SFU CMPT-212 2008-1 Topic: Exceptions

Ján Maňuch

E-mail: jmanuch@sfu.ca

Friday 4th April, 2008

2008 Ján Maňuch

Exceptions

- a mechanism for dealing with errors during runtime:
- *throwing an exception*:

```
1 throw expression_of_any_type;
```

Examples:

```
1 throw 10;
2 throw "Division by zero.";
3 throw myclass(10,5.4);
```

- *catching an exception*: has two components:
  - **a try block** — a part of the program in which we are testing for exceptions
  - **a handler** — follows the try block, handles a particular exception

```
1  try {
2    // code which can throw exceptions
3  }
4  catch (const char * s) {
5    // what to do if an exception
6    // of type char* was thrown
7  }
8  catch (Int &obj) {
9    // what to do if an exception
10   // of type Int was thrown
11 }
12 catch (...) {
13   // what to do if any other
14   // exception was thrown
15 }
```

What is it good for?

- consider class `intArray` again
- let's add range checking
- what to do when the user of the class tries to access element which is out of range?
  - terminate the program
  - return some special value, for example `INT_MAX`, and hope that a user of the class will check for this value
  - *better solution:* throw an **exception**:
    if a user of the class catches the exceptions, fine!
    if not, an *uncaught exception* will force the program to terminate
- let's look at the details of the mechanism first

What happens when an exception is thrown?

- the program stops normal execution, and starts to search for the nearest *handler* catching the exception ("unwinding stack")
- *if found,* executes the handler and continues with the code which follows the try block and its handlers
- *if not found (**uncaught exception**),* calls function `terminate()` which, by default, forces the program to terminate

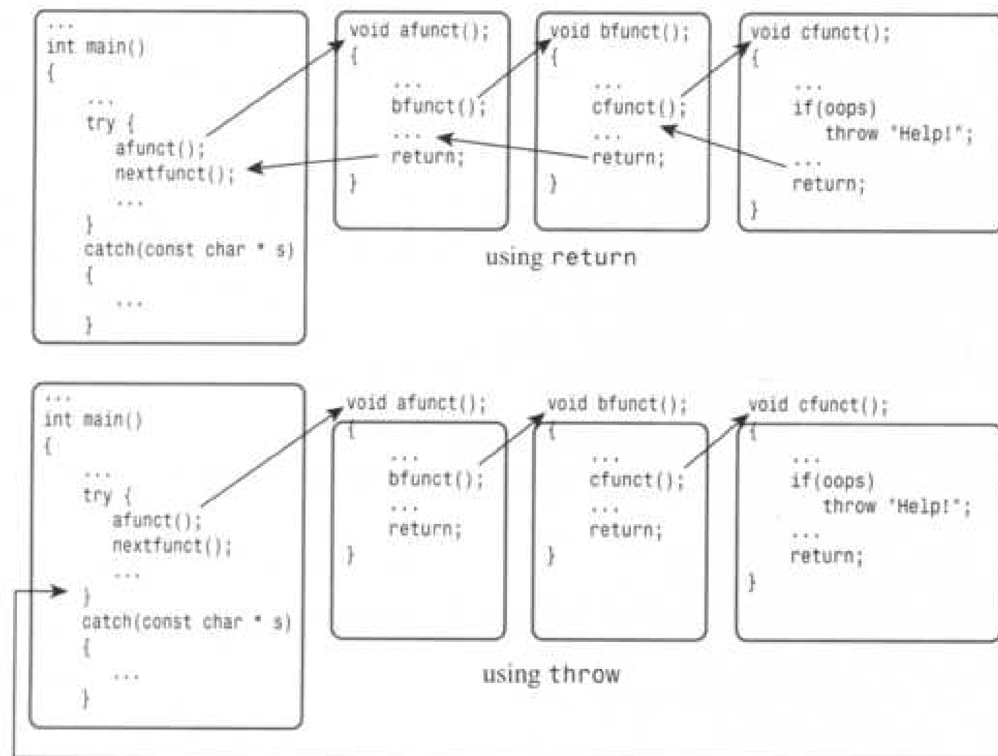
Example: exception1.cpp

```
1  void f(int n) {
2    if (n>2)
3      throw 12; // exception of type int
4    if (n>1)
5      throw 13.3; // exception of type double
6    cout <<"No exception thrown"<<endl;
7  }
```

```
1  int main() {
2    int number;
3    try {
4      cin >> number;
5      f(number);
6      cout <<"f() executed correctly."<<endl;
7    }
8    catch (int x) { // catch exception of type int
9      cout <<"int exception: "<<x<<endl;
10   }
11   catch (double x) { // catch exception of type double
12     cout <<"double exception: "<<x<<endl;
13   }
14   // after normal execution of try block or after
15   // handling exception program control continues here:
16   cout <<"after try block"<<endl;
17   return 0;
18 }
```

