Generic functions
Generic classes
Type name and export keyword

Lecture 28-29-30

INTRODUCTION

- Templates allow programmers to write functions and classes to be defined with any type or of a general type.
- Until now, we have been writing non-template functions in which the function parameters are declared to be of a particular type. While calling the function, the arguments passed to the function must strictly match the data types of the parameters.
- However, with a template function, that function can be called with any compatible type.
- Templates act as a model of function or a class that can be used to generate functions or classes. During compilation of a program that has templates, the C++ compiler generates one or more functions or classes based on the specified template.

INTRODUCTION

- Therefore, through templates, C++ supports the concept of generic programming.
- Generic programming is a technique of programming in which a general code is written first.
- The code is instantiated only when the need arises for specific types that are provided as parameters.

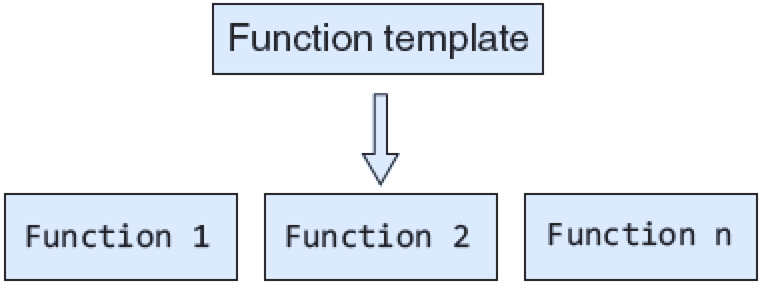

Figure 14.1(a) Generated functions

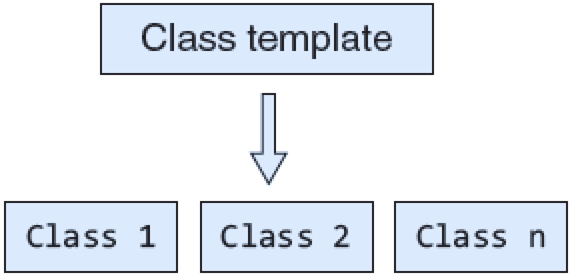

Figure 14.1(b) Generated classes

Templates

- Using templates, it is possible to create generic classes and functions.
- In generic class or function, the type of data upon which generic class or function operates is specified as parameter.
- One can use function or class with several types of data without explicitly defining version of each data type.

Function templates

- It defines a general set of operations that will be applied to various types of data.
- Syntax:

```
template <class type> ret-type func-name (parameter list)
{
        // body of function
}
```

- Template specification must directly precede the function definition.

FUNCTION TEMPLATE

- This enables programmers to write functions without having to specify the exact type(s) of some or all of the variables.
- Instead, we define the function using placeholder types (i.e. for which no data type is specified), called template type parameters.

Keyword

Template data types

Function prototype using arguments with template types

```
template< class type, ....>

return_typeFunc_name(Type arg1, ....)
{          ------------
           Body of Template Function
           ----------
}
```

Figure 14.2 Syntax of a function template

Example

Function template example.

```
#include <iostream>
using namespace std;
template <class X> void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
b = temp;
}
int main()
{
int i=10, j=20;
double x=10.1, y=23.3;
char a='x', b='z';
cout << "Original i, j: " << i << ' ' << j << '\n';
cout << "Original x, y: " << x << ' ' << y << '\n';
cout << "Original a, b: " << a << ' ' << b << '\n';
swapargs(i, j); // swap integers
swapargs(x, y); // swap floats
swapargs(a, b); // swap chars
cout << "Swapped i, j: " << i << ' ' << j << '\n';
cout << "Swapped x, y: " << x << ' ' << y << '\n';
cout << "Swapped a, b: " << a << ' ' << b << '\n';
return 0;
}
```

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

Function templates with multiple parameters

- #include <iostream>
- using namespace std;
- template <class type1, class type2>
- void myfunc(type1 x, type2 y)
- {
- cout << x << ' ' << y << '\n';
- }
- int main()
- {
- myfunc(10, "I like C++");
- myfunc(98.6, 19L);
- return 0;
- }

```
10 I like C++
98.6 19
```

