

Programming in C and C++

7. Exceptions — Templates

Dr. Anil Madhavapeddy

University of Cambridge
(based on previous years –
thanks to Alan Mycroft, Alastair Beresford and Andrew Moore)

Michaelmas Term 2014–2015

Exceptions

- ▶ Some code (e.g. a library module) may detect an error but not know what to do about it; other code (e.g. a user module) may know how to handle it
- ▶ C++ provides exceptions to allow an error to be communicated
- ▶ In C++ terminology, one portion of code throws an exception; another portion catches it.
- ▶ If an exception is thrown, the call stack is unwound until a function is found which catches the exception
- ▶ If an exception is not caught, the program terminates

Throwing exceptions

- ▶ Exceptions in C++ are just normal values, matched by type
- ▶ A class is often used to define a particular error type:

```
class MyError {};
```

- ▶ An instance of this can then be thrown, caught and possibly re-thrown:

```
1 void f() { ... throw MyError(); ... }
2 ...
3     try {
4         f();
5     }
6     catch (MyError) {
7         //handle error
8         throw; //re-throw error
9     }
```

Conveying information

- ▶ The “thrown” type can carry information:

```
1 struct MyError {
2     int errorcode;
3     MyError(i):errorcode(i) {}
4 };
5
6 void f() { ... throw MyError(5); ... }
7
8 try {
9     f();
10 }
11 catch (MyError x) {
12     //handle error (x.errorcode has the value 5)
13     ...
14 }
```

Handling multiple errors

- ▶ Multiple catch blocks can be used to catch different errors:

```
1 try {  
2     ...  
3 }  
4 catch (MyError x) {  
5     //handle MyError  
6 }  
7 catch (YourError x) {  
8     //handle YourError  
9 }
```

- ▶ Every exception will be caught with `catch(...)`
- ▶ Class hierarchies can be used to express exceptions:

```
1 #include <iostream>
2
3 struct SomeError {virtual void print() = 0;};
4 struct ThisError : public SomeError {
5     virtual void print() {
6         std::cout << "This Error" << std::endl;
7     }
8 };
9 struct ThatError : public SomeError {
10     virtual void print() {
11         std::cout << "That Error" << std::endl;
12     }
13 };
14 int main() {
15     try { throw ThisError(); }
16     catch (SomeError& e) { //reference, not value
17         e.print();
18     }
19     return 0;
20 }
```

Exceptions and local variables

- ▶ When an exception is thrown, the stack is unwound
- ▶ The destructors of any local variables are called as this process continues
- ▶ Therefore it is good C++ design practice to wrap any locks, open file handles, heap memory etc., inside stack-allocated object(s), with constructors doing allocation and destructors doing deallocation. This design pattern is analogous to Java's try-finally, and is often referred to as "RAII: Resource Allocation is Initialisation".

Templates

- ▶ Templates support meta-programming, where code can be evaluated at compile-time rather than run-time
- ▶ Templates support generic programming by allowing types to be parameters in a program
- ▶ Generic programming means we can write one set of algorithms and one set of data structures to work with objects of any type
- ▶ We can achieve some of this flexibility in C, by casting everything to `void *` (e.g. `sort` routine presented earlier)
- ▶ The C++ Standard Template Library (STL) makes extensive use of templates

An example: a stack

- ▶ The stack data structure is a useful data abstraction concept for objects of many different types
- ▶ In one program, we might like to store a stack of `ints`
- ▶ In another, a stack of `NetworkHeader` objects
- ▶ Templates allow us to write a single generic stack implementation for an unspecified type `T`
- ▶ What functionality would we like a stack to have?
 - ▶ `bool isEmpty();`
 - ▶ `void push(T item);`
 - ▶ `T pop();`
 - ▶ ...
- ▶ Many of these operations depend on the type `T`

Creating a stack template

- ▶ A class template is defined as:

```
1 template<class T> class Stack {  
2     ...  
3 }
```

- ▶ Where `class T` can be any C++ type (e.g. `int`)
- ▶ When we wish to create an instance of a `Stack` (say to store `ints`) then we must specify the type of `T` in the declaration and definition of the object: `Stack<int> intstack;`
- ▶ We can then use the object as normal: `intstack.push(3);`
- ▶ So, how do we implement `Stack`?
 - ▶ Write `T` whenever you would normally use a concrete type

```
1  template<class T> class Stack {
2
3      struct Item { //class with all public members
4          T val;
5          Item* next;
6          Item(T v) : val(v), next(0) {}
7      };
8
9      Item* head;
10
11     Stack(const Stack& s) {}           //private
12     Stack& operator=(const Stack& s) {} //
13
14 public:
15     Stack() : head(0) {}
16     ~Stack();           // should generally be virtual
17     T pop();
18     void push(T val);
19     void append(T val);
20 };
```

```
1 #include "example16.hh"
2
3 template<class T> void Stack<T>::append(T val) {
4     Item **pp = &head;
5     while(*pp) {pp = &((*pp)->next);}
6     *pp = new Item(val);
7 }
8
9 //Complete these as an exercise
10 template<class T> void Stack<T>::push(T) {/* ... */}
11 template<class T> T Stack<T>::pop() {/* ... */}
12 template<class T> Stack<T>::~~Stack() {/* ... */}
13
14 int main() {
15     Stack<char> s;
16     s.push('a'), s.append('b'), s.pop();
17 }
```

