Object Oriented Software Design
Exceptions and Templates

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

March 27, 2013

Outline

Try/catch

- An exception object is *thrown* by the programmer in case of an error condition
- An exception object can be caught inside a try/catch block

```
try {
   //
   // this code can generate exceptions
   //
} catch (ExcType1& e1) {
   // all exceptions of ExcType1 are handled here
}
```

- If the exception is not caught at the level where the function call has been performed, it is automatically forwarded to the upper layer
  - Until it finds a proper try/catch block that *cathes* it
  - or until there is no upper layer (in which case, the program is aborted)

More catches

- It is possible to put more catch blocks in sequence
- they will be processed in order, the first one that catches the exception is the last one to execute

```
try {
   //
   // this code can generate exceptions
   //
} catch (ExcType1&e1) {
   // all exceptions of ExcType1
} catch (ExcType2 &e2) {
  // all exceptions of ExcType2
} catch (...) {
  // every exception
}
```

Re-throwing

- It is possible to re-throw the same exception that has been caught to the upper layers

```
catch(...) {
  cout << "an exception was thrown" << endl;
  // Deallocate your resource here, and then rethrow
  throw;
}
```


Exception specification

- It is possible to specify which exceptions a function might throw, by listing them after the function prototype
- Exceptions are part of the interface!

```
void f(int a) throw(Exc1, Exc2, Exc3);
void g();
void h() throw();
```

- f() can **only** throw exception Exc1, Exc2 or Exc3
- g() can throw **any** exception
- h() **does not** throw any exception

Terminate

- In case of abort, the C++ run-time will call the terminate(), which calls abort()
  - It is possible to change this behaviour

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

void terminator() {
  cout << "I'll be back!" << endl; exit(0);
}
void (*old_terminate)() = set_terminate(terminator);

class Botch {
public:
  class Fruit {};
  void f() {
    cout << "Botch::f()" << endl;
    throw Fruit();
  }
  ~Botch() { throw 'c'; }
};

int main() {
  try {
    Botch b; b.f();
  } catch(...) {
    cout << "inside catch(...)" << endl;
  }
}
```


Resource management

- When writing code with exceptions, it's particularly important that you always ask, "If an exception occurs, will my resources be properly cleaned up?"
- Most of the time you're fairly safe,
- but in constructors there's a particular problem:
  - if an exception is thrown before a constructor is completed, the associated destructor will not be called for that object.
  - Thus, you must be especially diligent while writing your constructor.
- The difficulty is in allocating resources in constructors.
  - If an exception occurs in the constructor, the destructor doesn't get a chance to deallocate the resource.
  - see exceptions/rawp.cpp

How to avoid the problem

- To prevent such resource leaks, you must guard against these "raw" resource allocations in one of two ways:
  - You can catch exceptions inside the constructor and then release the resources
  - You can place the allocations inside an object's constructor, and you can place the deallocations inside an object's destructor.
- The last technique is called Resource Acquisition Is Initialization (RAII for short) because it equates resource control with object lifetime.
- Example: exception_wrap.cpp

Outline

Containers

- Consider the problem of providing a generic container of objects
- Example
  - We designed and developed a Stack class container
  - it is an object that *contains* other objects, and provides operations for inserting, extracting, finding object, and visiting them in a certain order
  - Our stack class contains integers
  - However, the code is generic enough and depends only in minimal part from the fact that it contains integers
- Problem:
  - How to extend it to contains other types of objects?
  - for example, Shapes

Use inheritance

- OO languages that do not have templates, use inheritance for implementing such containers
  - For example, in Smalltalk (and in Java), all classes derive from a common ancestor: Object
  - The containers will contain pointers to Object
  - however, the type is lost when you insert an object in a container
  - The user has to perform an appropriate downcast to get back to the original type
- We can do something similar in C++, by using multiple interface inheritance

Templates

- Templates are used for generic programming
- The general idea is: what we want to reuse is not only the abstract concept, but **the code itself**
- with templates we reuse algorithms by making them general
- As an example, consider the code needed to swap two objects of the same type (i.e. two pointers)

