Lecture 10

Templates

Topics

- Function Templates
- Function Template Arguments as Objects
- Function Templates with Multiple Arguments
- Class Templates
- Default Types for Class Template Parameters

Template Categories

- C++ supports source code reuse in different ways.
- Inheritance method (IS-A) and Composition (HAS-A, nested objects) method provide two ways to reuse code.
- The template feature in C++ provides another way to reuse source code.

- A template is a program code whose parameter data types are generic place holders.
- There are two categories of templates:
  - Function templates
  - Class templates

Reuse example

- Suppose a function is written, that returns the absolute value of a number.
- The function would be written for a particular data type (such as **int**).

```
int  abs (int  n)  {   // absolute value of integers
   return   (n < 0)   ?   -n   :   n;
   // Implicit if statement : if n is negative, then return  -n , else return  n
}
```

- If the parameter is a **float,** then a completely new function has to be written by overloading.

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

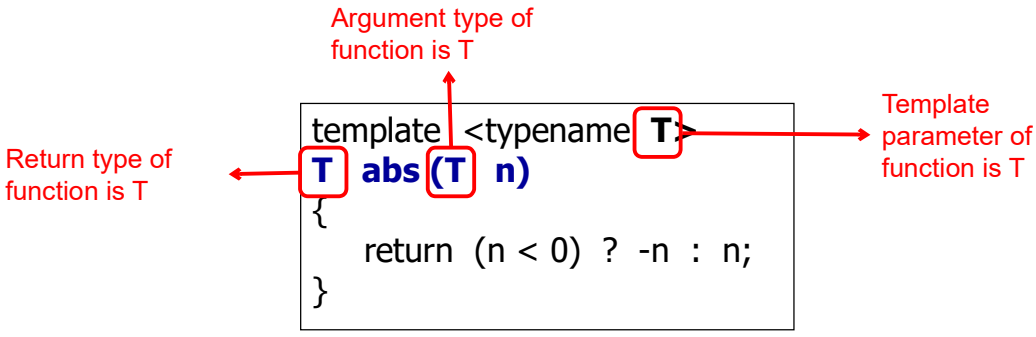
- Commands in the function are the same in each case, but they must be separate functions because they handle variables of different data types (int and float).

Template based programming

- C language does not support overloading, functions for different types can not even have the same name.
- In the C function library, this leads to similarly named functions, such as following absolute value functions :
  - **abs (int n)**
  - **fabs (float n)**
- Rewriting the same function codes again for different types wastes time.
- In C++, the solution is to write a function template, so that it can work for many different data types.
- Template based programming is also known as Generic Programming.

Example : Function Template

- The following is a function template, which works for any numeric data type.
- The "template" and "typename" are C++ keywords.
- The "abs" is the name of function that is defined by programmer.
- The "T" symbol is the name of template parameter.
  (It can be any other valid symbol or identifier name.)




```
int main() {
  // Display absolute numbers
  cout << abs <int> (-5)   <<endl;   // Long calling notation
  cout << abs (-5)         <<endl;   // Short calling notation
}
```

Calling the abs function template

```
int main()
{
  int       i1  =  5;
  int       i2  =  -6;
  long      l1  =  70000L;
  long      l2  =  -80000L;
  double  d1  =  9.73;
  double  d2  =  -10.34;

  // Callings instantiate the template functions
  cout << abs (i1) << endl;     // abs (int)
  cout << abs (i2) << endl;     // abs (int)
  cout << abs (l1) << endl;     // abs (long)
  cout << abs (l2) << endl;     // abs (long)
  cout << abs (d1) << endl;     // abs (double)
  cout << abs (d2) << endl;     // abs (double)
}
```


Topics

- Function Templates
- Function Template Arguments as Objects
- Function Templates with Multiple Arguments
- Class Templates
- Default Types for Class Template Parameters

Function Template Arguments

- In a function template, the data type used by function is not written as a specific type such as int, but as a generic name that can work for any type.
- The **template** keyword signals the compiler that the program is about to define a function template.

- The keyword "typename", within angle brackets <>, indicates the type.
  template < typename T >

- Instead of "typename" keyword, also the "class" keyword can be written.
  template <class T >

- The variable following the keyword class (T in this example) is called the template argument.
- Template argument name can be any valid identifier name.
  (Examples: T, DataType, DT, etc.)

Compilation of Function Template by Compiler

- The function template itself doesn't cause the compiler to generate any code.

- It can't generate code because it doesn't know yet what data type the function will be working with.

