C++

Exceptions

Introduction

- Traditionally error handling was done by returning from a function with a special value (an error code or a sentinel)
- This is probably the most frequent way of doing error handling in C
- This can make the code confusing and difficult to read
- C++ is providing built-in features in the language to raise and handle exceptions
  - This allows to separate the normal code from the error handling

Exception

- An **exception** is a runtime problem occurring in a program
- Examples of exception are: division by 0, array out of bounds, etc
- When such an event occurs, we say that an exception is **raised**
- Processing an exception is called **exception handling**

Exception in C++

- The C++ exception handling system is made of three parts: try blocks, catch blocks and throw statements.

```
try {
  // code that throws an exception
} catch (type1 e1) {
  // handling of exception of type type1
} catch (type2 e2) {
  // handling of exception of type type2
}
```

Try block

- To declare a try block: write the keyword try, then surround the code in curly braces.

```
try {
  cout << “inside a try block” << endl;
}
```

- Inside a try block code executes as normal and jumps to the code directly following the try block if no problem occurred

Try block

- If inside a try block, the program run into a situation from which it can not recover:
  - An exception is thrown (or raised) using the throw keyword
  - This exception is then thrown to the nearest matching exception handler (catch clause)
- Try blocks can be nested

Throw statement

- A throw statement is used to raise an exception
- The syntax is: **throw object** where the type of the object gives the type of the exception to be thrown
- Examples:
  - throw 0; // throws an int
  - throw new vector<int>; // throws a vector<int> *
  - throw 2.0; // throws a double

Catch clause

- Catch clauses are containing the code that will handle the exception
- It must immediately follow a try block
- The general syntax is:
  catch (ParameterType e) {
  // exception handler
  }
- In “catch (ParameterType e)” exceptions of type “ParameterType” not yet handled will be handled by this catch clause
- Catch clauses are specialized for a single type, it is legal to have cascading catch clauses; each designed to pick up a different type of exception
- For example: the following code catches exceptions of type int, string and vector<int>

Catch example

```
try {
      // code that can throw an exception
    } catch (int i) {
      // if the code throws an int, it continues here
    } catch (string s) {
      // if the code throws a string, it continues here
    } catch (vector<int> vi) {
      // if the code throws a vector<int>, it continues here
    }
```

If the try block throws an exception, control flow is passed to the correct catch clause

Catch all

- A special syntax is:
  catch (…) {
  // exception handler
  }
- In this case all exceptions not yet handled will be handled (catch all)

Catch all example

- Example:

```
try {
  // code that can throw an exception
} catch (int i) {
  // if the code throws an int, it continues here
} catch (string s) {
  // if the code throws a string, it continues here
} catch (vector<int> vi) {
  // if the code throws a vector<int>, it continues here
} catch (…) {
  // if the code does not throw an int or a string or a
vector<int>, it continues here
}
```

Catch clause – additional properties

- In a succession of catch() the parameter of each catch must be unique. The following is illegal:

```
try {
  // do something
} catch (int i) {
  // if the code throws an int, it continues here
} catch (int i) {
  // error
}
```

- No automatic type conversion is performed in catch clauses
- C++ allows a variety of options for catching: by value, by reference, or by pointer

How to catch

- It is possible to catch exceptions:
  - By value:
    catch (Exception e) {}
  - By reference:
    catch (Exception& e) {}
  - By pointer:
    catch (Exception* e) {}
- It is similar to passing arguments to functions

How to catch

- The recommendation is to catch by reference
- Why ?
  - Catch by value is more expensive because it makes (at least) a copy of the object
  - Catch by value can result in slicing and forbids dynamic binding of virtual functions in derived classes (if the base class is caught)
  - Catch by pointer may lead to memory leak (who has the responsibility to delete the exception)
  - Catch by pointer can cause memory violation if the object was thrown by taking the address of a local object

Propagation of exceptions

- Exceptions can not be ignored in C++
- If not caught after the try block, the exception goes to the next level:
  - Next level of try block (if nested try)
  - Try block surrounding the function call where the exception occurred
  - If not caught at any level, then the program terminates (C++ guarantee that **terminate()** will be called)

Propagation and stack unwinding

- Propagation of an exception can cause to exit from a function during its execution
- In that case, current stack is popped (we use the expression **stack unwinding**)
- Objects local to the function are destroyed (i.e. their destructors are called)
- Dynamically allocated resources are not released. e.g. pointers are not deleted, file handler are not closed, etc

Example

```
class E;
class B;

void f() {
 B b; // b is local to f(), its destructor will be called
 if (someCondition) throw E;
 // rest of f()
}

void g() { B b; f(); }

int main() {
 try {
  g();
 } catch (…) {
  cout << “exception caught” << endl;
 }
}
```

Resource management

- When an exception is thrown, dynamically allocated resources are not released, e.g. pointers are not deleted, file handler are not closed, etc
- The first way to handle dynamically allocated resources is called “catch and re-throw”:
  - If an exception occurs, catch it
  - Release resources and re-throw
  - If no exception occurred, continue execution and release resources at the end

