TDDE18 & 726G77

Templates & Operator Overloading

Test exam information

- Tuesday 12/12 – 2017
- Two sessions 1.15pm – 3.00pm and 3.15pm – 5.00pm
- Not mandatory but it’s a good thing to at least try out the exam system before the real exam.
- You will get a real exam questions from another similar course as this.
- Live correction of submissions.

Duplicate code – functions

```
int sum(int a, int b) {
        return a + b;
}

int main() {
        cout << sum(1, 2) << endl;
}
```

Duplicate code – functions

```
int sum(int a, int b) {
	return a + b;
}

int main() {
	cout << sum(1, 2) << endl;
	cout << sum(1.0, 2.5) << endl;    // Compiler warning and wrong result
}
```

Duplicate code – functions

```
int sum(int a, int b) {
	return a + b;
}
double sum(double a, double b) {
	return a + b;
}

int main() {
	cout << sum(1, 2) << endl;
	cout << sum(1.0, 2.5) << endl;
}
```

Duplicate code – functions

```
int sum(int a, int b) {
	return a + b;
}
double sum(double a, double b) {
	return a + b;
}

int main() {
	cout << sum(1, 2) << endl;
	cout << sum(1.0, 2.5) << endl;
	cout << sum(“a”, “b”) << endl; // Does not compiles
}
```

Duplicate code – functions

```
int sum(int a, int b) {
        return a + b;
}
double sum(double a, double b) {
        return a + b;
}
string sum(string a, string b) {
        return a + b;
}

int main() {
        cout << sum(1, 2) << endl;
        cout << sum(1.0, 2.5) << endl;
        cout << sum(“a”, “b”) << endl;
}
```

Function templates

- A function template defines a family of functions
- Function templates are special functions that can operate with *generic types.*
- Create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.
- This is achieved by using *template parameters*, which is a special kind of parameter that can be used to pass a type argument: just like regular function parameters can be used to pass values to a function.

Template parameters

- The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

- The only difference between both prototypes is the use of either the keyword class or the keyword typename. The use is indistinct, they have the exact same meaning and behave exactly the same way.

Function templates – example

```
template <typename T>
T sum(T a, T b) {
        return a + b;
}

int main() {
        cout << sum(1, 2) << endl;      // invoking sum(int, int);
        cout << sum(1.0, 2.5) << endl;  // invoking sum(double, double);
}
```

Function templates – example

```
template <typename T>
T sum(T a, T b) {
        return a + b;
}

int main() {
        cout << sum(1, 2) << endl;          // invoking sum(int, int);
        cout << sum(1.0, 2.5) << endl;      // invoking sum(double, double);
        cout << sum<double>(1, 2) << endl; // invoking sum(double, double);
}
```

Function templates – example

```
template <typename T>
T sum(T a, T b) {
        return a + b;
}

int main() {
        cout << sum(1, 2) << endl;          // invoking sum(int, int);
        cout << sum(1.0, 2.5) << endl;      // invoking sum(double, double);
        cout << sum<double>(1, 2) << endl; // invoking sum(double, double);
        cout << sum(‘1’, ‘2’) << endl;      // invoking sum(char, char);
}
```

Function templates – example

```
template <typename T>
T sum(T a, T b) {
        return a + b;
}

int main() {
        cout << sum(‘1’, ‘2’) << endl;  // invoking sum(char, char);
                                        // will return ‘c’ due to ascii table
                                        // value, but what if we want “12”
                                        // instead?
}
```

Function templates and overload resolution

- Function templates can be overloaded with both template functions and normal functions.
- Overload resolution basically goes through the following steps to find a function to match a call:
  - if there is a normal function that exactly matches the call, the function is selected, else
  - if a function template can be instantiated to exactly match the call, that specialization is selected, else
  - if type conversion can be applied to the arguments, allowing a normal function to be used as a unique best match, that function is selected, else
  - overload resolution fails

Function templates – example

```
template <typename T>
T sum(T a, T b) {
        return a + b;
}

string sum(char a, char b) {
        return string(1, a) + string(1, b);
}

int main() {
        cout << sum(‘1’, ‘2’) << endl;  // invoking sum(char, char);
                                       // returns “12”
}
```

Function templates and overload resolution

```
template <typename T>
T const& max(T const& x, T const& y);

int a, b;
double x, y;

max(a, b);              // Ok, a and b have same type, int
max(x, y);              // Ok, x and y have same type, double
max(a, x);              // ERROR, a and x has different type
max<double>(a, x);      // explicit instantiation allows for implicit type
                        // conversation, a is converted to double
```

Duplicate code – class

```
class Value_Int {
       int value;
...
};

class Value_Double {
       double value;
...
};

class Value_Char {
       char value;
...
};
```

