Overloading and Templates IV

DM2233 Advanced Data Structures & Algorithms

Module Schedule

Week	Lecture	Remarks
1	Overloading and Templates I	
2	Overloading and Templates II	Labour Day (Fri) – Lab 2 Make up on 27-Apr
3	Overloading and Templates III	
4	Overloading and Templates IV	
5	Exception Handling I	
6	Exception Handling II	
7	Preprocessing / Assignment 1	Vesak Day (Mon)
Week 8 and 9: Mid-Sem Break		
10	Sorting and Searching I	
11	Sorting and Searching II	
12	Sorting and Searching III	
13	Binary Tree I	Hari Raya Puasa (Fri)
14	Lab Test	
15	Binary Tree II	
16	Binary Tree III	SG50 Day (Fri)
17	Standard Template Library / Assignment 2	National Day (Mon)

Objective

- Function Overloading
- Function Templates
- Class Templates

Function Overloading

- Imagine that you have a function to add 2 integers and return the result.

```
int Add( int x, int y)
{
         return x+y;
}
```

- What if you need to add 2 double values and return the result?
  - Are you going to do this?

```
double AddDouble( double x, double y)
{
         return x+y;
}
```

- What if you need to add float, short, long etc?

Function Overloading

- More efficient and convenient if we have the functions having the same name, but they are all different function!
- Function Overloading is a feature of C++ that allows us to create multiple functions with the same name,
  - Must have different parameters.

```
int Add( int x, int y)
{
        return x+y;
}

double Add(double x, double y)
{
        return x+y;
}
```

Function Overloading

- How does the system know which version of Add() to call?
  - Depends on the arguments used in the call
    - if we provide two ints, C++ will know we mean to call Add(int, int).
    - If we provide two double numbers, C++ will know we mean to call Add(double, double).
- We can define many overloaded Add() functions,
  - As long as each Add() function has unique parameters.

Function Overloading

- We can also define Add() functions with a differing number of parameters

```
int Add( int x, int y)
{
        return x+y;
}

int Add( int x, int y, int z)
{
        return x+y+z;
}
```

Function Overloading

- What if you want to have same function name returning different data types?

```
int GetValue(void);
double GetValue(void);
```

- The compiler will give you an error message.
  - This is not function overloading!

Function Overloading

- Note that using *typedef* in function overloading, does not count to a different data type

```
typedef char * string;
void Print(string szValue);
void Print(char * szValue);
```

Function Overloading

- When you call an overloaded function, there are three possible outcomes:
  1. A match is found and the call is resolved to a particular overloaded function.
  2. No match is found as the arguments can not be matched to any overloaded function.
  3. An ambiguous match is found where the arguments matched more than one overloaded function.

No issue

Function Overloading

- If no exact match is found, C++ tries to find a match through promotion.
  - Char, unsigned char, and short is promoted to an int.
  - Unsigned short can be promoted to int or unsigned int, depending on the size of an int
  - Float is promoted to double
  - Enum is promoted to int

```
void Print(char *szValue);
void Print(int nValue);

Print('a'); // promoted to match Print(int)

// As there is no Print(char), the char ‘a’ is promoted to an
integer, which then matches Print(int).
```

Function Overloading

- If no promotion is found, C++ tries to find a match through standard conversion. Standard conversions include:
  - Any numeric type will match any other numeric type, including unsigned (eg. int to float)
  - Enum will match the formal type of a numeric type (eg. enum to float)
  - Zero will match a pointer type and numeric type (eg. 0 to char*, or 0 to float)
  - A pointer will match a void pointer

```
void Print(float fValue);
void Print(struct sValue);

Print('a'); // promoted to match Print(float)
```

Function Overloading

- Ambiguous matches

```
void Print(unsigned int nValue);
void Print(float fValue);
Print(3.14159);
```

- All literal floating point values are doubles unless they have the ‘f’ suffix. 3.14159 is a double, and there is no Print(double).
  - It matches both calls via standard conversion.
- Ambiguous matches are considered a compile-time error.

