CS217 – Object Oriented Programming (OOP)

Week – 13

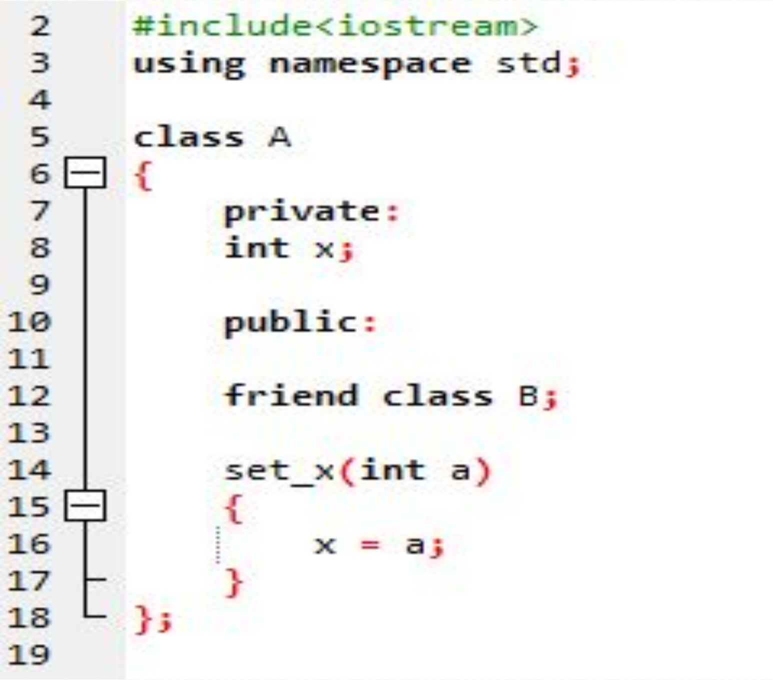
May 3-7, 2021

Instructor: **Basit Ali**

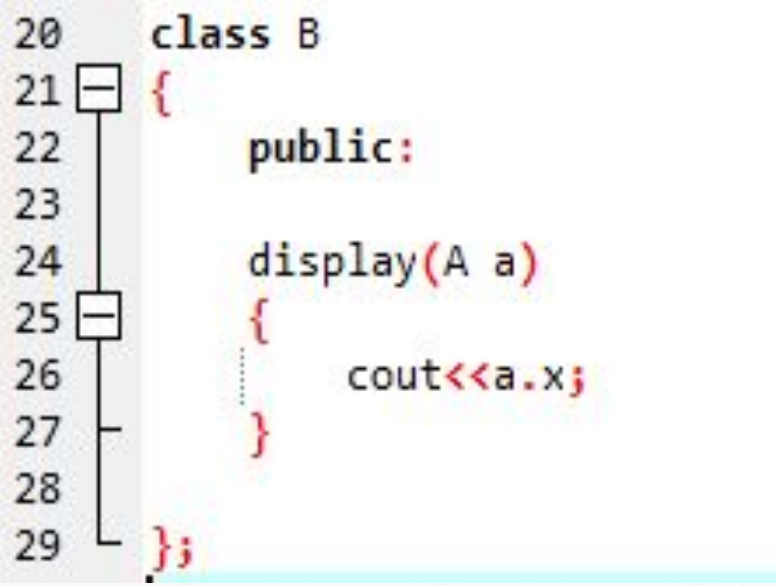
Friend Class

- like a friend function, a class can also be made a friend of another class using keyword.
- A friend class can access all the private and protected members of other class.
- In order to access the private and protected members of a class into friend class we must pass on object of a class to the member functions of friend class.
- When a class is made a friend class, all the member functions of that class becomes friend functions.

Example!

