Queens College, CUNY, Department of Computer Science

Object Oriented Programming in C++

CSCI 211 / 611

Summer 2018

Instructor: Dr. Sateesh Mane

September 7, 2018

Templates

- In this lecture we shall learn about the important concept of **templates**.
- Templates are employed in the broader concept of **generic programming**.
- Generic programming is an important programming concept.
- Generic programming is independent of any specific language.
- The **Standard Templates Library (STL)** is an important part of the C++ language.
- The STL implements generic programming, and does so via the use of templates.
- Hence its name: Standard "Templates" Library.

1 Introduction

- As with many other general programming concepts, it is helpful to explain the concept of **templates** via a simple example.

- Consider the following three overloads of a function "`printInput`" which simply prints its input parameter x.

```
void printInput(int x)
{
    cout << x << endl;
}

void printInput(double x)
{
    cout << x << endl;
}

void printInput(string x)
{
    cout << x << endl;
}
```

- All three functions are really identical copies of the same underlying function, which is to print the input parameter x.

- All that is different between the functions is the data type of the input parameter x.

- Let us replace the data type of x by a general symbol, say `T`.

- The underlying function looks like the following.

```
void printInput(T x)
{
    cout << x << endl;
}
```

- Basically, to use the above function, we just have to tell the compiler which data type to substitute for `T`.

- This is where **templates** enter the picture.

2 Templated function

- The example in Sec. 1 is a function.
- In this section, we shall learn how to write a **templated function**.
- Later we shall learn to write **templated classes.**
- The "underlying function" above already has almost everything we need.
- To complete the job, we prepend the following code:

```
template<typename T> void printInput(T x)
{
    cout << x << endl;
}
```

- The above is a complete definition of a **templated function**.
  1. *Note: there is an alternative notation.* We can equivalently write the following.

     ```
     template<class T> void printInput(T x)
     {
         cout << x << endl;
     }
     ```

  2. The expressions "**<typename T>**" and "**<class T>**" are equivalent.
  3. It is a matter of personal choice which version you prefer to write.
- The code to use the above function, for the data types `int`, `double` and `string`, is as follows.

```
printInput<int>(n);                // "n" is of type "int"
printInput<double>(d);             // "d" is of type "double"
printInput<string>(s);             // "s" is of type "string"
```

- The above notation is how we tell the compiler which data type to substitute for `T`.
- **Coding a templated function**
  1. It should be obvious that the easiest way to write, test and debug a templated function is to write a function with a specific data type (such as "`printInput`" in Sec. 1) and then substitute the typename `T` in the appropriate places.
  2. Practically everyone does this.

3 Example program: templated function

- Here is a working C++ program to demonstrate the use of the three original functions and the corresponding templated function versions.

```
#include <iostream>
#include <string>
using namespace std;

void printInput(int x)
{
  cout << x << endl;
}

void printInput(double x)
{
  cout << x << endl;
}

void printInput(string x)
{
  cout << x << endl;
}

template<typename T> void printInput(T x)     // templated function
{
  cout << x << endl;
}

int main()
{
  int n = 7;
  double d = 1.2345;
  string s("abcd");
  printInput(n);
  printInput(d);
  printInput(s);

  printInput<int>(n);                          // templated function call
  printInput<double>(d);                       // templated function call
  printInput<string>(s);                       // templated function call
  return 0;
}
```

4 Personal comment on style

- I prefer to write the "**template<typename T>**" on a separate line.

```
template<typename T>
void printInput(T x)
{
    cout << x << endl;
}
```

- Otherwise I find the templated function definition messy to read.
- It is a matter of personal choice.

5 Templates are a prescription

- Note that we must always specify a data type to use a templated function.

  1. We cannot call the templated function "as is" in programming code.
  2. The compiler would not know which data types to substitute for `T`.

- When we specify actual data types for `T`, the compiler automatically generates the code for actual C++ functions with those data types.

- Many authors therefore say that using a template is a "**prescription to write a function**" as opposed to an actual C++ function.

