CS170#09.1

Class Templates

Vadim Surov

Outline

- Stack. Problems And Solutions
- Stack As A Template
- Non-type Template Arguments
- Multiple Template Arguments
- Default Template Arguments
- Class Template Instantiation

Stack

- A class template is very much like a function template
  - It is a generic way of defining a user-defined type (class)
  - This is another example of how C++ implements *generic programming*

- We will use a Stack class as an example
  - For simplicity, it will be array-based

Stack (contd)

```
class Stack {
public:
  Stack(int capacity)
    : items(new int[capacity]), count(0) { }
  ~Stack() { delete[] items; }
  void push(int item) { items[count++] = item; }
  int pop(void) { return items[--count]; }
  bool isEmpty(void) const { return count == 0; }
private:
  int* items;
  int count;
};
```

Stack (contd)

- Using the class:

```
int main(void) {
  const int SIZE = 5;
  Stack s(SIZE);
  for (int i = 0; i < SIZE; i++)
    s.push(i);
  while (!s.isEmpty())
    std::cout << s.pop() << std::endl;
  return 0;
}
```

- Output:

```
4
3
2
1
0
```

- There are some problems with our class though

Stack. Problems

- Dynamic memory allocation problems
- Need to define copy constructor and assignment operator to perform deep copy and assignment
- No error handling
  - E.g., `pop()` may be called when empty
  - Just adjust the functions to handle such cases (maybe use exceptions which will be introduced later)
- Size must be decided on creation

Stack. Problems

- Can only handle **int** data
- Create more classes?
  - e.g., Stack_int, Stack_float, etc.
  - But what about StopWatch or Weapon?
- Use **typedef**
  - In header file: **`typedef int`** `DATA`;
  - In class declaration: `DATA* items;`
  - Have to alter the header file

Stack. Solution

Stack as a Class Template

Stack As A Template

- Changing our Stack into a template class (and separating interface from implementation):

```
template<typename T>
class Stack {
public:
  Stack(int capacity);
  ~Stack();
  void push(const T& item);
  T pop(void);
  bool isEmpty(void) const;
private:
  T* items;
  int count;
};
```

Stack As A Template (contd)

- Implementation:

```
template<typename T>
Stack<T>::Stack(int capacity)
  : items(new T[capacity]), count(0) {}

template<typename T>
Stack<T>::~Stack() {
  delete[] items;
}

template<typename T>
void Stack<T>::push(const T& item) {
  items[count++] = item;
}
```

Stack As A Template (contd)

- Implementation:

```
template<typename T>
T Stack<T>::pop(void) {
  return items[--count];
}

template<typename T>
bool Stack<T>::isEmpty(void) const {
  return count == 0;
}
```

Stack As A Template (contd)

- Syntax:

```
template<typename T>
Stack<T>::
```

- The template **keyword** indicates that the class is a template class
- The type parameter name is placed in angle brackets after the class name
- The rest of the class is the same (except for replacing the type with the parameter name)

Stack As A Template (contd)

- You can still implement the methods within the class:

```
template<typename T>
class Stack {
public:
   Stack(int capacity)
     : items(new T[capacity]), count(0) { }
   ~Stack() { delete[] items; }
   void push(const T& item) { ... }
   T pop(void) { return items[--count]; }
   bool isEmpty(void) const { return count == 0;
}
private:
   T* items;
   int count;
};
```

Stack As A Template (contd)

- Using the template class:

```
int main(void)
{
  const int SIZE = 5;
  Stack<int> s(SIZE); // only change

  for (int i = 0; i < SIZE; i++)
    s.push(i);

  while (!s.isEmpty())
    std::cout << s.pop() << std::endl;
  return 0;
}
```

- Output:

```
4
3
2
1
0
```

Stack As A Template (contd)

- Creating a Stack of **double**:

```
int main(void)
{
  const int SIZE = 5;
  Stack<double> s(SIZE); // double

  for (int i = 0; i < SIZE; i++)
    s.push(i / 10.0); // push a double

  while (!s.isEmpty())
    std::cout << s.pop() << std::endl;
  return 0;
}
```