Function Overloading

- Solution to Ambiguous Matches is,
  - define a new overloaded function that takes parameters of exactly the type you are trying to call the function with.
  - explicitly cast the ambiguous parameter(s) to the type of the function you want to call

```
void Print(unsigned int nValue);
void Print(float fValue);

~~Print(3.14159);~~

Print(static_cast<unsigned int>(3.14159)); // will call Print(unsigned int)
```

Function Overloading

- Pros
  - Function overloading can lower a programs complexity significantly while introducing very little additional risk.
  - Works transparently and without any issues.
  - The compiler flags all ambiguous cases, and they can generally be easily resolved.
- Cons
  - nil

Function Templates

- Function Overloading

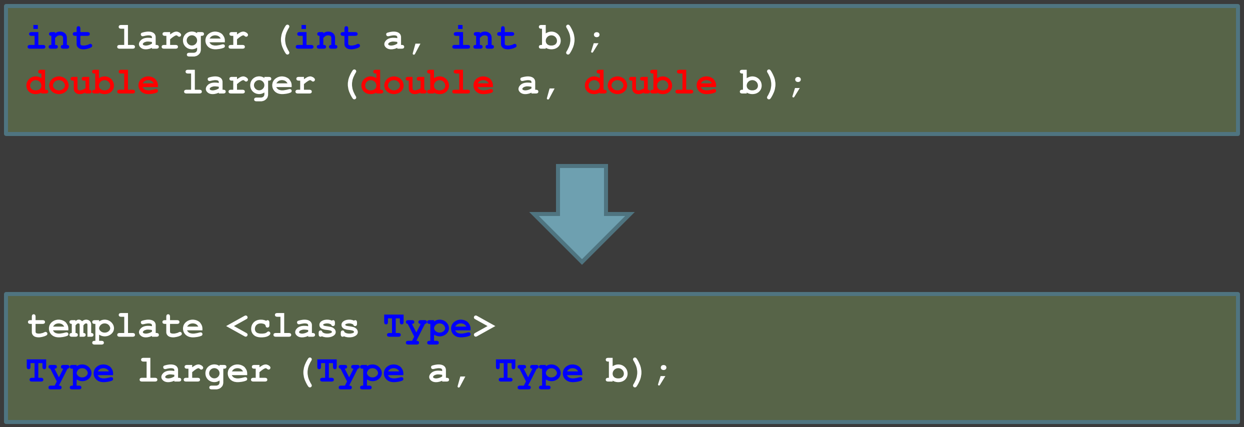
```
int larger (int a, int b) {
  if (a > b) return a;
  else return b;
}

double larger (double a, double b) {
  if (a > b) return a;
  else return b;
}
```

- We need 2 different functions to handle 2 different data types
- Templates allow us to write one set of codes for different data types

Function Templates

- Data types are abstracted out



```
int larger (int a, int b);
double larger (double a, double b);
```

```
template <class Type>
Type larger (Type a, Type b);
```

Function Templates

- Data types are abstracted out

```
int larger (int a, int b) {
  if (a > b) return a;
  else return b;
}

double larger (double a, double b) {
  if (a > b) return a;
  else return b;
}
```

⬇

```
template <class Type>
Type larger (Type a, Type b) {
  if (a > b) return a;
  else return b;
}
```

Function Templates

- No difference in usage

```
int largeInt = larger (5, 6);
double largeDouble = larger (4.56, 3.25);
```

- Write a function that checks the equivalence of 2 similar parameters passed in

```
template <class Type>
bool equal (Type a, Type b) {
  if (abs(a - b) < 0.0001) return true;
  else return false;
}
```

Function Templates

- Pro
  - Save a lot of time, as we only need to write one function, and it will work with many different types.
  - Reduces code maintenance, because duplicate code is reduced significantly.
  - Safer, because there is no need to copy functions and change types by hand whenever you need the function to work with a new type!
- Con
  - Older compilers generally do not have very good template support.
  - Template functions produce weird error messages that are much harder to decipher than those of regular functions.

Class Templates

- Like function templates, class templates allow us to write one set of codes for different data types
- For example, we can have one stack class that can support char, int, double without having 3 sets of codes
- Class templates are called parameterized types because, when instantiating an object, a specific class is created based on the parameter type