```
void swap(int &a, int &b)
{
    int tmp;
    tmp = a;
    a = b;
    b = tmp;
}
...
int x=5, y=8;
swap(x, y);
```

- Can we make it generic?

How does it work?

- The template mechanism resembles the macro mechanism in C
  - We can do the same in C by using pre-processing macros:

```
#define swap(type, a, b) { type tmp; tmp=a; a=b; b=tmp; }
...
int x = 5; int y = 8;

swap(int, x, y);
```

- in this case, the C preprocessor substitutes the code
  - it works only if the programmer knows what he is doing
- The template mechanism does something similar
  - but the compiler performs all necessary type checking

Code duplicates

- The compiler will instantiate a version of `swap()` with integer as a internal type
- if you call `swap()` with a different type, the compiler will generate a new version
  - Only when a template is instantiated, the code is generated
    - If we do not use `swap()`, the code is never generated, even if we include it!
    - if there is some error in `swap()`, the compiler will never find it until it tries to generate the code
- Looking from a different point of view:
  - the template mechanism is like cut&paste done by the compiler at compiling time

Parameters

- A template can have any number of parameters
- A parameter can be:
  - a class, or any predefined type
  - a function
  - a constant value (a number, a pointer, etc.)

```
template<T, int sz>
class Buffer {
    T v[sz];
    int size_;
public:
    Buffer() : size_(0) {}
};
...
Buffer<char, 127> cbuf;
Buffer<Record, 8> rbuf;
int x = 16;
Buffer<char, x> ebuf; // error!
```

Default values

- Some parameter can have default value

```
template<class T, class Allocator = allocator<T> >
class vector;
```


Generalizing Stack

- Now, let's go back to our Stack class, and generalize it to contain any type of object

```
template<class T>
class Stack {
    class Elem {
    public:
        T data_;
        ...
    };
public:
    class Iterator {
        friend class Stack<T>;
        ...
    public:
        inline T operator*() const
        {
            ...
        }
        ...
    };
```

```
    Stack() : head_(0), size_(0)
    {}
    ~Stack() {
        ...
    }

    void push(const T &a) {...}
    T pop() {...}
    ...
};
```

Inlines

- It is possible to define the members of a template class later on
  - Must be preceded by keyword template

```
template<class T>
class Array {
    enum { size = 100 };
    T A[size];
public:
    T& operator[](int index);
};

template<class T>
T& Array<T>::operator[](int index) {
    require(index >= 0 && index < size, "Index out of range");
    return A[index];
}

int main() {
    Array<float> fa;
    fa[0] = 1.414;
}
```


Template instantiation

- The code for the template is not instantiated until the template is used
  - It works similarly to in-lines
- The template code must go in the header file
  - Otherwise, the template is not seen by the compiler which does not know how to translate it

Standard Template Library

- The STL is provided with the compiler
- It contains generic code (templates) with
  - containers (vector, list, deque, map, set)
  - algorithms (sort, foreach, etc.)
  - I/O streams (cout, cin, fstreams, etc.)
  - string
- Recently, with the new standard, many more features have been added (will see a selection later in the course)

Vector

- Vector is the generalisation of the C array

```
vector<int> v_int;
vector<string> v_s;
vector<int> v_int2;

v_int.push_back(5);
cout << v_int.size() << endl;

for (int i=0; i<10;i++)
    v_int.push_back(i);

cout << v_int[0] << endl;

cout << v_int[12] << endl;
cout << v_int.at(12) << endl;

v_int2 = v_int;
```

an array of integers

a vector of strings

inserts an element in the vector

prints 1

prints the first element

out of range: undefined behaviour

out of range: raises exception

copies the entire vector

Vector of objects

- Vector requires the following basic properties of the template class
  - Copy constructor; (otherwise you cannot insert elements)
  - Assignment operator; (otherwise you cannot return an object)
- It is possible to pre-allocate space for the vector;
  - This is used to avoid excessive allocation overhead when we have an idea of the size we need

