University of Wollongong in Dubai

UNIVERSITY
OF WOLLONGONG
IN DUBAI

CSCI251 Advanced Programming – Dr HC Lim/Dr Abdellatif Tchantchane

Coverage

- Review Functions
- Generic Programming with C++

review

- Here, we review two areas of functions in C++:
  - Parameters (Students should already know this concept)
  - Overloaded

functions

- Function Overloading
  - Can we have two functions with the same name? The answer is positive (yes) if their parameter lists are different (in type, in number, or in order).
  - In C++, the practice is called function overloading. The criteria the compiler uses to allow two functions with the same name in a program is referred to as the function signature. The signature of a function is the combination of the name and the types in the parameter list.
  - If two function definitions have different signatures, the compiler can distinguish between them.
  - Note that the return type of a function is not included in the signature because C++ must select between overloaded functions when the function is called; the return type is not included in the syntax of the function call.

functions

- The following shows how to define different functions to find the maximum between two integers and two floating-point values.
- The two functions below are considered two different (overloaded) functions because their signatures are different. The first function (the one on the left) has the signature max (int, int); the second function has the signature max (double, double)

```
int max (int a, int b)
{
   ...
}
```

```
double max (double a, double b)
{
   ...
}
```

functions

- The following two functions are not recognized as two overloaded functions because their signatures are the same.
- The first function has the signature get ( ); the second function has also the signature get ( ).
- If we write a program with the above two definitions, it does not compile.

```
int get ()
{
   ...
}
```

```
double get ()
{
   ...
}
```

functions

- Function pointers
  - Much like you can do with variables, you can also do the same with functions; below is a snippet of code that shows a function pointer being defined. Key sections will be highlighted:
  - The basic structure for defining a function pointer is like so
  - <Return_Type> (*<Name>) (<Parameters>)
  - Where in this case:
    - <Return_Type> = void
    - <Name> = functionPtr
    - <Parameters> = int, int

```
1  /*************************************************************
2  * This program shows how we use function pointer.           *
3  *************************************************************/
4  #include <iostream>
5  using namespace std;
6
7  void printAddition(int value1, int value2)
8  {
9      int result = value1 + value2;
10     cout << "The result is: " << result << endl;
11 }
12
13 int main()
14 {
15     //Function pointer definition
16     //<retrunType>(*<Name>)(<Parameters>)
17     void(*functionPtr)(int, int);
18
19     functionPtr = &printAddition;
20
21     //Invoking call to pointer function
22     (*functionPtr)(100, 200);
23     return 0;
24 }
25
```

```
e:\#PrepSpring2021\CSCI251\VS
The result is: 300
Press any key to continue .
```

functions

- Another example with function pointers
- Here what is happening we are passing the function ‘add’ and ‘sub’ as parameters for the function calculator, as you see from the highlight the function parameter is defined like it is above with the return type, name and parameters being defined, all that is passed into calculator is &add and &sub for the function pointers.
- The calculator function then goes on to invoke the pointer and passes in the values and returns the result.

```
e:\#PrepSpring2021\CSCI251\VS Code\lect08\
The result from the operation: 30
The result from the operation: -10
Press any key to continue . . .
```

```
1  /*************************************************************
2  * This program shows how we use function pointer II.         *
3  *************************************************************/
4  #include <iostream>
5  using namespace std;
6  void calculator(int value1, int value2, int(*opp)(int,int))
7  {
8      int result = (*opp)(value1, value2);
9      cout << "The result from the operation: " << result << endl;
10 }
11 //Adds two values
12 int add(int num1, int num2)
13 {
14     return num1 + num2;
15 }
16 //Subtracts two values
17 int sub(int num1, int num2)
18 {
19     return num1 - num2;
20 }
21
22 int main()
23 {
24     calculator(10, 20, &add);
25     calculator(10, 20, &sub);
26     return 0;
27 }
```

Generic programming

- We concentrate on generalization (generic programming), which means to write a general program that can be used in several special cases.
- C++ calls this process template programming (generic programming).
- We first discuss how to write general functions (called function templates) and then we discuss general classes (called class templates).

Generic programming

- Function Template
  - When programming in any language, we sometimes need to apply the same code to different data types. For example, we may have a function that finds the smaller of two integer data types, another function that finds the smaller of two floating-point data types, and so on.
  - First, we should think about the code (program logic) and then about the data type to be used. We can separate these two tasks. We can write a program to solve a problem with a generic data type. We can then apply the program to the specific data type we need. This is known as generic programming or template programming.