- Code is generated (compiled) according to the **function call** statements.

```
int  x;

cout  <<  abs (x);
// Function code is generated by compiler
```

Template Argument Instantiating

- When the compiler sees a function call, it knows that the type to use is integer.
- So the compiler generates a specific version of the **abs (T n)** function for type int, substituting **int** wherever it sees the name **T** in the function template.

  T → **int**
- This process is called ***instantiating*** the function template, and each instantiated version of the function is called a *template function*.
- Data type must support the operations performed in the function.

```
cout << abs ("string");
// Compiler error, because of invalid data type (non-numeric)
```


Template Arguments may be Objects

- Example: A template function named MAX can find maximum of two integers, two floating point numbers, or two complex numbers.
- Integers and floats are built-in data types.
- The ComplexT is a user defined type (class).

```
template <typename T>
const T & MAX (const T &v1 , const T &v2)
{
   if (v1 > v2) return v1;   // Overloaded operator> function called
      else      return v2;
}
```

```
class ComplexT {   // Complex number class
    float re, im;
  public:
    bool operator> (const ComplexT &) ;
    // Prototype of overloaded operator> function
};
```

Calling the MAX function template

```
// Member function for overloaded operator >
bool   ComplexT ::  operator>  (const  ComplexT &    z)  {
   // Compare two hypothenuse lengths
   float  h1    =    re * re    +    im * im;
   float  h2    =    z.re * z.re    +    z.im * z.im;
   return   h1  >  h2;   // Returns true or false
}
```

```
int main() {
   int           i1 = 5,            i2 = -3;
   char          c1 = 'A',          c2 = 'B';
   float         f1 = 3.5,          f2 = 12.4;
   string        s1 = "ABCD",       s2 = "EFG";
   ComplexT   z1 (1.4, 0.6),    z2 (4.6, -3.8);
   cout <<  MAX (i1,  i2)   << endl;
   cout <<  MAX (c1,  c2)   << endl;
   cout <<  MAX (f1,  f2)   << endl;
   cout <<  MAX (s1,  s2)   << endl;
   cout <<  MAX (z1,  z2)   << endl;
}
```

Screen output

```
5
B
12.4
EFG
( 4.6 , -3.8 )
```


Topics

- Function Templates
- Function Template Arguments as Objects
- Function Templates with Multiple Arguments
- Class Templates
- Default Types for Class Template Parameters

Function Templates with Multiple Arguments

- Example: The following find function template takes three arguments. Two template arguments, and one basic data type (integer).
- The function searches an array for a specific value, and returns the array index for that value if it finds it, or -1 if it can't find it.
- The arguments are a pointer to the array, the value to search for, and the size of the array.
- The name of the template argument is T.

```
// Function returns index number of item if found.
// Returns -1 if not found.
template <typename T>
int find (T* array, T searched_value, int size)
{
   for (int j = 0; j < size; j++)
       if ( array [ j ] == searched_value )
          return j;

 return -1;
}
```


Main Program

```
int main() {
   char letters [6]  =  {'a', 'b', 'c', 'd', 'e', 'f'};   // array of letters
   char ch = 'd';    // value to search

   int inumbers [6]  =  {1, 3, 5, 9, 11, 13};
   int in = 7;

   double dnumbers [6]  =  {2.0, 4.0, 6.0, 8.0, 10.0, 12.0};
   double db = 8.0;

   string strings [4]  =  { "AAA", "BBB", "CCC", "DDD" };
   string str = "CCC";

    cout << "Index = " << find (letters,     ch, 6);
    cout << "Index = " << find (inumbers,  in,  6);
    cout << "Index = " << find (dnumbers, db,  6);
    cout << "Index = " << find (strings,     str, 4);
}
```

Template Argument Types

- When a template function is invoked, the compiler expects all template arguments to be the same.
- For example, in **find()**, if the array variable is of type int, the value to search for must also be of type int.

```
int array [4]  =  {1, 3, 5, 7};     // integer array

float f = 5.0;                      // float searched value

int index = find (array , f , 4);
// Compiler error.
// Array is integer, but the searched value is float.
```


Using More Than One Template Argument

- More than one template argument can be used in a function template.
- For example, the type of the array size and also the type of data stored can be specified, when the function is called.

```
template <typename atype, typename btype>
btype find (atype* array,
            atype searched_value,
            btype size)
{
   for ( btype j = 0;  j < size;  j++)   // Using the btype for j
      if ( array [j] == searched_value )
         return j;   // Returned type is btype

   return -1;
}
```

Calling the find function template

- The type int, or type long can be used for the size of array.
- The compiler will generate different functions based not only on the type of the array and the value to be searched for, but also on the type of the array size.

```
short int result1, size1 = 100;
int value1 = 5;

result1 = find (IntArray, value1, size1);
```

```
long result2, size2 = 100000;
float value2 = 5.2;

result2 = find (FloatArray, value2, size2);
```


Topics

- Function Templates
- Function Template Arguments as Objects
- Function Templates with Multiple Arguments
- Class Templates
- Default Types for Class Template Parameters

Class Templates

- The template concept can be applied to classes , as well as to functions.
- Class templates are generally used for data storage (container) classes.

