CS-1201 Object Oriented Programming

Templates

Arbish Akram

Department of Computer Science
Government College University

Virtual Destructors

- Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behavior.
- To correct this situation, the base class should be defined with a virtual destructor.

Virtual Destructor

```
class base {
    public:
    base() {
        cout << "Constructing base \n";  }
    ~base() {
        cout << "Destructing base \n";   }
};
class derived : public base {
    public:
    derived() {
        cout << "Constructing derived \n"; }
    ~derived() {
        cout << "Destructing derived \n";  }
};
int main() {
    derived* d = new derived();
    base* b = d;
    delete b;
    return 0;
}
```

Virtual Destructor

```
class base {
    public:
    base() {
        cout << "Constructing base \n"; }
    Virtual ~base() {
        cout << "Destructing base \n";  }
};
class derived : public base {
    public:
    derived() {
    cout << "Constructing derived \n";  }
    ~derived() {
    cout << "Destructing derived \n";   }
};
int main()
{
    derived* d = new derived();
    base* b = d;
    delete b;
    return 0;
}
```

Templates

- Rewriting the same function body over and over for different types is time-consuming.
- Allow the programmer to write type-independent classes and functions using templates.

```
int abs(int n) {
    // Absolute value of integers
    return (n < 0) ? -n : n;
}
float abs(float n) {
    // Absolute value of floats
    return (n < 0) ? -n : n;
}
```

This is repetitive, as we are defining the same logic for different types (int, float).

Class Templates and Function Templates

Class templates:

- The class declaration is preceded by a line of the form:

```
template <class Type1, class Type2, ..., class Typen>
```

- Where `template` and `class` are keywords, and `Type1`, ..., `Typen` are the names of the type parameters.
- You can use `typename` rather than `class`.
- You typically use `class` if you always expect the type parameter to be a class, and `typename` if the type parameter might be either a class or a primitive type.

Function templates:

- The function declaration is preceded by a line of the form:

```
template <typename Type1, ..., typename Typen>
```

Class Template: Example I

```cpp
// Class template
template <class T>
class Number {
    private:
        T num; // Variable of type T
    public:
        Number(T n) : num(n) {}  // constructor
    T getNum() {
        return num;
    }
};
int main() {
    // create object with int type
    Number<int> numberInt(7);
    // create object with double type
    Number<double> numberDouble(7.7);
    cout << "int Number = " << numberInt.getNum() << endl;
    cout << "double Number = " << numberDouble.getNum() << endl;
    return 0;
}
```

Class Template: Example II

```
template <class T>
class Calculator {
   private:
    T num1, num2;
   public:
    Calculator(T n1, T n2) {
        num1 = n1;
        num2 = n2;
    }
    void displayResult() {
        cout << "Numbers: " << num1 << " and " << num2 << "." << endl;
        cout << num1 << " + " << num2 << " = " << add() << endl;
        cout << num1 << " - " << num2 << " = " << subtract() << endl;
        cout << num1 << " * " << num2 << " = " << multiply() << endl;
        cout << num1 << " / " << num2 << " = " << divide() << endl;
    }
    T add() { return num1 + num2; }
    T subtract() { return num1 - num2; }
    T multiply() { return num1 * num2; }
    T divide() { return num1 / num2; }
};
```

Class Template: Example II

```
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;
}
```

Function Template: Example I

```
template <typename T>
T add(T num1, T num2) {
    return (num1 + num2);
}

int main() {
    int result1;
    double result2;
    // calling with int parameters
    result1 = add<int>(2, 3);
    cout << "2 + 3 = " << result1 << endl;

    // calling with double parameters
    result2 = add<double>(2.2, 3.3);
    cout << "2.2 + 3.3 = " << result2 << endl;

    return 0;
}
```

How the Compiler Processes Template Functions?

Template Declaration:

- The compiler reads the function template and knows it can be used with any type `T`.
- The actual function is not generated at this point.

Template Instantiation:

- When you call the function with a specific type, such as `add(3, 4)`, the compiler generates the function for `int`.
- If you call `add(2.5, 3.5)`, the compiler generates the function for `double`.