Function template

- A function in C++ is an entity that applies operations on zero or more objects and creates zero or more objects.
- Using function templates, actions can be defined when we write a program, and the data types can be defined when we compile the program.
- In other words, we can define a family of functions, each with one or more different data types.

Function template

- Using a Family of Functions
  - If we do not use template and generic programming, we must define a family of functions. Assume we need to compare and find the smaller data item for a variety of data types in a program.
  - For example, assume that we need to find the smaller between two characters, two integers, and two floating-point numbers. Since the types of the data are different, without templates we would need to write three functions as shown in this slide.

```
e:\#PrepSpring2021\CSCI251\VS Code\lect
Smaller of 'a' and 'B': B
Smaller of 12 and 15: 12
Smaller of 44.2 and 33.1: 33.1
Press any key to continue . . .
```

```
1  /*************************************************************
2  * A program to find the smaller between three types of data    *
3  *************************************************************/
4  #include <iostream>
5  using namespace std;
6
7  // Function to find the smaller between two characters
8  char smaller (char first, char second);
9
10 // Function to find the smaller between two integers
11 int smaller (int first, int second);
12
13 // Function to find the smaller between two doubles
14 double smaller (double first, double second);
15
16 int main ()
17 {
18     cout << "Smaller of 'a' and 'B': " << smaller ('a', 'B') << endl;
19     cout << "Smaller of 12 and 15: " << smaller (12, 15) << endl;
20     cout << "Smaller of 44.2 and 33.1: " << smaller (44.2, 33.1) << endl;
21     return 0;
22 }
```

Function template

- Writing three similar functions can be avoided if we use templates and write only one function.
- Using Function Template
  - We give the syntax of a function template to be defined and then the concept of instantiation, which is the way the compiler handles instantiations.
- Syntax
  - To create a template function, we can use a placeholder for each generic type. Slide shows the general syntax for a generic function, in which T, U, …, and Z are replaced by actual types when the function is called.

```
template <typename T, typename U, ..., typename Z>
T functionName  (U first, ... Z last)
{
   ...
}
```

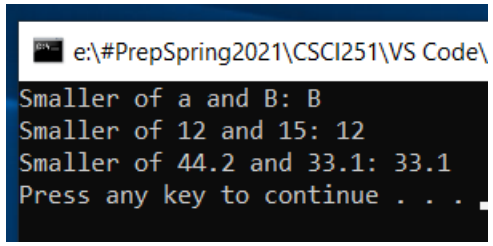
Function template

- As shown in the slide, the template header contains the keyword template followed by a list of symbolic types inside two angle brackets. The template function header follows the rules for function headers except that the types are symbolic as declared in the template header. While multiple generic types are possible, a function template with more than two generic types is rare.
- Some older code uses the term class rather than typename. Here, it uses the keyword typename because it is the current standard and is used in the C++ library.

```
template <typename T, typename U, ..., typename Z>
T functionName  (U first, ... Z last)
{
   ...
}
```

Function template

- In this program we have only one generic type, but the type is used three times: twice as parameters and once as the return type. In this program the type of the parameters and the returned value are the same.
- We can see that the result is the same as in the previous slide. We have saved code by writing only one template function instead of three overloaded functions. Note that we have used only one single type name, T, which is used to define the two parameters and the return value. We did that because the type of the two parameters and the return type are the same.




```cpp
/*************************************************************
* A program that uses a template function to find the smaller  *
* of two values of different types                             *
*************************************************************/
#include <iostream>
using namespace std;

// Definition of a template function
template <typename T>
T smaller (T first, T second)
{
    if (first < second)
    {
        return first;
    }
    return second;
}

int main ( )
{
    cout << "Smaller of a and B: " << smaller ('a', 'B') << endl;
    cout << "Smaller of 12 and 15: " << smaller (12, 15) << endl;
    cout << "Smaller of 44.2 and 33.1: " << smaller (44.2, 33.1) << endl;
    return 0;
}
```