Class hierarchy solution (1)

```
class Value {
};

class Value_Int : public Value {
    int value;
};

class Value_Double : public Value {
    double value;
};
```

Class hierarchy solution (2)

- Class hierarchy solves the problem of separating different behavior into different subclasses. As you can see the difference between sub classes are data members. The classes have the same behavior.

```
class Value_Int : public Value {
      int value;
      int getValue();
};

class Value_Double : public Value {
      double value;
      double getValue();
};
```

Different data types

Class templates

```
template <typename T>
class Value {
	T value;
	T getValue();
};
```

Class template instantiation and specialization

- implicit instantiation occurs when the context requires an instance of a class template
  - class template arguments are never deduced

    ```
    vector v; // error: missing template arguments before 'v'
    ```

  - class template member functions are instantiated when called

    ```
    v.push_back(x);
    ```

Keyword: typename

- In a template declaration, ***typename*** can be used as an alternative to class to declare ***type template parameters***

  template <typename T>

- Inside a declaration or a definition of a template, ***typename*** can be used to declare that a ***dependent name*** is a type

Declaration/definition of a template

- A name that is not a member of the current instantiation and is dependent on a template parameter is not considered to be a type unless the keyword typename is used

Declaration/definition of a template

```
template <typename T>
class Foo {
     class Bar {};
     Bar f();
};
```

Inner class Bar

f returns Bar object

`How to declare function f() in cc-file?`

Declaration/definition of a template

```
Foo::Bar f() {
	return Bar{};
}

// error: invalid use of template-name ‘Foo’ without and argument list
```

Declaration/definition of a template

```
template <typename T>
Foo<T>::Bar f() {
        return Bar{};
}

// error: need ‘typename’ before ‘Foo<T>::Bar’ because ‘Foo<T>’ is a dependent scope
```

Declaration/definition of a template

```
template <typename T>
typename Foo<T>::Bar f() {
        return Bar{};
}

// compiles and works
```

Templates – file naming convention

- Header file – where the declarations need to be. Convention is to have the extension .h
- Implementation file – where the implementation needs to be. Convention is to have the extension .tpp

Example:

List.h

List.tpp

Template instantiating

- The compiler needs to have access to the implementation of the methods, to instantiate them with template arguments.
- If these implementations were not in the header, they wouldn't be accessible, and therefore the compiler wouldn't be able to instantiate the template.
- No need to compile the implementation file!

```
// header-file
#ifndef _LIST_H_
#define _LIST_H_
...

#include “List.tpp”
#endif
```

```
// implementation file
// no need to include the h-file
template<typename T>
typename List<T>::List() {
        ...
}
```

operator overloading

- Customizes the C++ operators for operands of user-defined types
- When an operator appears in an expression, and a least one of its operands has a class type then overload resolution is used to determine the user-defined function to be called among all the functions whose signatures match.

Syntactic sugar

- Operator overloading allows syntactic sugar for the programmer.

```
cout << 3;             // instead of operator<<(cout, 3);
++i;                   // instead of i.operator++();
str + “world”;         // instead of str.operator+(“world”);
```

Operator overload - operators

- Any of the following 38 operators can be overloaded

```
+  -  *  /  %  ^  &  |  ~  !  =  <  >  +=  -=  *=  /=  %=
^=  &=  |=  <<  >>  >>=  <<=  ==  !=  <=  >=  &&  ||
++  --  ,  ->*  ->  ()  []
```

Operator overload – restrictions

- The operators :: (scope resolution), . (member access), .* (member access through pointer to member), and ?: (ternary conditional) cannot be overloaded
- New operators such as **, <>, or &| cannot be created
- The overload of operators && and || lose short-circuit evaluation.
- The overload of operator -> must either return a raw pointer or return an object (by reference or by value), for which operator -> is in turn overloaded.

Operator overload – non-member functions

- Implemented on a global scope (not inside a class)

```
cout << “hello”;
```

```
ostream & operator<< (ostream & os, string & s);
```

- The overloads of operator>> and operator<< that take a std::istream& and std::ostream& as the left hand and the user-defined type as the right hand, this must be implemented as non-members.

Operator overload – member functions

- Implemented in a class scope (inside a class)
- The left hand is the class object (and the right is usually the arguments)

```
class Foo {
   int operator+(int);
};

Foo f{};
f + 3;
```

Operator overload – user-defined conversion

- Enables conversion from a class type to another type
- Declared like a non-static member function with no parameters, no explicit return type

operator *conversion-type-id*

```
class Foo {
  operator int();
}
Foo f{};
int t{static_cast<int>(f)};
```

Exam information

- Jan 12th 2018 2pm – 7pm (5 hours)
- Be there early around 1.45pm to log in and prepare.
- 1 C++ book is allowed
- 1 dictionary
- en.cppreference.com – only STL part is available