Class templates

- Class templates are used when a class uses logic that can be generalized.
- The compiler will automatically generate the correct type of object, based upon the type you specify when object is created.
- Syntax:

```
template <class Ttype> class class-name
{
        …….
}
```

- Once you created class template, object can be generated using the following form:

Class-name<type> ob;

CLASS TEMPLATE

- Class templates specify how individual classes can be constructed so that every individual class supports similar operations on different data types.
- Therefore, templates enable programmers to create abstract classes that define the behavior of the class without actually knowing what data type will be handled by the class operations.
- Defining a class template is similar to defining an ordinary class except the two differences:-
- The template class is prefixed with template <class Type> that tells the compiler that a template is being declared.
- The use of Type in declaring members of the class.

CLASS TEMPLATE

The syntax for declaring a class template is as follows:

```
template<class Type1, class Type2, ....>
classclass_name
{   ............
     // Body of the class
     ................
};
```

```
template<class Type>
class Array
{   private:
      Type *arr;
      int size;
    public:
      Array(int n)
     {   arr = new Type [n];
         size = n;
     }
     void read();
     void print();
};
```

A class created from a template is called a template class. The syntax for creating an object of the template class is as follows:

```
class_name<Type>object_name(arguments…);
```

Therefore, to declare objects of the template class `Array`, we may write

```
Array <int>A(5);
Array <char>B(7);
```

KEY POINTS

- C++ allows programmers to specify default types for the template class data types. The syntax for specifying default data types can be given as follows:

```
template<class Type1, class Type2 = int>
classclass_name
{   ……………………..
// Body of the class
…………………….
};
```

- To define a member function of a template class outside the class, it must be trea as a function template and the following syntax must be used.

```
template<class Type>
return_typeclass_name<Type> :: function_name(args….)
{   ……………………..
    // Function Body
    …………………….
};
```

Example: Simple Calculator Using Class Template

```
#include <iostream>
using namespace std;
template <class T>
class Calculator
{
private:            T num1, num2;
public:  Calculator(T n1, T n2)
{
num1 = n1; num2 = n2;
}
void displayResult()
{
cout << "Numbers are: " << num1 <<
    " and " << num2 << "." << endl;
cout << "Addition is: " << add() <<
    endl;
cout << "Subtraction is: "
    <<subtract() << endl;
cout << "Product is: " <<multiply()
    << endl;
cout << "Division is: " << divide()
    << endl;
}
T add() { return num1 + num2; }
T subtract()
 { return num1 - num2; }
T multiply()
 { return num1 * num2; }
T divide() { return num1 / num2; }
};
```

Continue……

```
int main()
{
Calculator<int> intCalc(2, 1);
Calculator<float> floatCalc(2.4, 1.2);

cout << "Int results:" << endl;
intCalc.displayResult();

cout << endl << "Float results:" <<
    endl;
floatCalc.displayResult();

    return 0;
}
```

- **Output :**

```
Int results:
Numbers are: 2 and 1.
Addition is: 3
Subtraction is: 1
Product is: 2
Division is: 2

Float results:
Numbers are: 2.4 and 1.2.
Addition is: 3.6
Subtraction is: 1.2
Product is: 2.88
Division is: 2
```

Example

This function demonstrates a generic stack.

```
#include <iostream>
using namespace std;
const int SIZE = 10;
template <class StackType> class stack {
StackType stck[SIZE]; // holds the stack
int tos; // index of top-of-stack
public:
stack() { tos = 0; } // initialize stack
void push(StackType ob); // push object on stack
StackType pop(); // pop object from stack
};
// Push an object.
template <class StackType> void stack<StackType>::push(StackType ob)
{
if(tos==SIZE) {
cout << "Stack is full.\n";
return;       }
stck[tos] = ob;
tos++;
}
// Pop an object.
template <class StackType> StackType stack<StackType>::pop()
{
if(tos==0) {
cout << "Stack is empty.\n";
return 0; // return null on empty stack
}
tos--;
return stck[tos];       }
int main()
{
stack<char> s1, s2; // create character stacks
int i;
s1.push('a');
s2.push('x');
s1.push('b');
s2.push('y');
s1.push('c');
s2.push('z');
```

