COURSE CODE: CSE 205
COURSE TITLE: OBJECT ORIENTED PROGRAMMING

Prepared by- ***Sumaiya Afroz Mila***

An Old Example in C

```
int abs (int n){

  return (n < 0) ? -n : n;

}
long labs (long n){
  return (n < 0) ? -n : n;
}
float fabs (float n){
  return (n < 0) ? -n : n;
}
```

```
main(){
   int i=-3;
   cout<<abs (i);

   long l=-7;
   cout<< labs (l);

   float f=-2.0;
   cout<<fabs (f);
 }
```

An Old Example in C++

```
int abs (int n){

  return (n < 0) ? -n : n;

}
long abs (long n){
  return (n < 0) ? -n : n;
}
float abs (float n){
  return (n < 0) ? -n : n;
}
```

```
main(){
  int i=-3;
  cout<<abs (i);

  long l=-7;
  cout<< abs (l);

  float f=-2.0;
  cout<<abs (f);
 }
```

Using C++ Template

Generic Function/ Template Function

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

```
main(){
  int i=-3;
  cout<<abs (i);

  long l=-7;
  cout<< abs (l);

  float f=-2.0;
  cout<<abs (f);
 }
```

Generic Function

- Programming that works regardless of type is called generic programming.
- A generic function defines a general set of operations that will be applied to various types of data.
- Another type of polymorphism, known as parametric polymorphism.
- Give the user the ability to reuse code in a simple, type-safe manner that allows the compiler to automate the process of type *instantiation*.

Function Template Syntax

- A generic function is created using the keyword **template**.
- Also called Template Function

```
template <class identifier>return_type function_name (arguments of type identifier)
{ … }
```

```
template <class identifier>

return_type function_name (arguments of type identifier) { … }
```

- If Second form is used, no other statement can occur between the template statement and the start of the generic function definition
- Function return type does not take into account to select the template function (just like function overloaded).

Function Template Syntax

- Instead of using the keyword ***class,*** keyword ***typename*** can also be used to specify a generic type in a template definition

```
template <class identifier>

return_type function_name (arguments of type identifier) { … }
```

```
template <typename identifier>

return_type function_name (arguments of type identifier) { … }
```

What Compiler does?

- Function template does not cause the compiler to generate any code (similar to the way a class does not create anything in memory).

- Compiler simply remembers the template for possible future use.

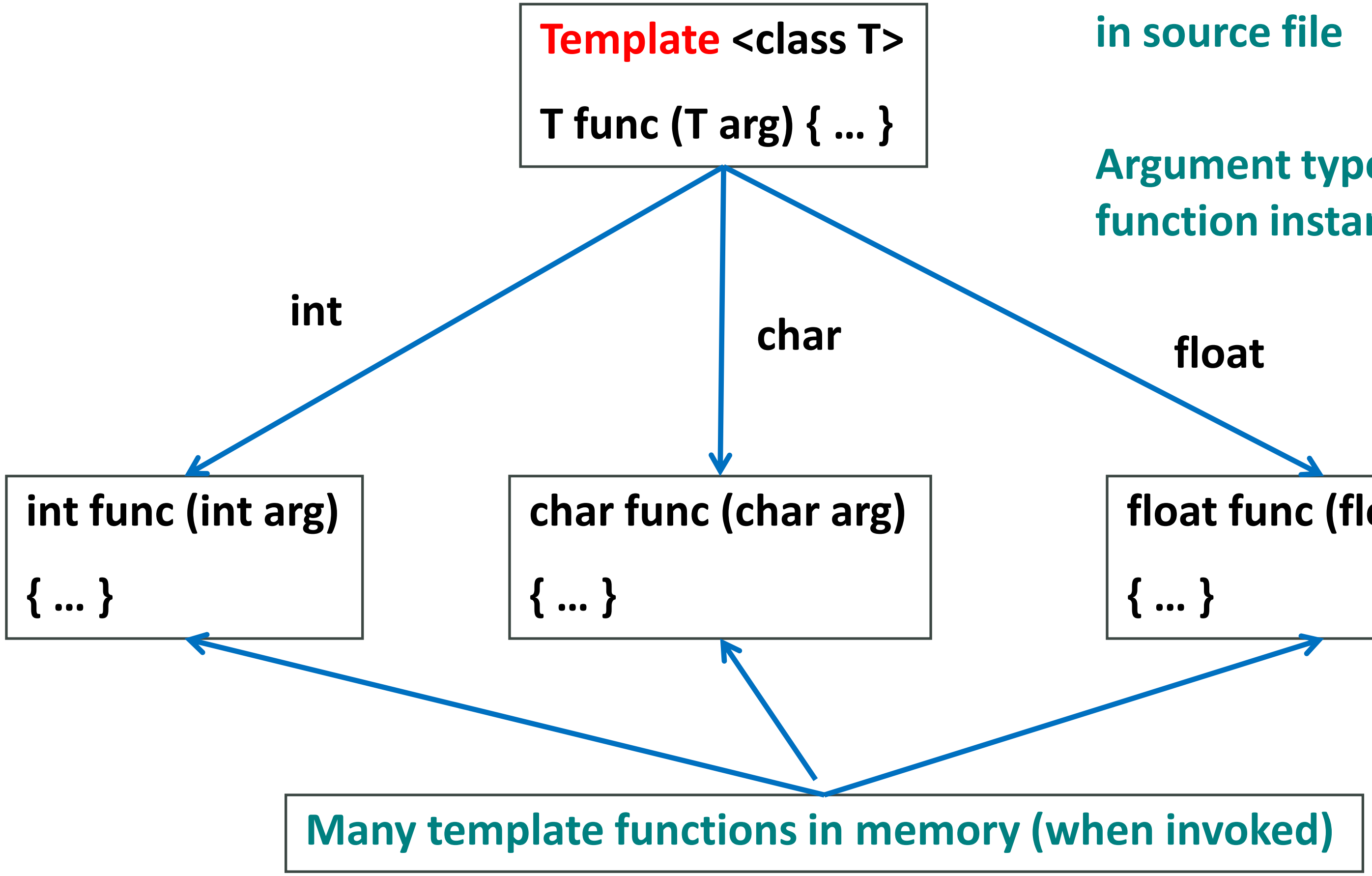
- Code generation does not take place until the function is actually called (invoked).