- Stacks and linked lists, are examples of data storage classes.
- Example:The Stack class below can store only integer data.

```
class  Stack
{
       int  st [MAX];          // array of integers
       int  top;               // index number of top of stack

     public:
       Stack ();               // constructor
       void push (int var);    // takes int as argument
       int pop ();             // returns int value
};
```


- If float data are required in a stack, then programmer would need to define a completely new class:

```
   class  FStack  {
      float  st [MAX];         // array of floats
      int  top;                // index number of top of stack
    public:
      FStack ();               // constructor
      void push (float var);   // takes float as argument
      float pop ();            // returns float value
    };
```

- A better solution is to use a **class template** for general purpose:

```
template  <typename  T>
class  Stack  {
    T  st [MAX];               // stack is array of any data type
     int  top;                 // index number of top of stack
  public:
     Stack () {top = 0;}       // constructor
     void  push (T);           // put data on stack
     T  pop ();                // take data off stack
};
```

Defining the Member Functions
(Non-inline Syntax)

When writing a member function with the non-inline syntax notation, the **template** keyword should be written again.

```
template  < typename   T >
void    Stack <T>     ::     push (T   var)
{
  if  (top > MAX-1)              // If stack is full,
      throw  "Stack is full! " ;  // Throw exception and exit

  st [top]  =  var;  // Put variable value on stack
  top++;
}
```

```
template  < typename   T >
T    Stack <T>     ::     pop ()
{
  if  (top <= 0)                    // If stack is empty,
     throw  "Stack is empty! " ;   // Throw exception and exit

  top--;
  return st [top];  // Take top data off stack
}
```


Using the Stack class template

```
int main()
{
  // s1 is object of class  Stack<float>
  Stack <float>    s1;

  // Push two floats, pop two floats
  try
  {
    s1 . push (3.25);
    s1 . push (1.78);
    cout << "1: " << s1.pop() << endl;
    cout << "2: " << s1.pop() << endl;
  }
  // Exception handler
  catch (const  char *  msg)
  {
     cout << msg << endl;
  }
```

```
  // s2 is object of class  Stack<int>
  Stack  <int>   s2;

  // Push two ints, pop two ints
  try
  {
    s2 . push (60);
    s2 . push (70);
    cout << "1: " << s2.pop() << endl;
    cout << "2: " << s2.pop() << endl;
  }
  // Exception handler
  catch (const  char *  msg)
  {
     cout << msg << endl;
  }

  return 0;
}     // End of main program
```

Topics

- Function Templates
- Function Template Arguments as Objects
- Function Templates with Multiple Arguments
- Class Templates
- Default Types for Class Template Parameters

Default Types for Class Template Parameters

- Default data values or default data types can be set, for class template parameters.
- Example : Default type of T is string, default value of N is 10.

```
template  <typename  T  =  string ,  int  N  =  10 >
class  Sequence
{
  T   array [N];
public:
  void  set (int  j ,  T  value);  // j is the index location
  T     get (int  j);
};
```

Class template syntax

Member function signature

```
template  <typename  T,  int  N >
void  Sequence <T,  N>  ::  set (int  j , T  value)
{
   array [j]  =  value;
}
```

Using the Sequence class template

Return type of function is T

```
template <typename T, int N >
T Sequence <T, N> :: get (int j)
{
   return array [j];
}
```

```
int main () {
Sequence <int, 6>      A;   // A has 6 integer elements
Sequence <double, 6>   B;   // B has 6 double elements
Sequence <>            C;   // C has 10 string elements (defaults used)

A . set (2, 50);              // 2nd element of A is assigned
B . set (3, 7.24);            // 3rd element of B is assigned
C . set (8, "ABCD");          // 8th element of C is assigned

cout << A . get (2) << endl;  // 2nd element of A is displayed
cout << B . get (3) << endl;  // 3rd element of B is displayed
cout << C . get (8) << endl;  // 8th element of C is displayed
}
```