- Output:

```
0.4
0.3
0.2
0.1
0
```

Stack As A Template (contd)

- A Stack of StopWatch:

```
int main(void)
{
  const int SIZE = 5;
  Stack<StopWatch> s(SIZE);

  s.push(StopWatch(60));
  s.push(StopWatch(90));
  s.push(StopWatch(120));

  while (!s.isEmpty())
    std::cout << s.pop() << std::endl;
  return 0;
}
```

- Output:

```
00:02:00
00:01:30
00:01:00
```

Stack As A Template (contd)

- The template Stack class declaration does *not* generate any code
  - This is also true for class and structure declarations
- Code is *only* generated when an object of type Stack is instantiated
  - If no Stack is ever created, no code is generated

Stack As A Template (contd)

- Only one instance of the class is generated for *each* type

```
Stack<int> s1(10);
    // Code for Stack<int> generated

Stack<int> s2(20);
    // Nothing is generated

Stack<int> s3(30);
    // Nothing is generated

Stack<double> s4(3.14);
    // Code for Stack<double> generated
```

Stack As A Template (contd)

- You *must* specify the template arguments when you instantiate a template class
  - For template functions, the arguments can sometimes be deduced automatically by compiler
  - For template classes, nothing can be deduced
  - `Stack s; // A stack of what???`
- Template classes such as Stack are sometimes referred to as **containers**

Non-type Template Arguments

- Class templates can have non-type arguments
- Let's create an IntArray1 class that represents an array of **int**:

```
class IntArray1 {
public:
  IntArray1(int size)
     : size(size), items(new int[size]) {}
  ~IntArray1() { delete[] items; }
  // Other useful methods, e.g., operator[]
private:
  int size;
  int* items;
};
```

Non-type Template Args (contd)

- We use the class like this:

```
IntArray1 ar1(20);
IntArray1 ar2(30);
```

- Both ar1 and ar2 are of type IntArray1
- The arrays are dynamically allocated at runtime

Non-type Template Args (contd)

- Alternatively, we can use an **int** template argument for the size:

```
template<int S>
class IntArray2 {
public:
  IntArray2() : size(S) { }
  // No destructor required
  // Other useful methods, e.g., operator[]

private:
  int size;
  int items[S];
};
```

Non-type Template Args (contd)

- We use the class like this:
  ```
  IntArray2<20> ar3;
  IntArray2<30> ar4;
  ```
- ar3 and ar4 are of *different* types
  - Different code is generated for `IntArray2<20>` and `IntArray2<30>`
- The arrays are *statically* allocated at compile time
  - Avoids the overhead of dynamic memory allocation

Non-type Template Args (contd)

- Non-type template parameters represent values, not types
- These parameters can only be:
  - integral (including enumeration)
  - pointer (to object or to function)
  - reference (to object or to function)
- Therefore, they *cannot* be floating point type, **`struct`**, **`class`**, or **`void`**

Non-type Template Args (contd)

- They must also be constants (evaluated at compile time)

```
const int csize = 30; // Constant
int size = 20;        // Variable

IntArray2<30> ar1;           // OK - constant expression
IntArray2<csize> ar2;        // OK - constant expression
IntArray2<csize + 5> ar3;    // OK - constant expression
IntArray2<size> ar4;         // Error
IntArray2<size + 5> ar5;     // Error
```

- Note that ar1 and ar2 are of the same type, namely `IntArray2<30>`

Multiple Template Arguments

- Just like for function templates, you can have multiple class template arguments
- The arguments can be both type or non-type

Multiple Template Args (contd)

- Example (type-generic Array class):

```
template<typename T, int S>
class Array {
public:
  Array(void) : size(S) { }
  // No destructor required
  // Other useful methods, e.g., operator[]

private:
  int size;
  T items[S];
};
```

Multiple Template Args (contd)

- Using the class:

```
Array<int, 10> ar1;
Array<double, 20> ar2;
Array<StopWatch, 30> ar3;
Array<StopWatch*, 40> ar4;
```