6 References and pointers

- References and pointers work perfectly well with templates.

```
template<typename T> void printInputRef(T &x)        // reference
{
    cout << x << endl;
}

template<typename T> void printInputPtr(T *p)        // reference
{
    cout << *p << endl;
}
```

- Example main program.

```
#include <iostream>
#include <string>
using namespace std;

template<typename T> void printInputRef(T &x)     // reference
{
  cout << x << endl;
}

template<typename T> void printInputPtr(T *p)     // pointer
{
  cout << *p << endl;
}

int main()
{
  int n = 7;
  double d = 1.2345;
  string s("abcd");

  printInputRef<string>(s);
  printInputPtr<double>(&d);
  return 0;
}
```

7 Points to note

- The typename `T` does not have to be a primitive data type.
- Suppose we declare a class `MyClass`. Then we can write the following.

```
MyClass m;
printInput<MyClass>(m);
```

- The compiler generates the following code using the templated function.

```
void printInput(MyClass x)
{
  cout << x << endl;
}
```

- The above example raises some issues.
  1. The above code will not compile unless `MyClass` has a public copy constructor.
  2. The copy constructor may be generated by the compiler or may be written by us, but either way *it must be public.*
  3. Next, the statement "`cout << x << endl;`" must make sense for `MyClass`.
- **If either of the above conditions are not met, the compiler will generate an error.**
- Hence there are some restrictions to the possible data types for `T`.
- The restriction is actually simple to state:
  Will the code compile if we write non-templated code with "`MyClass`" in place of "`T`" everywhere?
- After all, that is exactly what the compiler generates from the templated function.

8 Example & caution

- Let us write a function to sum the values of an array a of type `int` of length n.

```
int sumArray(int n, const int a[])
{
    int sum = 0;
    for (int i = 0; i < n; ++i) {
      sum = sum + a[i];
    }
    return sum;
}
```

- Next let us convert the above into a templated function.

  1. **Not every occurrence of `int` gets replaced by `T`.**
  2. The input "`int n`" really is an integer. It is not converted to a templated data type.
  3. The loop counter really is an integer. It is not converted to a templated data type.
  4. Only the hightlighted occurrences of "`int`" are replaced by `T`.
  5. The input argument "`const int a[]`" will become "`const T a[]`" (note the syntax).
  6. The local variable "`int sum`" will change to a templated data type **T sum**.
  7. **The initialization "`sum = 0`" is an important issue we must address below.**
  8. The return type will also change to "**T**" (the return value can be a templated data type).

- Here is the templated function.

```
T sumArray(int n, const T a[])
{
    T sum;                          // what about the initialization to zero?
    for (int i = 0; i < n; ++i) {
      sum = sum + a[i];
    }
    return sum;
}
```

- **See next page(s).**

- The above templated function for `sumArray` highlights some serious issues.
- Almost certainly the code in a program will look something like this.

```
int n = // etc
MyClass a[] = // etc
MyClass m;
m = sumArray<MyClass>(n, a);
```

- This will work only if `MyClass` has a **public copy constructor and assignment operator.**
- Examine the code in the loop:

```
    sum = sum + a[i];
```

- For this to work, the **overloaded `operator+`** must exist for `MyClass`.
- The overloaded `operator+` can be a class method or an external function, but *it must exist*.
- However, arguably the most serious issue is the declaration **T sum.**
- First of all, the above statement will only work if `MyClass` has a **public default constructor.**
- We cannot invoke a non-default constructor because we have no idea what inputs arguments to supply, for an arbitrary templated data type.
- But that is not the end of it.
- ***What happened to the initialization to zero?***
  1. If we do not initialize `sum = 0`, the value of sum will be an undefined value.
  2. The function will return nonsense data.
  3. Unfortunately, a statement such as "`sum = 0`" is impossible to write for a user-defined class (in general).
  4. **We cannot initialize the variable `sum` to zero.**
  5. **The default constructor must initialize `sum` to a value that can be interpreted as "zero" for the class `MyClass`.**
  6. The definition of an "empty object" will depend on the details of `MyClass`.
  7. However, there is no default constructor for primitive data types such as `int` or `double`.
- ***We cannot use the above templated function for primitive data types such as `int` or `double`.***

Not everything can be templated

9 Templates and polymorphism

- Templates have some features in common with polymorphism.
- Templates and polymorphism are both ways to write abstract code that is not restricted to a specific class or data type.
- With polymorpism, we write a base class which provides some functionality and also a set of virtual functions.
- Derived classes inherit from the base class and override some or all of the virtual methods.
- However, the derived classes are restricted to be members of the inheritance tree.
- A class which is not a member of the inheritance tree cannot be part of a polymorphic library.
- With templates, there is no inheritance tree.
- Users can write (almost) any classes they wish for the templated data type.
- However, as we have seen, there are some restrictions on the properties the classes, to be able to use the templated function.
- Templates and polymorphism are both useful techniques, but they implement "generality" in different ways.