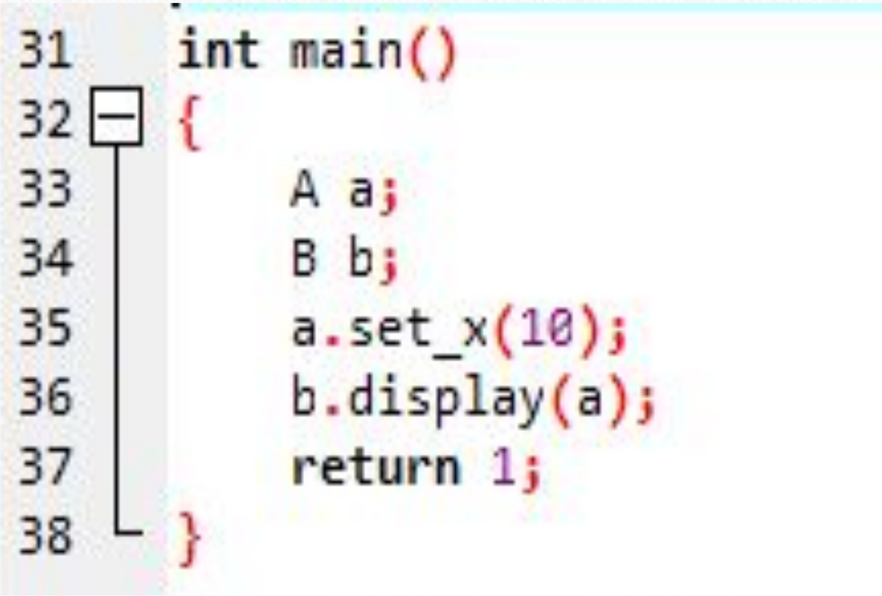


```
#include<iostream>
using namespace std;

class A
{
    private:
    int x;

    public:

    friend class B;

    set_x(int a)
    {
        x = a;
    }
};

```



```
class B
{
    public:

    display(A a)
    {
        cout<<a.x;
    }

};
```

Continue..



```
int main()
{
    A a;
    B b;
    a.set_x(10);
    b.display(a);
    return 1;
}
```

Generic Programming!

- Generic Programming is the idea to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes and interfaces.

- The method of Generic Programming is implemented to increase the efficiency of the code

Generic Programming!

- Generic Programming enables the programmer to write a general algorithm which will work with all data types.

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

The advantages of Generic Programming

- Code Reusability

- Avoid Function Overloading

- Once written it can be used for multiple times and cases.

Generics

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

Templates!