Function template

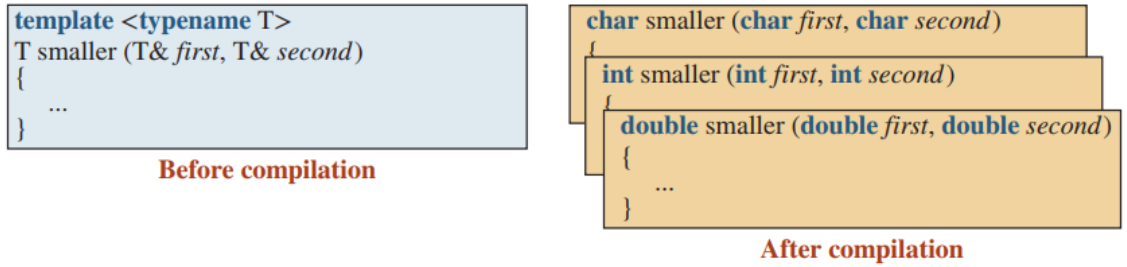
- Slide shows how we can create a generic swap function to exchange two integers, two doubles, and so on.
- Note the use of T& instead of T to allow the objects of the user-defined type to be passed and returned by reference instead of value.

```
e:\#PrepSpring2021\CSCI251\VS Code\lect08\lect08Prep.e
After swapping 5 and 70: 70 5
After swapping 101.5 and 402.7: 402.7 101.5
Press any key to continue . . .
```

```
1  /*************************************************************
2  * A program that uses a template function to swap two values   *
3  *************************************************************/
4  #include <iostream>
5  using namespace std;
6  // Definition of template function
7  template <typename T>
8  void exchange (T& first, T& second)
9  {
10     T temp = first;
11     first = second;
12     second = temp;
13 }
14 int main ()
15 {
16     // Swapping two int types
17     int integer1 = 5;
18     int integer2 = 70;
19     exchange (integer1, integer2);
20     cout << "After swapping 5 and 70: ";
21     cout << integer1 << " " << integer2 << endl;
22     // Swapping two double types
23     double double1 = 101.5;
24     double double2 = 402.7;
25     exchange (double1, double2);
26     cout << "After swapping 101.5 and 402.7: ";
27     cout << double1 << " " << double2 << endl;
28     return 0;
29 }
30
```

Function template

- Instantiation
  - Using template functions postpones the creation of non-template function definitions until compilation time. This means that when a program involving a function template is compiled, the compiler creates as many versions of the function as needed by function calls.
  - In the previous slide, the complier creates three versions of the function named smaller to handle the three function calls with their parameters, as shown in here. This process is referred to as template instantiation, but it should not be confused with instantiation of an object from a type.



Function template

- Variations
  - There are several variations from the basic function template syntax
- Non-type Template Parameter
  - Sometimes we need to define a value instead of a type in our function template. In other words, the type of a parameter may be the same for all template functions that we want to use. In this case, we can define a template, but the type cannot be specifically defined.
  - Assume we want a function that prints the elements of any array regardless of the type of the element and the size of the array. We know that the type of each element can vary from one array to another, but the size of an array is always an integer (or unsigned integer).

Function template

- We have two template parameters, T and N. The parameter T can be any type; the parameter N is a C++type (the type is predefined as integer).

```
e:\#PrepSpring2021\CSCI251\VS Code
7 3 5 1
7.5 6.1 4.6
Press any key to continue . . .
```

```cpp
/*****************************************************************
* A program that uses a template function to print elements of *
* any array of any type                                         *
*****************************************************************/
#include <iostream>
using namespace std;

// Definition of the print template function
template <typename T, int N>
void print (T (&array) [N])
{
    for (int i = 0; i < N ; i++) {
        cout << array [i] << " ";
    }
    cout << endl;
}
int main ()
{
    // Creation of two arrays
    int arr1 [4] = {7, 3, 5, 1};
    double arr2 [3] = {7.5, 6.1, 4.6};
    // Calling template function
    print (arr1);
    print (arr2);
    return 0;
}
```

Function template

- Explicit Type Determination
  - If we try to call the function template to find the smaller between an integer and a floating point value such as the following, we get an error message.

```
cout << smaller (23, 67.2); // Errors! Two different types for the same template type T.
```

Function template

- In other words, we are giving the compiler an integral value of 23 for the first argument of type T and a floating-point value of 67.2 for the second argument of type T.
- The type T is one template type, and the values for it must always be the same as each other. The error with the previous case can be avoided if we define the explicit type conversion during the call.
- This is done by defining the type inside the angle brackets as shown below:

```
cout << smaller <double> (23, 67.2); // 23 will be changed to 23.0
```