```
vector<MyClass> v(10); // reserves 10 elements
```

- However, in this case `MyClass` must declare a default constructor

Iterators

- Iterators are a generic way to access elements in a container, according to a predefined order
- The iterator is usually a class provided by the container itself
- It can be seen as a *pointer* to the elements of the container
  - `begin()` returns an iterator to the first element
  - `end()` returns the iterator pointing *beyond* the last element of the array
  - it is possible to use `++` and `-` to increment/decrement the iterator (i.e. move to the next/previous element)
  - it is possible to access the *pointed element* by using the dereferencing `operator*`

```
vector<int> v;
vector<int>::iterator i;
...
for (i = v.begin(); i!=v.end(); i++) cout << *i;
```

Iterators

iterator-example.cpp

```
int main()
{
    int a[4] = {2, 4, 6, 8};
    vector<int> v = {2, 4, 6, 8};

    // visit the container with indexes
    for (int i=0; i<4; i++) cout << a[i];
    cout << endl;
    for (int i=0; i<4; i++) cout << v[i];
    cout << endl;

    // visit the container with pointers/iterators
    for (int *p=a; p!=&a[4]; p++) cout << *p;
    cout << endl;
    for (vector<int>::iterator q=v.begin();
         q != v.end(); q++) cout << *q;
    cout << endl;

    vector<int>::iterator q = v.end();
    do {
        q--; cout << *q;
    } while (q != v.begin());
    cout << endl;
```


Why iterators

- Iterators are available for **all** containers in the standard library
- They represent a simple and uniform way to visit a container
- Many template functions and member functions accept iterators parameters

./examples/06.exceptions-templates-examples/iterator-example2.cpp

- Exercise: generalise function print, so that it can print the content of vectors of **any** type
- Solution:
  ./examples/06.exceptions-templates-examples/iterator-example3.cpp

Lists

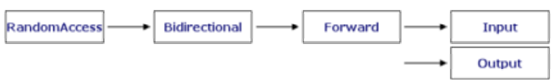
- The STL also provides the simple linked list we have seen in the course
- In the STL, the template parameter indicates the data type

```
list<int> lst;
for (int i=0; i<10; i++)
    lst.push_back(i);

// going through all elements
list<int>::iterator i = lst.begin();
int sum = 0;
while (i!=lst.end()) {
    sum += *i;
    i++;
}
```


Iterator types

- There are five types of iterators, depending on the functionality they provide:



- The difference consists in the type of operations that are supported:
  - all types support operator `++` and `*`
  - `input` supports copy construction and copy, operator `->`, equality `==` and inequality `!=`
  - `output` supports assignment as lvalue (to the left of an assignment operator)
  - `forward` is as `input` and `output`, but also supports default constructor
  - `bidirectional` is as `forward`, but it also supports operator `-`
  - `random` is as `bidirectional`, but also supports operators like `+`, `-`, `+=`, `-=`, comparison (<, <=, >=, >), offset `[]`

Iterator types

category				characteristic	valid expressions
all categories				Can be copied and copy-constructed	X b(a); b = a;
				Can be incremented	++a a++ *a++
Random Access	Bidirectional	Forward	Input	Accepts equality/inequality comparisons	a == b a != b
				Can be dereferenced as an *rvalue*	*a a->m
			Output	Can be dereferenced to be the left side of an assignment operation	*a = t *a++ = t
				Can be default-constructed	X a; X()
				Can be decremented	--a a-- *a--
				Supports arithmetic operators + and -	a + n n + a a - n a - b
				Supports inequality comparisons (<, >, <= and >=) between iterators	a < b a > b a <= b a >= b
				Supports compound assignment operations += and -=	a += n a -= n
				Supports offset dereference operator ([])	a[n]

Associative arrays

- An associative array generalize the concept of array
- Two subtypes: sets and maps
  - set<key> and multiset<key> contain ordered sets of objects
  - in set<key> the key must be unique
  - in multiset<key>, the same key can be inserted several times
- map<key,value> and multimap<key,value> contains pairs <key,value>, where key is the “index” in the array
  - in map<key, value>, each different key must be associated one unique value
  - map<key, value>, several values can be associated to the same key

Map example