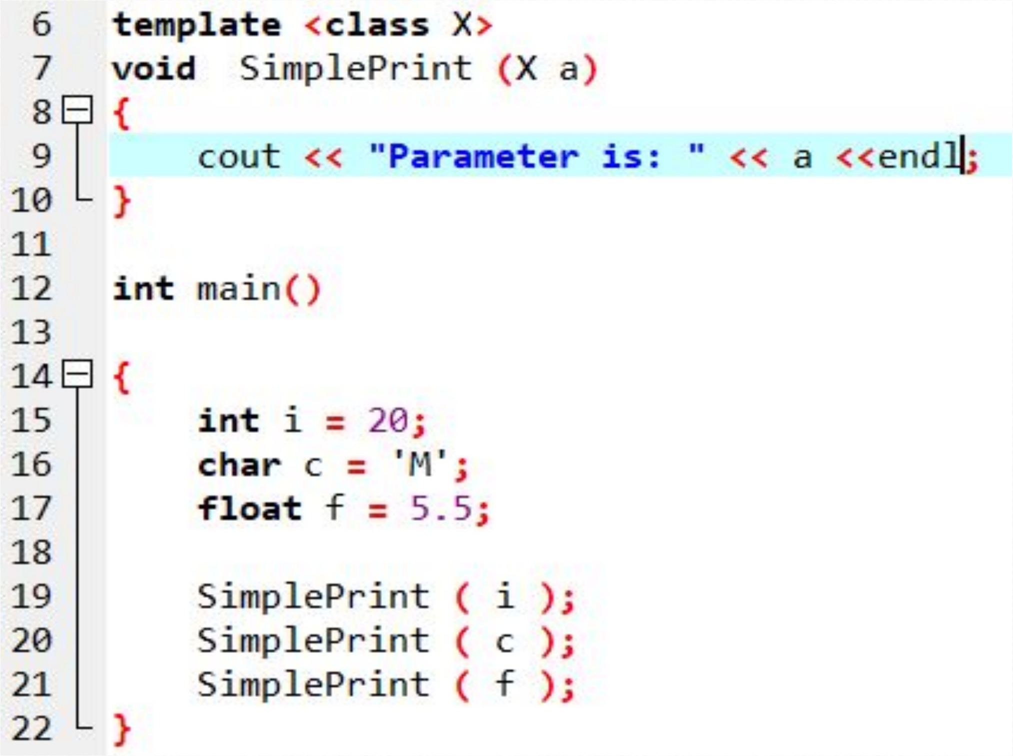
Function Templates

- The general form of a template function definition is:

```
template <class T>
ret-type function-name(parameters)
{
    // body of function
}
```

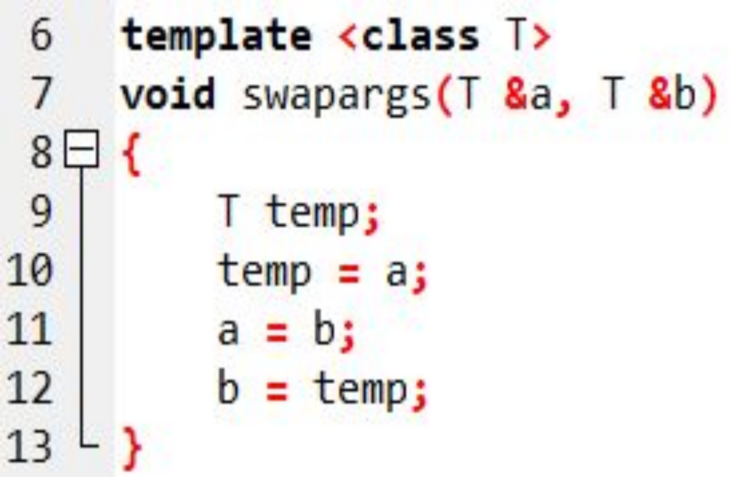
T is a placeholder that the compiler will automatically replace with an actual data type

Example

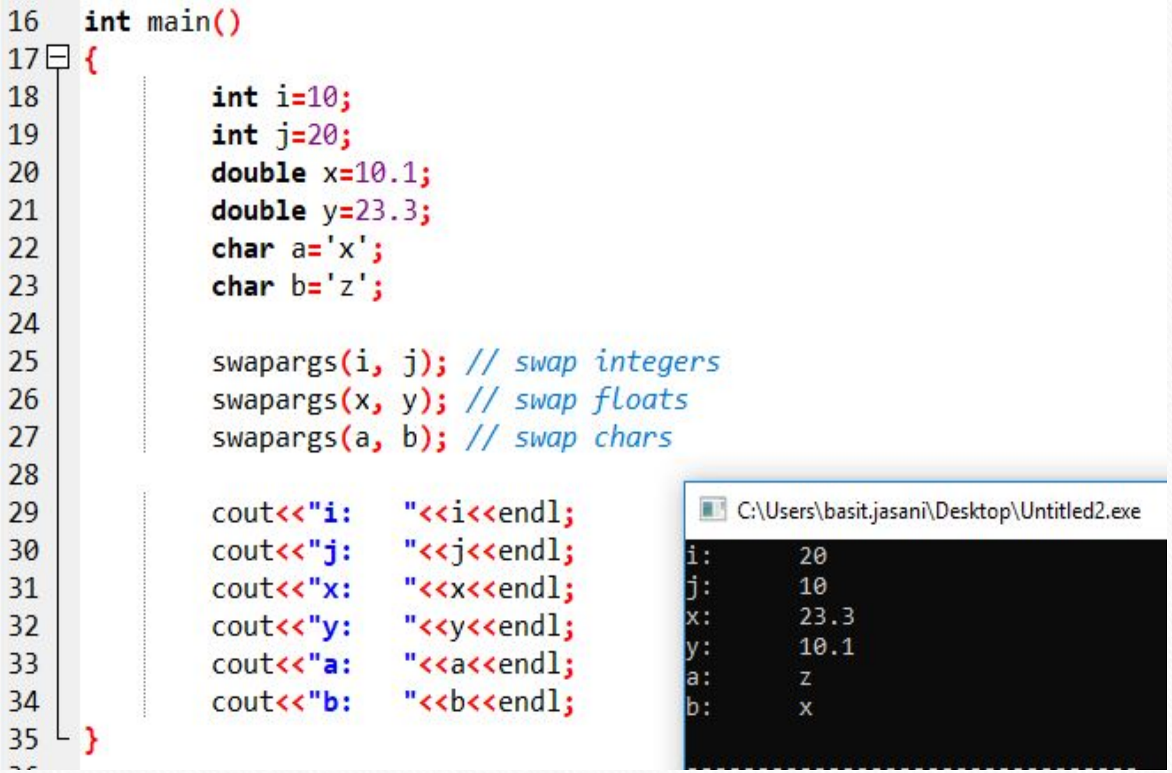


```
template <class X>
void  SimplePrint (X a)
{
    cout << "Parameter is: " << a <<endl;
}

int main()

{
    int i = 20;
    char c = 'M';
    float f = 5.5;

    SimplePrint ( i );
    SimplePrint ( c );
    SimplePrint ( f );
}
```