Remarks:

- `throw` acts in a similar way as `return` — if the exception is not caught directly in the function where it was thrown, it will terminate the execution of the function
- this behavior is recursive, until the handler is found (or `terminate()` is called)



[Prata]

Example: exception2.cpp

```
1  void f() {
2    throw 10;
3    cout <<"f() - ok"<<endl;
4  }
5  void g() {
6    try { f(); }
7    catch (const char* s) { cout <<s<<endl; }
8    cout <<"g() - ok"<<endl;
9  }
10 void h() {
11   try { g(); }
12   catch (int x) {
13     cout <<"Exception caught: "<<x<<endl;
14   }
15   cout <<"h() - ok"<<endl;
16 }
```

Is it safe?

- can throwing an exception cause a memory leak?
- **automatic variables**: “No”
  the compiler will take care of all automatic variables when the program control backs up from the functions which didn’t handle the thrown exception
  Example: exception3.cpp
- **dynamic variables**: “Yes”
  freeing dynamically allocated space is up to the programmer
  Example: exception4.cpp
- *what can we do?*
  - function `g()` doesn’t know how to handle the exception `int`
  - but it should call `delete` on `b` when it happens

- *Solution 1:*
  - let's catch the exception in `g()`,
  - free the allocated memory, and
  - rethrow the exception to be caught by a function higher in the stack

  Example: exception5.cpp

- **Notes:**
  - `catch(...)` catches any exception
  - `throw;` rethrows the same exception as caught in the current handler

- *Solution 2:*
  - use a **smart** pointer: template `auto_ptr`
  - defined in the header file `memory` in the namespace `std`
  - the smart pointer is an automatic variable, so its destructor is called automatically
  - and the destructor will call `delete` on the "owned pointer"
    Example: exception6.cpp
  - the unary constructor is declared as `explicit`, hence the object of class `auto_ptr` can be constructed only *explicitly*:

```
1 auto_ptr<double> pd(new double); // ok
2 auto_ptr<int> pi=new int;         // not allowed
```

  - *Question:* What happens if we assign one smart pointer to another? Would it cause a crash?

- `auto_ptr` is a smart pointer with the *ownership concept*:
  - always only one smart pointer points to the object; after assignment, the pointer is passed from the smart pointer on the right hand side to the smart pointer on the left hand side of the assignment
- the assignment operator and the copy constructor **transfer** the *ownership*:

```
1 auto_ptr<double> pd(new double);
2 cout <<pd.get() <<endl;          // prints the address
3 auto_ptr<double> pd_copy=pd;
4 cout <<pd.get() <<endl;          // prints NULL
5 cout <<pd_copy.get() <<endl;     // prints the address
6 *pd=10.0;                        // runtime error
```

- other concepts: deep copy, reference counting (*supplemental example:* countptr.h)

Qualifying the function headers

- we can indicate what kind of exception a function throws in the prototype of the function:

```
1 void f() throw(int);
2   // function can throw int exception
3 void g() throw(double, const char*);
4   // function can throw either double
5   // or const char* exception
6 void h() throw();
7   // function is not throwing any exception
```

