plus the template argument*: `Stack<float>`.
  - This distinguishes it from other classes that might be created from the same template, such as `Stack<int>` or `Stack<long>`.

- In the previus example, the member functions of the class template were all defined within the class. If the member functions are defined externally (outside of the class specification), we need a new syntax.

- The next program shows how this works.

```cpp
#include <iostream>
using namespace std;
const int MAX = 100;
//////////////////////////////////////////////////
template <class Type>
class Stack
{
     Type st[MAX];
     int top;
public:
     Stack();
     void push(Type var);
     Type pop();
};
/////////////////////////////////////////////////
template<class Type>
Stack<Type>::Stack() //constructor
{
     top = -1;
}

template<class Type>
void Stack<Type>::push(Type var)
{
     st[++top] = var;
}

template<class Type>
Type Stack<Type>::pop()
{
     return st[top--];
}
```

```cpp
int main()
{
     Stack<float> s1; //s1 is object of class Stack<float>
     s1.push(1111.1F); //push 3 floats, pop 3 floats
     s1.push(2222.2F);
     s1.push(3333.3F);
     cout << “1: “ << s1.pop() << endl;
     cout << “2: “ << s1.pop() << endl;
     cout << “3: “ << s1.pop() << endl;

     Stack<long> s2; //s2 is object of class Stack<long>
     s2.push(123123123L); //push 3 longs, pop 3 longs
     s2.push(234234234L);
     s2.push(345345345L);
     cout << “1: “ << s2.pop() << endl;
     cout << “2: “ << s2.pop() << endl;
     cout << “3: “ << s2.pop() << endl;
return 0;
}
```