Continued…

```
for(i=0; i<3; i++) cout << "Pop s1: " << s1.pop() << "\n";
for(i=0; i<3; i++) cout << "Pop s2: " << s2.pop() << "\n";
// demonstrate double stacks
stack<double> ds1, ds2; // create two double stacks
ds1.push(1.1);
ds2.push(2.2);
ds1.push(3.3);
ds2.push(4.4);
ds1.push(5.5);
ds2.push(6.6);
for(i=0; i<3; i++) cout << "Pop ds1: " << ds1.pop() << "\n";
for(i=0; i<3; i++) cout << "Pop ds2: " << ds2.pop() << "\n";
return 0;
}
```

```
Pop s1: c
Pop s1: b
Pop s1: a
Pop s2: z
Pop s2: y
Pop s2: x
Pop ds1: 5.5
Pop ds1: 3.3
Pop ds1: 1.1
Pop ds2: 6.6
Pop ds2: 4.4
Pop ds2: 2.2
```

Class template with multiple parameters

```cpp
#include <iostream>
using namespace std;
template <class Type1, class Type2> class myclass
{
Type1 i;
Type2 j;
public:
myclass(Type1 a, Type2 b) { i = a; j = b; }
void show() { cout << i << ' ' << j << '\n'; }
};
int main()
{
myclass<int, double> ob1(10, 0.23);
myclass<char, char *> ob2('X', "Templates add power.");
ob1.show(); // show int, double
ob2.show(); // show char, char *
return 0;
}
```

```
10 0.23
X Templates add power.
```

ADVANTAGES OF TEMPLATES

- A template not only enhances code reusability but also makes the code short and easy to maintain.
- A single template can handle different types of parameters.
- Testing and debugging efforts are reduced.
- For non-templated functions or classes, the compiler has to compile all n copies. Since they are all a part of the source-code. However, with a template, the compiler would compile only for required set of data-types.
- This means that compilation would be faster if number of different data-types actually required during implementation is less than.

DISADVANTAGES OF TEMPLATES

- Some compilers provide little support for templates. This reduces the code portability.
- Many compilers do not have clear instructions on when and how to detect a template definition error.
- Even when the compiler detects an error, it takes more time and effort to debug the code and resolve the error.
- Since a separate code is generated for each template type, frequent use of templates can result in larger executables.
- Templates are usually defined in the headers, which exposes the code to the entire program, thereby defeating the concept of information hiding

TypeName and Export Keyword

Typename and export Keywords

- Recently, two keywords were added to C++ that relate specifically to templates: **typename and export. Both play specialized roles in C++ programming. Each is** briefly examined.
- The **typename keyword has two uses. First, as mentioned earlier, it can be** substituted for the keyword **class in a template declaration. For example, the swapargs( ) template function could be specified like this:**

```
template <typename X> void swapargs(X &a, X &b)
{
X temp;
temp = a;
a = b;
```

Typename and export Keywords

```
b = temp;
}
```

Here, **typename specifies the generic type X. There is no difference between using class** and using **typename in this context.**

- The second use of **typename is to inform the compiler that a name used in a template** declaration is a type name rather than an object name. For example,

```
typename X::Name someObject;
```

ensures that **X::Name is treated as a type name.**

- The **export keyword can precede a template declaration. It allows other files to use** a template declared in a different file by specifying only its declaration rather than duplicating its entire definition.

- The C++ keyword export was originally designed to eliminate the need to include a template definition (either by providing the definition in the header file or by including the implementation file).

- Only a few compilers ever supported this capability, such as Comeau C++ and Sun Studio, and the general concensus was that it was not worth the trouble. Because of that, the C++11 standardization committee has voted to remove the export feature from the language. Assuming this meets final approval, export will remain a reserved word but it will no longer have any meaning in the standard.

- If you are working with a compiler that supports the export keyword, it will probably continue to support the keyword via some sort of compiler option or extension until its users migrate away from it.

- If you already have code that uses export, you can use a fairly simple discipline to allow your code to easily migrate if/when your compiler stops supporting it entirely.

Thanks