- Compiler generate a template function (specific version of a function template) depending on the function signature (that's why we called parametric polymorphism).

Function Template Example

One function template
in source file

Argument type determines
function instantiation



Template <class T>

T func (T arg) { … }

int

char

float

int func (int arg)

{ … }

char func (char arg)

{ … }

float func (float arg)

{ … }

Many template functions in memory (when invoked)

Function Template with Multiple Arguments

```
template<class T1, class T2> void copy(T1 a[], T2 b[], int n){ //two arguments template
          for(int i=0;  i<n; i++)
                    a[i] = b[i];
}
```

```
main(){
  char c[50] = {‘a’, ‘b’, ‘d’, ‘e’, ‘f’};
  int i[50] = {10, 20 30, 40, 50};
  float f[50] = {1.1, 2.2, 3.3, 4.4, 5.5};
  copy (i,f,5);     // ok, but loss of data
  copy (c,f,5);   // ok, but loss of data
  copy (f,i,5);   // ok, int to float
  copy (f,c,5);  //ok, char to float
}
```

Function Template with Multiple Arguments

```
template <class T> void swap( T& x, T& y){
    T tmp;
    tmp = x;
    x = y;
    y = tmp;
}
```

```
main(){
     int x, y;
    char c;
    double m, n;
    swap( x, y );
    swap( m, n );
    swap( x, m ); // error
}
```

Generic Function vs Function Overloading

- When functions are overloaded, one can have different actions performed within the body of each function but **generic function must perform the same general version for all actions.**

- A generic function can be overloaded. In that case the overloaded function overrides the generic function relative to that specific version.

Generic Function vs Function Overloading

```
template <class X>
 void func(X n){
     cout << n << endl;
}
```

```
template <class X, class Y>
 void func(X a, Y b){
     cout <<a<<"  " << b << endl;
}
```

```
main(){
     func(10);
     func(20, 30);
}
```

Generic Function vs Function Overloading

```
template <class X>
 void func(X n){
    cout << “Inside Generic Function” << endl;
}
```

```
int func(int n){
    cout<<“Inside Specific Function: ”<<n;
}
```

```
main(){
         int i=10;
         func(i);
         double d=5.5;
         func(d);
}
```

Generic Class

- A generic class defines the algorithm used by the class but the actual type of data is specified when object of that class are created.

- The *identifier* is a template argument that essentially stands for an arbitrary type.

- The template argument can be used as a type name throughout the class definition.

```
template < class identifier >

class classname { … };
```

```
classname<type> ob; //Object Instantiation
```

Generic Class

```
// template stack implementation
template <class TYPE>
class stack {
  TYPE*  s;
  int  top;
  public:
   stack (int size =100){
         s = new TYPE[size];
         top=0;
    }
  ~stack() { delete []s; }
    void  push (TYPE c){
           s[top++] = c;   }
   TYPE  pop(){
           return s[--top];  }
};
```

```
void main(){

 //100  char stack
 stack<char>    stk_ch;

 //200   int stack
 stack<int>    stk_int(200);

 //20  float stack
 stack<float>    stk_float(20);
}
```

Thank you!