Class Templates

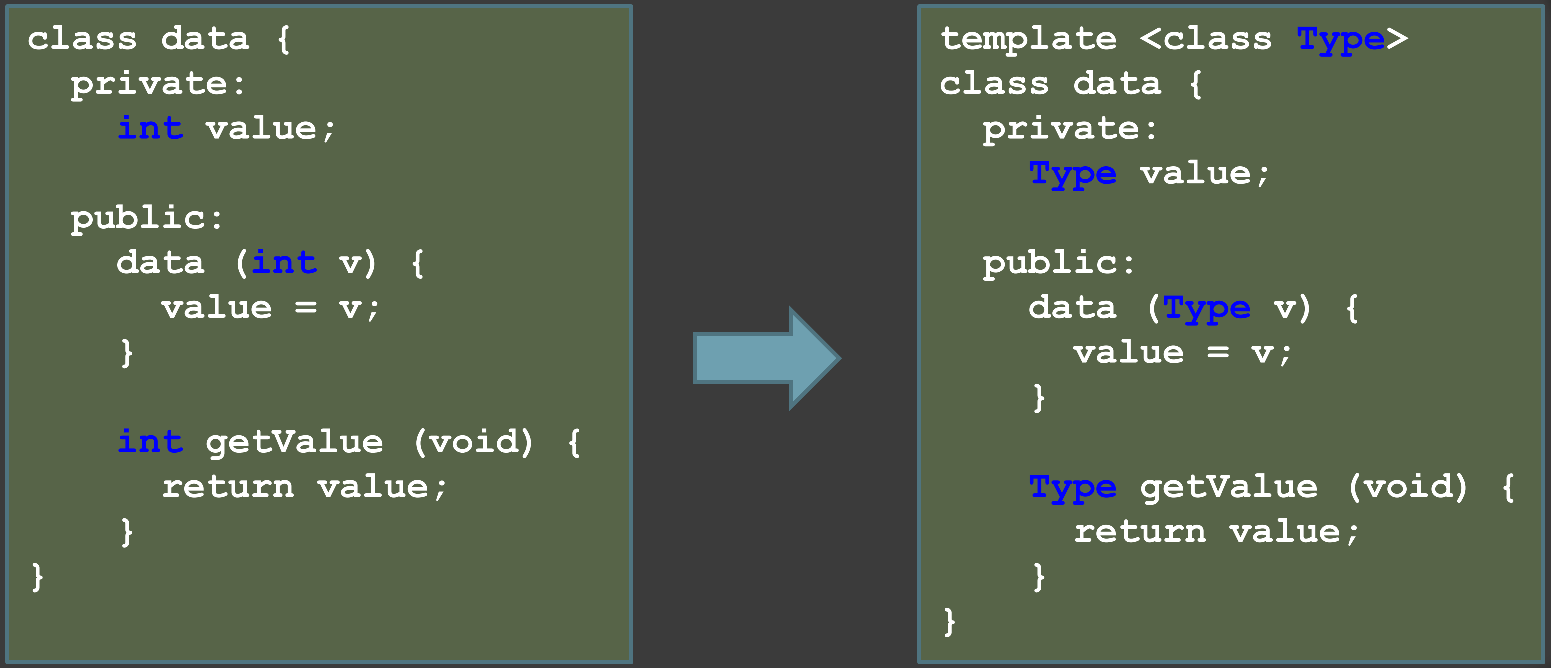
- Function Template

```
template <class Type>
Type larger (Type a, Type b);
```

- Class Template

```
template <class Type>
class stack { ... }
```

Class Templates



```
class data {
  private:
    int value;

  public:
    data (int v) {
      value = v;
    }

    int getValue (void) {
      return value;
    }
}
```

```
template <class Type>
class data {
  private:
    Type value;

  public:
    data (Type v) {
      value = v;
    }

    Type getValue (void) {
      return value;
    }
}
```

Class Templates

```
data obj1 (123);
cout << obj1.getValue() << endl;
```

```
data<int> obj1 (123);
cout << obj1.getValue() << endl;

data<float> obj2 (3.54);
cout << obj2.getValue() << endl;

data<string> obj3 (”Hello”);
cout << obj2.getValue() << endl;
```

Class Templates

- Templated classes are instanced in this way
  - Compiler stencils a copy upon demand
    - With the template parameter replaced by the actual data type that the user needs
    - And then compiles the copy.
  - If you didn’t use a template class in your project, the compile won’t even compile it.
- Ideal for implementing container classes,
  - because it is good to have containers work across a wide variety of data types, and
  - templates allow you to do so without duplicating code.

Class Templates

- Pro
  - Makes C++ very dynamic and powerful.

- Con
  - The syntax is ugly, and
  - The error messages can be cryptic

Summary

- We had just discussed about,
  - Function Overloading
  - Function Templates
  - Class Templates