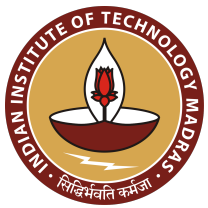
Templates

Saurav Samantaray

Department of Mathematics

Indian Institute of Technology Madras

October 29, 2024



Templates

- If we want to write a function that returns the larger of two numbers, and we want this function to be used for both integer variables and double precision floating point variables, then we could use function overloading and write two functions:
- one for integer variables and the other for double precision floating point variables.
- Both of these functions would require only a few lines of code, and it would not be difficult to maintain both functions.
- For larger functions maintaining more than one function to do the same operations may be problematic.
- This may be avoided by the use of templates, a feature of the C++ language that allows very general code to be written.

Templates to Control Dimensions and Verify Sizes

- Many scientific computing applications are underpinned by vectors and matrices.
- These are represented in C++ by arrays.
- Under normal circumstances there is no check, when we attempt to access elements of an array, that the index is a valid index.
- For example, in the code fragment below we attempt to access the element with index 7 when the array only has 5 elements.

```
double A[5];
A[7] = 5.0;
```

- Although this is clearly an error, it may not trigger a compiler or run-time error.
- The most likely outcome when code including these lines is executed is a segmentation fault or an incorrect answer.

- If this fragment is part of a large program, it could be difficult to locate this error.
- It would therefore be useful if we could use arrays with an additional feature that a check for validity of the index is performed each time an element of the array is accessed.
- This may be achieved using the class shown below, which is referred to as a *templated class*.

Template Declaration Syntax

```
template <parameter list>
template function / class declaration..
```

- The keyword `template` marks the start of a template declaration and is followed by the template parameter list.
- This parameter list contains the keyword `typename` that defines the template parameter `objType`, making it a placeholder for the type of the object that the template is being instantiated for.

Template For Classes

A simple template class that uses a single parameter T to hold a member variable and can be written as the following:

```
template <typename T>
class HoldVarTypeT
{
   private:
      T value;
   public:
      void SetValue (const T& newValue)
         { value = newValue; }
      T& GetValue() {return value;}
};
```

- The type of the variable value is `T`, and that is assigned at the time the template is used, that is, instantiated.
- a sample usage of this template class:

```
HoldVarTypeT <int> holdInt; // template instantiation for int
holdInt.SetValue(5);
cout << "The value stored is: " << holdInt.GetValue() << endl;
```

Templates to Control Dimensions and Verify Sizes

- See ”DoubleVector.hpp”
- The class in the listing allows us to declare instances of `DoubleVector`, specifying the length of the array.
- The entries of the array are private members of this class and so can’t be accessed in the normal way that we would access elements of an array.
- Instead we access members of this class by overloading the square bracket operator.

Templates to Control Dimensions and Verify Sizes

- Overloading this operator allows us to check that the index is a valid index before returning the variable requested.
- using this class requires us to declare the array v as an instance of a `DoubleVector`, with the size of this array being enclosed within pointed brackets.
- Subsequently this array is accessed in exactly the same way as a normal array,
- but with the additional feature that a check is carried out on the index every time an element of the array is accessed through the overloading of the square bracket operator.

```
DoubleVector<5> v;
```

- the above syntax is quite similar to

```
double v[5];
```

Templates for Polymorphism

- Programming languages, distinguish between integer variables and floating point variables, for reasons which are quite prudent.
- the argument(s) used to access an element of an array may only take integer values which provides one level of validation that the index is correct.
- integers may be stored much more efficiently than floating point variables.
- One slight drawback in having to distinguish between these variables is that if we want to write a function that is valid for all numerical variables, i.e. , both integers and floating point variables, we have to write more than one instance of the same function
- `Templates`, however, provide a way around this.

Templates for Polymorphism

- The program "run_poly.cpp" demonstrates how a function `GetMaximum` that returns the maximum of two numbers, either integers or floating point variables, may be written.
- The code is very similar to the code that we would write to calculate the maximum of two numbers, with two important differences.
- The first difference is that the function prototype in the listing specifies that the function is defined for a general class `T`,
- and that the return type and both function arguments will be instances of the same class `T`.
- To call the function, we have to put the data type used in angled brackets as is shown.
- The function `GetMaximum` demonstrates polymorphism, because it can perform the same operation on different types of input argument.
- This type of polymorphism is also called static polymorphism or compile-time polymorphism,
- because when the compiler sees the listing it makes a specific version of `GetMaximum` ready for the `int` or `double` type.

Declaring Templates with Multiple Parameters

- The template parameter list can be expanded to declare multiple parameters separated by a comma.
- So, if we want to declare a generic class that holds a pair of objects that can be of differing types,
- we can do so using the construct as shown in the following sample:

```
template <typename T1, typename T2>
class HoldsPair
{
   private:
      T1 value1;
      T2 value2;
   public:
   // Constructor that initializes member variables
      HoldsPair (const T1& val1, const T2& val2)
      {
         value1 = val1;
         value2 = val2;
      };
   // ... Other member functions };
```

Declaring Templates with Multiple Parameters

- In this example, class `HoldsPair` accepts two template parameters named `T1` and `T2`.
- We can use this class to hold two objects of the same type or of different types

```
// A template instantiation that pairs an int with a double
HoldsPair <int, double>  pairIntDouble (6, 1.99);
```

```
// A template instantiation that pairs an int with an int
HoldsPair <int, int> pairIntDouble (6, 500);
```