Example



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



```cpp
int main()
{
        int i=10;
        int j=20;
        double x=10.1;
        double y=23.3;
        char a='x';
        char b='z';

        swapargs(i, j); // swap integers
        swapargs(x, y); // swap floats
        swapargs(a, b); // swap chars

        cout<<"i:   "<<i<<endl;
        cout<<"j:   "<<j<<endl;
        cout<<"x:   "<<x<<endl;
        cout<<"y:   "<<y<<endl;
        cout<<"a:   "<<a<<endl;
        cout<<"b:   "<<b<<endl;
}
```

C:\Users\basit.jasani\Desktop\Untitled2.exe

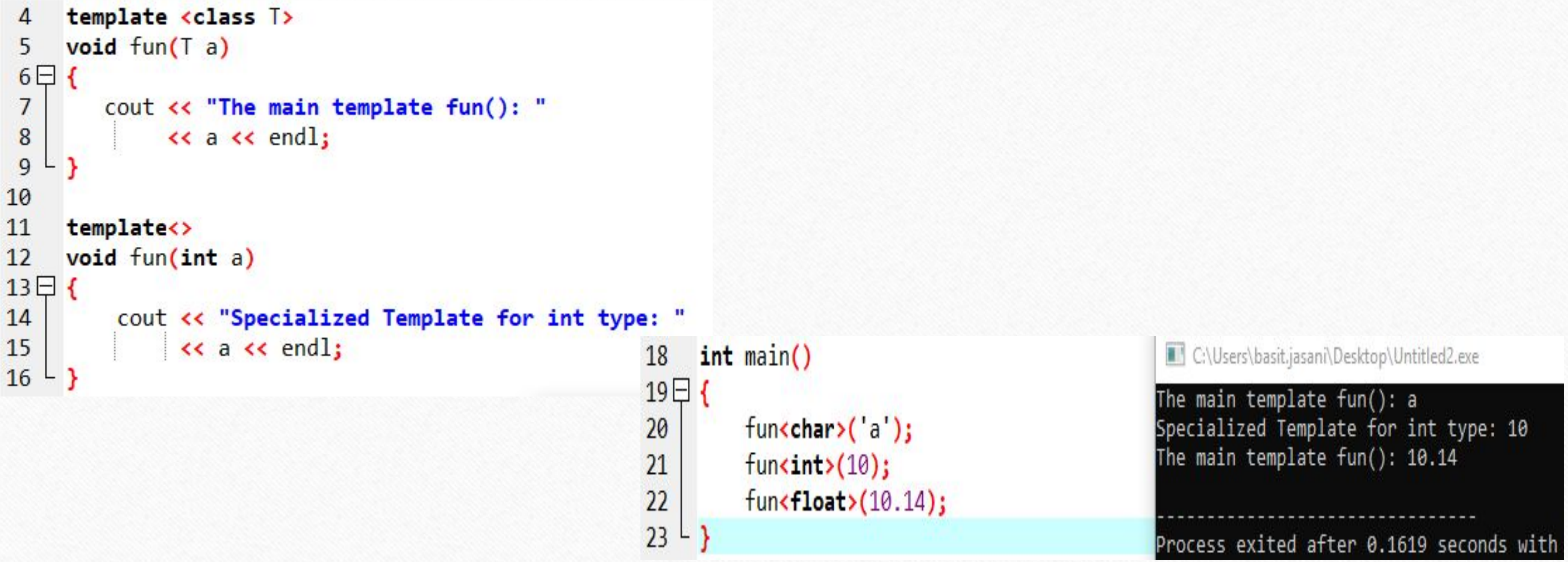
```
i:      20
j:      10
x:      23.3
y:      10.1
a:      z
b:      x
```