10 Two or more template types

- It is straightforward to write a templated function with two templated data types.
- Let us write a templated function to print two input parameters x and y.
- The data types of x and y are independent.

```
template<typename T, typename U> void printInput(T x, U y)
{
    cout << x << endl;
    cout << y << endl;
}
```

- Code like the above gets messy to read. I prefer to place the prepend on a separate line.

```
template<typename T, typename U>        // easier to read (I think)
void printInput(T x, U y)
{
    cout << x << endl;
    cout << y << endl;
}
```

- Here is a main program. The use of the templated function with two data types is obvious.

```
#include <iostream>
#include <string>
using namespace std;

template<typename T, typename U> void printInput(T x, U y)
{
  cout << x << endl;
  cout << y << endl;
}

int main()
{
  int n = 7;
  double d = 1.2345;
  string s("abcd");
  printInput<int,double>(n,d);          // two templated types int, double
  printInput<double,int>(d,n);          // two templated types double, int
  printInput<int,string>(n,s);          // two templated types int, string
  printInput<string,double>(s,d);       // two templated types string, double
  return 0;
}
```

11 Templated class

- We can write a templated class, not just a templated function.
- As always, begin with a specific data type, say `string`.
- We declare a class "`Sclass`" ("string class") which has (i) a private data member of type `string`, (ii) a non-default constructor, (iii) accessor, (iv) mutator and (v) a public method.

```
class Sclass {
public:
    Sclass(const string &a) { x = a; }
    string get() const { return x; }
    void set(const string &a) { x = a; }
    void print() const { cout << x << endl; }
private:
    string x;
};
```

- The procedure to convert this to a templated class is essentially the same as for a function.
- We replace "`string`" by "`T`" and prepend a "`template<typename T>`" statement.
- We change the class name to "`Tclass`" ("template class").

```
template<typename T> class Tclass {
public:
    Tclass(const T &a) { x = a; }
    T get() const { return x; }
    void set(const T &a) { x = a; }
    void print() const { cout << x << endl; }
private:
    T x;
};
```

- Here is the declaration to overload `operator<<` for the above templated class.

```
template<typename T>
ostream& operator << (ostream &os, const Tclass<T> &t)
{
  return os << "(" << t.get() << ")";
}
```

12 Example program: templated class

- Here is a working C++ program to demonstrate the use of the templated class `Tclass`.
- To instantiate objects of specific data types, we write code such as the following.

```
  Tclass<int>    it(n);             // "n" is type int
  Tclass<double> dt(d);             // "d" is type double
  Tclass<string> st(s);             // "s" is type string
```

```
#include <iostream>
#include <string>
using namespace std;

template<typename T> class Tclass {
public:
  Tclass(const T &a) { x = a; }
  T get() const { return x; }
  void set(const T &a) { x = a; }
  void print() const { cout << x << endl; }
private:
  T x;
};

int main()
{
  int n = 7;
  double d = 1.2345;
  string s("abcd");
  Tclass<int>    it(n);
  Tclass<double> dt(d);
  Tclass<string> st(s);

  it.print();
  dt.print();
  st.print();
  cout << it << endl;
  cout << dt << endl;
  cout << st << endl;

  return 0;
}
```

13 Templated class with non-inline functions definitions

- All the class methods were declared inline in the templated class in Sec. 11.
- Some important modifications are required for non-inline function definitions.
- Here is the same templated class, with all function definitions written non-inline.

```
template<typename T> class Tclass {
public:
  Tclass(const T &a);
  T get() const;
  void set(const T &a);
  void print() const;

private:
  T x;
};


template<typename T>
Tclass<T>::Tclass(const T &a) { x = a; }

template<typename T>
T Tclass<T>::get() const { return x; }

template<typename T>
void Tclass<T>::set(const T &a) { x = a; }

template<typename T>
void Tclass<T>::print() const { cout << x << endl; }
```

- Notice that we must write **Tclass<T>::** (unlike `Tclass::` for non-templated classes).