Function template

- We are telling the compiler that we want to use the version of the smaller programs in which the value of T is of type double. The compiler then creates that version and converts 23 to 23.0 before finding the smaller.

```
cout << smaller <double> (23, 67.2); // 23 will be changed to 23.0
```

Function template

- Overloading
  - Like overloading of regular functions, we can apply the same concept to function templates. We can overload a function template to have several functions with the same name but different signatures.
  - Normally, the template type is the same, but the number of parameters is different.
  - As an example, we overload the smaller template function to accept two or three parameters (we call it smallest because it uses more than two arguments). Slide shows the interface. Note that we have defined the second function in terms of the first one. This is why the second function is shorter

Function template

- Example of template function overloading

```
h:\#PrepSpring2021\CSCI251\VS Code\lect08\lect08Prep.
Smallest of 17, 12, and 27 is 12
Smallest of 8.5, 4.1, and 19.75 is 4.1
Press any key to continue . . .
```

```cpp
/*****************************************************************
* Overloaded version of the smaller template function           *
*****************************************************************/
#include <iostream>
using namespace std;
// Template function with two parameters
template <typename T>
T smallest (const T& frst, const T& second)
{
    if (frst < second) {
        return frst;
    }
    return second;
}
// Template function with three parameters
template <typename T>
T smallest (const T& frst, const T& second, const T& third)
{
    return smallest (smallest (frst, second), third);
}
int main()
{
    // Calling the overloaded version with three integers
    cout << "Smallest of 17, 12, and 27 is ";
    cout << smallest (17, 12, 27) << endl;
    // Calling the overloaded version with three doubles
    cout << "Smallest of 8.5, 4.1, and 19.75 is ";
    cout << smallest (8.5, 4.1, 19.75) << endl;
    return 0;
}
```

Function template

- Interface and Application Files
  - The definition of a template function can be put in an interface file, and the header can be included in an application file. This means that we can write one definition for a template function and then use it in different programs. The interface file in this case must include the definition of the function, not only the declaration.
  - Slide shows how we can create an interface file, smaller.h, and put the definition of the function template in it. Any program can include this interface and use the function. (see next slide)

```
1  /*************************************************************************
2  * The interface fle for the function template named smaller    *
3  *************************************************************************/
4  #ifndef SMALLER_H
5  #define SMALLER_H
6  #include <iostream>
7  using namespace std;
8  template <typename T>
9  T smaller (const T& frst, const T& second)
10 {
11     if (frst < second) {
12         return frst;
13     }
14     return second;
15 }
16 #endif
```

Function template

- Interface and Application File:

```
h:\#PrepSpring2021\CSCI251\VS Code\lect08\lect0
Smaller of 'a' and 'B': B
Smaller of 12 and 15: 12
Smaller of 44.2 and 33.1: 33.1
Press any key to continue . . .
```

```
1  /*****************************************************************************
2  * The application fle to test a function template                           *
3  *****************************************************************************/
4  #include "smaller.h"
5
6  int main ( )
7  {
8      cout << "Smaller of 'a' and 'B': " << smaller ('a', 'B') << endl;
9      cout << "Smaller of 12 and 15: " << smaller (12, 15) << endl;
10     cout << "Smaller of 44.2 and 33.1: " << smaller (44.2, 33.1) << endl;
11 return 0;
12 }
13
```

Class template

- Class Template
  - The concept of a class template makes the C++ language very powerful. We have learned that a class is a combination of data members and member functions.
  - We may also have a class with data types and another class with the same functionality but with different data types. In these cases, we can use a class template. Templates are used in C++ libraries such as string and stream classes. They are also used in the Standard Template Library, STL, To create a class template, we must make both the data members and the member functions generic.

Class template

- Interface and Implementation
  - A class has an interface and an implementation. When we need to create a class template, we must have generic parameters in both the interface and the implementation.
- Interface
  - The interface of a class must define the typename for both data members and member functions that use the parameterized type. Slide shows the syntax of a simple class template. We use only one data member, one default constructor, and two member functions. The constructor does not use the data member in this case. The accessor function returns a value of type T. The mutator function has one parameter of type T

```
template <typename T>
class Name
{
  private:
      T data;
  public:
      Name ();                    // default constructor
      T get () const;             // accessor
      void set (T data);          // mutator function
};
```