Template details

- ▶ A template parameter can take an integer value instead of a type:
`template<int i> class Buf { int b[i]; ... };`
- ▶ A template can take several parameters:
`template<class T,int i> class Buf { T b[i]; ... };`
- ▶ A template can even use one template parameter in the definition of a subsequent parameter:
`template<class T, T val> class A { ... };`
- ▶ A templated class is not type checked until the template is instantiated:
`template<class T> class B {const static T a=3;};`
 - ▶ `B<int> b;` is fine, but what about `B<B<int> > bi;`?
- ▶ Template definitions often need to go in a header file, since the compiler needs the source to instantiate an object

Default parameters

- ▶ Template parameters may be given default values

```
1 template <class T,int i=128> struct Buffer{
2     T buf[i];
3 };
4
5 int main() {
6     Buffer<int> B; //i=128
7     Buffer<int,256> C;
8 }
```

Specialisation

- ▶ The `class T` template parameter will accept any type `T`
- ▶ We can define a specialisation for a particular type as well (effectively type comparison at compile-time)

```
1 #include <iostream>
2 class A {};
3
4 template<class T> struct B {
5     void print() { std::cout << "General" << std::endl;}
6 };
7 template<> struct B<A> {
8     void print() { std::cout << "Special" << std::endl;}
9 };
10
11 int main() {
12     B<A> b1;
13     B<int> b2;
14     b1.print(); //Special
15     b2.print(); //General
16 }
```

Templated functions

- ▶ A function definition can also be specified as a template; for example:

```
1 template<class T> void sort(T a[],  
2                             const unsigned int& len);
```

- ▶ The type of the template is inferred from the argument types:

```
int a[] = {2,1,3}; sort(a,3);  $\implies$  T is an int
```

- ▶ The type can also be expressed explicitly:

```
sort<int>(a,3)
```

- ▶ There is no such type inference for templated classes

- ▶ Using templates in this way enables:

- ▶ better type checking than using `void *`
- ▶ potentially faster code (no function pointers in vtables)
- ▶ larger binaries if `sort()` is used with data of many different types


```
1 #include <iostream>
2
3 template<class T> void sort(T a[], const unsigned int& len) {
4     T tmp;
5     for(unsigned int i=0;i<len-1;i++)
6         for(unsigned int j=0;j<len-1-i;j++)
7             if (a[j] > a[j+1]) //type T must support "operator>"
8                 tmp = a[j], a[j] = a[j+1], a[j+1] = tmp;
9 }
10
11 int main() {
12     const unsigned int len = 5;
13     int a[len] = {1,4,3,2,5};
14     float f[len] = {3.14,2.72,2.54,1.62,1.41};
15
16     sort(a,len), sort(f,len);
17     for(unsigned int i=0; i<len; i++)
18         std::cout << a[i] << "\t" << f[i] << std::endl;
19 }
```

Overloading templated functions

- ▶ Templated functions can be overloaded with templated and non-templated functions
- ▶ Resolving an overloaded function call uses the “most specialised” function call
- ▶ If this is ambiguous, then an error is given, and the programmer must fix by:
 - ▶ being explicit with template parameters (e.g. `sort<int>(…)`)
 - ▶ re-writing definitions of overloaded functions
- ▶ Overloading templated functions enables meta-programming:

Meta-programming example

```
1 #include <iostream>
2
3 template<unsigned int N>
4 struct fact {
5     static const long int value = N * fact<N-1>::value;
6     char v[value]; // value is computed at compile time!
7 };
8
9 template<>
10 struct fact<0> {
11     static const long int value = 1;
12 };
13
14 struct fact<7> foo; // struct with char v[5040] and a const.
15
16 int main() {
17     std::cout << sizeof(foo) << ", " << foo.value << std::endl;
18 }
```

Templates are a Turing-complete compile-time programming language.

Exercises

1. Provide an implementation for:

```
template<class T> T Stack<T>::pop(); and  
template<class T> Stack<T>::~~Stack();
```

2. Provide an implementation for:

```
Stack(const Stack& s); and  
Stack& operator=(const Stack& s);
```

3. Using meta programming, write a templated class `prime`, which evaluates whether a literal integer constant (e.g. 7) is prime or not at compile time.
4. How can you be sure that your implementation of class `prime` has been evaluated at compile time?