C++ Templates

Dr. Md. Humayun Kabir
CSE Department, BUET

C++ Templates

- Templates allows to create a generic function or class for several data types instead of creating several specific functions or classes for several data types.
- A generic function or class defines a general set of operations that will be applied to the data (of various types).
- The type of the data is specified as the parameter whenever a specific version of the function or class for a specific data type is required.
- Helps programmer to write reusable code
- Helps programmer to write compact code

C++ Generic Function with Template

- Template allows to create a function that can automatically overload itself !!!
- General form

```
template <class Ttype1, class Ttype2, …, class TtypeN>
ret-type func-name (param list)
{
   // body of function
}
```

Here,

template is a keyword

We can use keyword “typename” in place of keyword “class”

“TtypeN” is the placeholder for data types used by the function

C++ Generic Function with Template

```
template <class X> void swapargs(X &a, X &b) {
   X temp;
   temp = a;
   a = b;
   b = temp;
}
template <class X> void print (X x, X y) {
   cout << x << “, ” << y << endl;
}
```

C++ Generic Function with Template

```
void main() {
   int i = 10, j = 20;
   double x = 11.11, y = 22.22;
   print(i, j); // 10, 20
   swapargs (i, j); // (int, int)
   print(i, j); // 20, 10
   print(x, y); // 11.11, 22.22
   swapargs (x, y); //(double, double)
   print(x, y); // 22.22, 11.11
}
```

C++ Generic Function with Template

- The compiler generates as many different versions of a template function as required
- Generic functions are more restricted than overloaded functions
  - Overloaded functions can alter their processing logic
  - But, a generic function has only a single processing logic for all data types
- We can also write an explicit overload of a template function

C++ Generic Function with Template

```
template <class X>
void swapargs(X &a, X &b) { cout << “template
    version\n”; }
void swapargs(int &a, int &b) { cout << “int version\n”; }
void main() {
   int i = 10, j = 20;
   double x = 11.11, y = 22.22;
   swapargs(i, j); // “int version”
   swapargs(x, y); // “template version”
}
```

C++ Generic Class with Template

- Makes a class data-type independent
- Useful when a class contains generalized logic
  - A generic stack
  - A generic queue
  - A generic linked list etc. etc. etc.
- The actual data type is specified while declaring an object of the class
- General form

```
template <class Ttype1, class Ttype2, …, class TtypeN>
class class-name
{
  // body of class
};
```

C++ Generic Class with Template

```
template <class X>
class stack {
   X stck[10];
   int tos;
public:
   void init( ) { tos = 0; }
   void push(X item);
   X pop( );
};
```

C++ Generic Class with Template

```
template <class X>
void stack<X>::push(X item) {
    if(tos==10){
        cout<<“Stack is full”<<endl;
        return;
    }
    stck[tos++]=item;
}
```

C++ Generic Class with Template

```
template <class X>
X stack<X>::pop( ) {
   if(tos==0){
        cout<<“Stack Empty”<<endl;
        return 0;
   }
   return stck[--tos];
}
```

C++ Generic Class with Template

```
int main( ) {
   stack<char> s1, s2;
   s1.init( );
   s2.init( );
   s1.push('a');
   s1.push('b');
   s2.push('x');
   s2.push('y');
   cout << s1.pop( ); // b
   cout << s2.pop( ); // y
   stack<double> ds1, ds2;
   ds1.init( );
   ds2.init( );
   ds1.push(1.1);
   ds1.push(2.2);
   ds2.push(3.3);
   ds2.push(4.4);
   cout << ds1.pop( ); // 2.2
   cout << ds2.pop( ); // 4.4
}
```

cout<<“Thank You”<<endl;

cout<<“**Have a Good Day**”<<endl;