Default Template Arguments

- Like function parameters, we can provide defaults for template parameters
  - It's true only for classes, not for functions!
- The same rules apply
  - For the parameter, give the = symbol followed by the default value
  - All default parameters must be at the end of the list of parameters

Default Template Args (contd)

- Example:

```
template <typename T = int, int S = 10>
class Array {
  // etc.
};
```

- Usage:

```
Array<double, 5> ar1; // Array<double, 5>
Array<double> ar2;    // Array<double, 10>
Array<> ar3;          // Array<int, 10>
Array<5> ar4; // Error (5 is not a type)
Array ar5;    // Error (Array is a template class)
```

Default Template Args (contd)

- Can we do this?

```
Array<Stack<int>, 20> ar;
```

- This translates to the following data member:

```
Stack<int> items[20];
```

- However, our Stack class has no default constructor, so this does not compile
- One solution (give default parameter):

```
Stack(int capacity = 10) ...
```

Default Template Args (contd)

- Pop quiz:

```
template <typename
 T = int, int S = 10>
class Foo {
public:
  Foo(int x = 0) { }

private:
  T items[S];
};
```

```
class A {
public:
  A() { }
};

class B {
public:
  B(int x) : x(x) { }
  operator int(void)
const
    { return x; }
private:
  int x;
};
```

Default Template Args (contd)

- Do these declarations compile?
  - `Foo<int, 5> foo1;`
  - `Foo foo2<5>;`
  - `Foo<int, B(5)> foo3;`
  - `Foo<A> foo4(B(5));`
  - `Foo<B, 5> foo5;`

Default Template Args (contd)

- Do these declarations compile?
  - `Foo<B, 5> foo6(5);`
  - `Foo<A(), 5> foo7;`
  - `Foo<> foo8;`
  - `Foo<5> foo9;`
  - `Foo<A, 5> foo10;`

Class Template Instantiation

- Just like for template functions, template classes are *implicitly* instantiated (only when needed)

```
Array<int, 10> ar; // implicit
```

- Unlike template functions, only the necessary class member functions are instantiated
  - Either when it is called; or
  - When its address is required (e.g., assigned to a pointer-to-function)

Class Template Inst. (contd)

- Example:

```
void f(Stack<int> &s); // no instantiations, declaration
int main(void)
{
  // instantiates Stack<int> (ctor and dtor)
  Stack<int> s1(10);
  // instantiates Stack<int>::ctor
  Stack<int> *s2 = new Stack<int>(10);

  f(s1); // no instantiations (by reference)
  delete s2; // instantiates Stack<int>::dtor
  sizeof(Stack<int>); // instantiates Stack<int>
                      //   (no methods)
  return 0;
}
```

Class Template Inst. (contd)

- Example:

```
void g(Stack<int> s) // instantiates Stack<int>
                        (copy ctor, dtor)
{
  s.Push(10); // instantiates Stack<int>::Push
}

void f(Stack<int> &s) // no instantiations (reference)
{
  // instantiates Stack<double> (ctor and dtor)
  Stack<double> t(5);
  Stack<int> *ps = &s; // no instantiations (pointer)
  ps->Push(10); // instantiates Stack<int>::Push
}
```

Class Template Inst. (contd)

- You can instantiate the entire class using *explicit template instantiation*:

```
template Stack<int>; // all methods
```

- You can also instantiate individual methods

```
template Stack<int>::Stack(int);
template Stack<int>::~Stack();
template Stack<int>::push(int);
```

Class Template Inst. (contd)

- This is useful when building libraries because non-instantiated template classes are not compiled into object files
- Most compilers allow implicit template instantiation to be switched off, e.g.,
  - g++ -fno-implicit-templates main.cpp

Summary

- Class templates are similar to function templates in that they are used to create classes that can take members of any data type
    - They tell the compiler how to generate a class
    - Code is only generated when instantiated
- Only one instance of the class is generated for each type

Summary

- Class template parameters can be non-type
  - Must be integral, pointer or reference
  - Must be constant
- Multiple parameters can be used
- Default values can be given for class template parameters

Summary

- Class template methods are only instantiated when required
  - When the method is called
- Use *explicit instantiation* to instantiate a class or class method without having to instantiate an object of that class