```
int main()
{
    map<string, int> age;

    age["Peppe"] = 40;
    age["Roberto"] = 25;
    age["Giovanna"] = 30;

    pair<string, int> elem = {"Pippo", 32};

    cout << elem.first << " = " << elem.second << endl;

    age.insert(elem);

    map<string, int>::iterator i;
    for (i = age.begin(); i != age.end(); i++)
        cout << i->first << " = " << i->second << endl;
    ...
}
```

See map_example.cpp

The typename keyword

- The typename keyword is needed when we want to specify that an identifier is a type

```
template<class T> class X {
  typename T::id i;    // Without typename, it is an error:
public:
  void f() { i.g(); }
};

class Y {
public:
  class id {
  public:
    void g() {}
  };
};

int main() {
  X<Y> xy;
  xy.f();
}
```

General rule

- if a type referred to inside template code is qualified by a template type parameter, you must use the typename keyword as a prefix,
- unless it appears in a base class specification or initializer list in the same scope (in which case you must not).

Usage

- The typical example of usage is for iterators

```
template<class T, template<class U, class = allocator<U> >
         class Seq>
void printSeq(Seq<T>& seq) {
  for(typename Seq<T>::iterator b = seq.begin();
       b != seq.end();)
    cout << *b++ << endl;
}

int main() {
  // Process a vector
  vector<int> v;
  v.push_back(1);
  v.push_back(2);
  printSeq(v);
  // Process a list
  list<int> lst;
  lst.push_back(3);
  lst.push_back(4);
  printSeq(lst);
}
```

Making a member template

- An example for the complex class

```
template<typename T> class complex {
public:
  template<class X> complex(const complex<X>&);
  ...
};

complex<float> z(1, 2);
complex<double> w(z);
```

- In the declaration of w, the complex template parameter T is double and X is float. Member templates make this kind of flexible conversion easy.

Another example

```
int data[5] = { 1, 2, 3, 4, 5 };
  vector<int> v1(data, data+5);
  vector<double> v2(v1.begin(), v1.end());
```

- As long as the elements in v1 are assignment-compatible with the elements in v2 (as double and int are here), all is well.
- The vector class template has the following member template constructor:

```
template<class InputIterator>
vector(InputIterator first, InputIterator last,
       const Allocator& = Allocator());
```

- InputIterator is interpreted as vector<int>::iterator

Another example

```
template<class T> class Outer {
public:
  template<class R> class Inner {
  public:
    void f();
  };
};

template<class T> template<class R>
void Outer<T>::Inner<R>::f() {
  cout << "Outer == " << typeid(T).name() << endl;
  cout << "Inner == " << typeid(R).name() << endl;
  cout << "Full Inner == " << typeid(*this).name() << endl;
}

int main() {
  Outer<int>::Inner<bool> inner;
  inner.f();
}
```


Restrictions

- Member template functions cannot be declared virtual.
  - Current compiler technology expects to be able to determine the size of a class's virtual function table when the class is parsed.
  - Allowing virtual member template functions would require knowing all calls to such member functions everywhere in the program ahead of time.
  - This is not feasible, especially for multi-file projects.

Function templates

- The standard template library defines many function templates in algorithm
  - sort, find, accumulate, fill, binary_search, copy, etc.
- An example:

```
#include <algorithm>
...
int i, j;
...
int z = min<int>(i, j);
```

- Type can be deducted by the compiler
- But the compiler is smart up to a certain limit . . .

```
int z = min(x, j); // x is a double, error, not the same types

int z = min<double>(x, j); // this one works fine
```