Template Function with Two Generic Types

- You can define more than one generic data type in the template statement by using a comma-separated list

```
template <class T1, class T2>
   void myfunc(T1 a, T2 b)
   {
     cout << a << “  &  ” << b << '\n';
   }
```

Specialized Template



```cpp
template <class T>
void fun(T a)
{
    cout << "The main template fun(): "
         << a << endl;
}

template<>
void fun(int a)
{
     cout << "Specialized Template for int type: "
          << a << endl;
}
```

```cpp
int main()
{
    fun<char>('a');
    fun<int>(10);
    fun<float>(10.14);
}
```

C:\Users\basit.jasani\Desktop\Untitled2.exe

```
The main template fun(): a
Specialized Template for int type: 10
The main template fun(): 10.14


--------------------------------
Process exited after 0.1619 seconds with
```

Overloading a Generic Function

- In addition to creating explicit, overloaded versions of a generic function, you can also overload the template specification itself

- To do so, simply create another version of the template that differs from any others in its parameter list

Example

```
// First version of f() template
   template
template <class X>
void f(X a)
{
     cout << "Inside f(X a)";
   b)";
}
```

```
// Second version of f()

template <class X, class Y>
void f(X a, Y b)
     {
     cout << "Inside f(X a, Y

     }
```

Using Normal Parameters in Generic Functions

- You can mix *non-generic parameters* with *generic parameters* in a template function:

```
template<class X> void func(X a, int b){
        cout << “General Data:  ” << a;
        cout << “Integer Data:  ”  << b;
}
```

Generic Classes

- In addition to generic functions, you can also define a *generic class*

- The actual type of the data being used (in class) will be specified as a parameter when objects of that class are created

- Generic classes are useful when a class uses logic that can be generalized e.g. Stacks, Queues

Generic Classes

- The general form of a generic class declaration is shown here:

```
template <class T> class class-name
{
    …
}
```

Generic Classes

- If necessary, we can define more than one generic data type using a comma-separated list
- We create a specific instance of that class using the following general form:

class-name <type> ob;

Example

```
template <class T1, class T2> class myclass {
    T1 i;
    T2 j;
    public:
    myclass (T1 a, T2 b) { i = a; j = b; }
    void show( ) { cout << i << “ & ” << j; }
};
```

Example (cont.)

```
int main(){
    myclass<int, double> ob1(10, 0.23);
    myclass<char, char *> ob2('X', “Hello”);


    ob1.show();    // show int, double
    ob2.show();    // show char, char *
}
```

Using Non-Type Arguments with Generic Classes

- In a generic class, we can also specify non-type arguments:

```
template <class T, int size> class MyClass
{
    T arr[size]; // length of  array is passed in size
    // rest of  the code in class
}
```

Example (cont.)

```
int main()
{
        atype<int, 10> intob;
        atype<double, 15> doubleob;
}
```