14 Templated class with non-inline functions definitions, dynamic memory

- Here is the a templated class with non-inline function definitions, dynamic memory allocation and deep copy.
- The class has an object `T obj`, pointer `T *ptr` and dynamic array `T *array`.
- Note in particular the non-inline signature for the assignment `operator=`.

```
template<class T>
class DynAlloc {
public:
  DynAlloc(T a, T b, int n);
  ~DynAlloc();
  DynAlloc(const DynAlloc& orig);
  DynAlloc& operator= (const DynAlloc& rhs);

  T getObj() const { return obj; }               // inline version
  const T* getPtr() const { return ptr; }        // inline version

  void setObj(T a);
  void setPtr(T b);
  T* getElement(int n);

private:
  int len;
  T obj;
  T *ptr;
  T *array;
};
```

```
//template<class T>
//T DynAlloc<T>::getObj() const { return obj; }           // non-inline version

//template<class T>
//const T* DynAlloc<T>::getPtr() const { return ptr; }   // non-inline version

template<class T>
void DynAlloc<T>::setObj(T a) { obj = a; }

template<class T>
void DynAlloc<T>::setPtr(T b) { *ptr = b; }

template<class T>
T* DynAlloc<T>::getElement(int n)
{
  if ((n >= 0) && (n < len))
    return &array[n];
  else
    return NULL;
}

template<class T>                        // memberwise initialization
DynAlloc<T>::DynAlloc(T a, T b, int n) : obj(a), ptr(new T), array(NULL), len(0)
{
  *ptr = b;
  if (n > 0) {
    len = n;
    array = new T[len];
  }
}

template<class T>
DynAlloc<T>::~DynAlloc() {
  delete ptr;
  delete [] array;
}
```

```
template<class T>                              // memberwise initialization
DynAlloc<T>::DynAlloc(const DynAlloc& orig) : array(NULL)
{
  obj = orig.obj;
  ptr = new T;
  *ptr = *orig.ptr;
  len = orig.len;
  if (len > 0) {
    array = new T[len];
    for (int i = 0; i < len; ++i)
      array[i] = orig.array[i];
  }
}

template<class T>
DynAlloc<T>& DynAlloc<T>::operator=(const DynAlloc& rhs)
{
  if (this == &rhs) return *this;
  obj = rhs.obj;
  *ptr = *rhs.ptr;
  len = rhs.len;
  delete [] array;
  if (len > 0) {
    array = new T[len];
    for (int i = 0; i < len; ++i)
      array[i] = rhs.array[i];
  }
  else {
    array = NULL;
  }
  return *this;
}
```

15 Caveats

- The same caveats apply to a templated class as for a templated function.
- If the data type for `T` is a user-defined class `MyClass`, **will the code compile?**
- Inspection of the class declaration in Sec. 11 immediately reveals the following.
  1. `MyClass` must have a public default constructor, else the declaration of the private data member "`T x`" will not compile.
  2. The constructor, accessor and mutator require that `MyClass` must have a public copy constructor and assignment operator.
  3. The print statement "`cout << x << endl`" must make sense for `MyClass`.

16 Generic Programming

- **Generic programming** is a style of programming in which the code is independent of any specific data type.
- **Generic programming is a *concept*, independent of any programming language.**
- Hence the *data type is specified later*, not when the code is actually written.
- Templates provide a mechanism to implement generic programming.
- For example, consider a data structure such as a dynamically resizeable array.
  1. The fundamental software design of the array does not depend on a specific data type such as `int` or `double`, etc.
  2. The “`vector`” class in the STL is a dynamically resizeable array.
  3. It is a templated class.
  4. That is why we write “`vector<int>`” and “`vector<double>`” etc. to instantiate actual objects of vectors.
- For example, consider an algorithm to sort an array of objects.
  1. The fundamental concept of sorting data does not depend on a specific data type.
  2. We only need (i) an array of objects, and (ii) a comparison function to say that one object is “less than” another object.
  3. Then we can establish a ranking of the objects.

17 STL: Standard Templates Library

- The **STL (Standard Templates Library)** is a very important part of the standard C++ library.
- The STL employs the concept of generic programming.
- All of the classes in the STL are templated classes.
- The STL provides support for four broad categories of functionality.
  1. Containers.
  2. Algorithms.
  3. Iterators.
  4. Function objects (also called **functors**).
- A vector (dynamically resizeable array) is a container.
- Sorting is an example of an algorithm.
- Iterators are a generalization of the concept of pointers. The concept is too advanced and complicated for this course, to explain what the generalization is.
- A function object is an object which can be called and used as if it were a function. The concept is too advanced and complicated for this course. Function objects are also known as functors.