Class template

- Implementation
  - In the implementation, we must mention the typename for each member function that uses the generic type. Slide shows the syntax of the implementation for the simple class we defined in the previous slide.
  - Note that the name of the class that is used before the resolution operator (::) should be Name <T>, not just Name.
  - To concentrate on the syntax, we define a very simple class, which we call Fun, with only one data member (which can be of type int, double, char, or even string). We want to show how the syntax is actually used in defining the interface, the implementation, and the application files of the class.

```
1   #pragma once
2   #ifndef FUN_H
3   #define FUN_H
4   #include <iostream>
5   using namespace std;
6   template <typename T>
7   class Fun
8   {
9   private:
10      T data;
11  public:
12      Fun (T data);
13      ~Fun ();
14      T get () const;
15      void set (T data);
16  };
17  #endif
```

Class template

- Slide shows the implementation file for the template class Fun. Note that each function definition should use a template function with the template <typename T> declaration. Every time we need a type declaration, we use T as a generic type.

```cpp
#pragma once
/*************************************************************
* The implementation file for the template class Fun         *
*************************************************************/
#ifndef FUN_CPP
#define FUN_CPP
#include "fun.h"
// Constructor
template <typename T>
Fun <T> :: Fun (T d)
: data (d)
{ }
// Destructor
template <typename T>
Fun <T> :: ~Fun ()
{ }
// Accessor Function
template <typename T>
T Fun <T> :: get () const
{
    return data;
}
// Mutator Function
template <typename T>
void Fun <T> :: set (T d)
{
    data = d;
}
#endif
```

Class template

- A very important difference we notice in the implementation of a nontemplate class and the one for a template class is that the compiler needs to see the parameterized version of the template function when it compiles the application file. In other words, the application file (defined later) needs to have the implementation file as a header file, which means we need to add the macros ifndef, define, and endif to the implementation file.
- Slide shows the application that uses the template class Fun. Note that we have included fun.cpp as a header file to help the compiler create different versions of the class. Also note that to instantiate template classes (lines 10 to 13), we must define the actual type that replaces the typename T.
- The actual type for the data member of class Fun1 is int, the one for Fun2 is double, the one for Fun3 is char, and the one for Fun4 is a string.

```
e:\#PrepSpring2021\CSCI251\VS Code\
Fun1: 23
Fun2: 12.7
Fun3: A
Fun4: Hello
Fun1 after set: 47
Fun3 after set: B
Press any key to continue . . .
```

```
1  /*****************************************************************
2  * The application file to test the template class Fun           *
3  *****************************************************************/
4
5  #include "fun.cpp"
6
7  int main ( )
8  {
9      // Instantiation of four classes each with different data type
10     Fun <int> Fun1 (23);
11     Fun <double> Fun2 (12.7);
12     Fun <char> Fun3 ('A');
13     Fun <string> Fun4 ("Hello");
14     // Displaying the data values for each class
15     cout << "Fun1: " << Fun1.get() << endl;
16     cout << "Fun2: " << Fun2.get() << endl;
17     cout << "Fun3: " << Fun3.get() << endl;
18     cout << "Fun4: " << Fun4.get() << endl;
19     // Setting the data values in two classes
20     Fun1.set(47);
21     Fun3.set ('B');
22     // Displaying values for newly set data
23     cout << "Fun1 after set: " << Fun1.get() << endl;
24     cout << "Fun3 after set: " << Fun3.get() << endl;
25     return 0;
26 }
```

summary

- Function templates allow us to define a function but defer the definition of the types until the program is compiled. When the program is compiled, the compiler creates as many versions of the function as there are function calls with different types. The syntax includes the reserved words template and typename in angle brackets. To facilitate sharing, function templates are placed in an interface file that is included in the program being compiled.
- There are three rules for type determination in a function template:
  - (1) We cannot mix the types.
  - (2) If mixed types are required, the secondary types must be explicitly typed.
  - (3) The operations in the function, such as compare, must be valid for the types being specified. If they are not, we can use specialization; that is, we can define another function with the specific type to handle the exception. We can overload a template function as long as the signatures are different.

summary

- Class templates can be created by using generic parameters in both the interface and the implementation. The interface of a class must define the typename for both data members and member functions that use the parameterized type.
- In the implementation, we must use the typename for each member function that involves the generic type. However, the compiler must see the parameterized version of the template function when it compiles the application file, which means that the implementation file must be included as a header file.
- While the implementation file can be included using either the inclusion approach or by separate compilations, separate compilation is not yet available for all compilers. Class templates can include multiple types as long as each typename is specified with a different identifier.