- *what happens if a function throws an exception not specified in the prototype?*

  "**unexpected exception**" — program calls function `unexpected()`, which, by default, calls `terminate()` (use `set_unexpected` to call a different function)

Exceptions are usually objects

- throwing an `int` or an `double` exception is not very informative, it's difficult to deduce what caused the exception
- usually, the exceptions are objects of some special design classes:
  - the type of the object tells us what kind of error caused the exception
  - object can contain additional information about the error

Example:

- in the `intArray` class, we could throw an exception of type class `OutOfRange`
- an exception object could contain information about the minimal and maximal allowed indexes, and about the offending index which caused the exception

```
1  class OutOfRange {
2    int minIndex,maxIndex,wrongIndex;
3   public:
4    OutOfRange(int mini, int maxi, int i)
5     : minIndex(mini), maxIndex(maxi), wrongIndex(i) {}
6    void report() {
7      cout <<"The index "<<wrongIndex<<" is out of range ["
8           <<minIndex<<".."<<maxIndex<<"]."<<endl; }
9  };
10
11 int intArray::get(long index) {
12   if (index<0 || index>=size)
13     throw OutOfRange(0,size-1,index);
14   return data[index];
15 }
```

```
1 intArray array(5);
2 try {
3   cout <<array.get(10);
4 }
5 catch (OutOfRange &exc) {
6   exc.report();
7 }
```

Example: array.cpp

Remark:

- after `throw` the exception object is copied (by value) to another location, and then the `catch` block is called with this another location
- hence it's ok to throw a temporary object `OutOfRange(0,size-1,index)`
- `catch` block should use passing arguments by reference, otherwise exception object is copied twice: *Example:* twice.cpp

The exception class

- all predefined exception classes are derived from the class `exception` (defined in the header file `<exception>`):

```
1 class exception {
2  public:
3   virtual const char *what() const throw();
4   // returns description of exception
5 };
```

- hence, any standard exception class has this function defined
- when compiler searches a handler for an exception, it can use upcasting; therefore, we can catch any standard exception like this:

```
1 try { code throwing exceptions }
2 catch (exception &e) // use reference here!
3                      // otherwise no upcasting
4 { cout <<"Exception: "<<e.what() <<endl;
5   exit(1); }
```

Useful derived classes

- `out_of_range`: thrown by `vector`'s `at` method
- `bad_alloc`: thrown by operator `new` if it fails to allocate memory (assuming the header file `<new>` is included)
- if `<new>` is not included, the `new` operator just returns `NULL` when it cannot allocate memory

Example: new.cpp

```
1  #include <new>
2  ...
3  try {
4    int *p=new int[10000];
5  }
6  catch (bad_alloc &exc) {
7    cout <<"Exception: "<<exc.what()<<endl;
8    exit(1);
9  }
```

Sample questions

- What do you have to do to throw an exception? What types of exception can you throw?
- What do you have to do to catch an exception?

- What is the output of the following program?

```
 1  void f() {
 2    cout <<1;
 3    throw -1;
 4    cout <<2;
 5  }
 6  void g() {
 7    cout <<3;
 8    f();
 9    cout <<4;
10  }
11  int main() {
12    try {
13      cout <<5;
14      g();
15      cout <<6;
16    }
17    catch (int i) {
18      cout <<7;
19    }
20    cout <<8;
21    return 0;
22  }
```

What would be the output if we add `throw()` at the end of function heading of `g()`?

- What is the output of the following fragment of code:

```
1  class A {
2   public:
3     A() { cout <<1; }
4     ~A() { cout <<2; }
5  };
6  ...
7  try {
8     A x;
9     throw 3.2;
10 }
11 catch (double d) { cout <<d; }
```

- What is the purpose of the statement

```
1  throw;
```


2008 Ján Maňuch