Resource Management

```
void fun(const string& fn) {
    FILE* f = fopen(fn.c_str(), “r”);
    try {
      // do something with f
    } catch (…) {
      fclose(f);
      throw;
    }
   fclose(f);
  }
```

Resource Acquisition Is Initialization (RAII)

- Since destructor of local objects are always called even when an exception is thrown, let us encapsulate the resource in an object
- When the destructor is called, the resource (allocated in the constructor) will be released in the destructor
- This is called: Resource Acquisition Is Initialization (RAII)

RAII

```
void fun(const string& fn) {
  FilePtr f(fn.c_str(), “r”);
  // do something with f that may throw an exception
}

class FilePtr {
  FILE* p;
public:
  FilePtr(const char* n, const char* a) { p = fopen(n, a); }
  ~FilePtr() { fclose(p); }
  // operations, e.g. read(), write()
  // operators * and -> to make it behave like a pointer
}
```

RAII

- The code in fun() works because the class FilePtr encapsulates the management of the FILE*
- If a FilePtr object is local to a function, it is destroyed at the end of the enclosing scope
- The FilePtr destructor is called, which in turns release the handle by calling fclose()

Smart pointers and exception

- Smart pointers are objects that behave like pointers (e.g. have operators *, ->, ...)
- The standard library exports one type of smart pointers: auto_ptr in the header <memory>
- An auto_ptr object is initialized by a pointer and can be dereferenced in the way that a pointer can
- auto_ptr objects have a destructive copy semantics: when an auto_ptr is copied into another, the source no longer points to anything

auto_ptr and exception

```
NodeT* getNewNode() {
    NodeT* newNode = new NodeT();
    newNode->next = NULL;
    newNode->data = someFunction();
    return newNode;
   }
```

If someFunction() throws an exception, the resource allocated for newNode will be lost, resulting in memory leak

auto_ptr and exception

```
NodeT* getNewNode() {
    auto_ptr<NodeT> newNode(new NodeT);
    newNode->next = NULL;
    newNode->data = someFunction();
    return newNode.release(); // auto_ptr stops to manage
   the memory
   }
```

If someFunction() throws an exception, the resource allocated by new NodeT will be released when the destructor for the auto_ptr object is called.
This destructor is guaranteed to be called even if an exception occurs in someFunction().

Handling failure in constructors

- Constructors are not returning a separate value so it is not possible to return an error code
- Throwing an exception is the best solution to indicate a failure in a constructor
- Exception handling allows to transmit a failure during construction out of the constructor

Example

```
class Vec {
      public:
        class SizeException {};
        static const int maxSize = 32000;
        Vec(int sz) {
          if (sz < 0 || sz > maxSize) throw SizeException();
          // ...
        }
    };

    Vector* fun(int sz) {
      try {
        Vector* p = new Vector(sz);
          // …
          return p;
      } catch (Vector::SizeException& ex) {
          // handle the size error
      }
    }
```

Exception in constructors

- If the constructor of an object is throwing an exception, the object's destructor is not run.
  - Resources allocated in the constructor have to be handled carefully
- The safest way of handling constructors that acquire resource is with the idiom "resource acquisition is initialization" (RAII)

Handling failure in destructor

- Exceptions should never be thrown in a destructor
- The reason is related to stack unwinding:
  - During stack unwinding local objects are destroyed by calling their destructors
  - If one of these destructors is throwing an exception then the C++ run-time system is in an ambiguous situation:
    - Should it continue to unwind the stack until the catch associated to the first try?
    - Or should it ignore the first try and unwind the stack until the catch associated to the raised exception ?
- When failure occurs inside a destructor, one practical solution is to save the current status in a log file

Handling failure in destructor

- Exceptions should not be propagated outside of destructors
- Example:

```
class Session {};
Session::~Session() {
  try {
    log();
  }
  catch (…) {}      // catch all exception and do nothing
}
```

Standard exceptions

- It is good coding standard to have user defined class for exceptions
- It is possible to have user defined class derived from one of the standard exceptions (the exceptions defined in the standard library)
- All standard exceptions derive from **exception** (defined in header <exception>)
- Common base class for exception can be **runtime_error** (derived from **exception** and defined in header <stdexcept>)

Standard exceptions

- Note: by substitution principle, an exception handler for a base class can also catch a derived class
- catch(Base b) { } will also handle derived class from Base

Example of a user defined exception

```
#include <stdexcept>
#include <iostream>
class MyException : public std::runtime_error {
  public:
    MyException() : std::runtime_error(“MyException”) {};
};

void f() { // some code
  throw MyException();
}

int main() {
  try {
     f();
  } catch (runtime_error& e) {
     std::cout << e.what() << std::endl; // what() returns what the error was
  }
}
```

bad_alloc

- We saw in the lesson on pointers, that if there is not enough memory, the new operator will throw a bad_alloc exception
- Here is an example to catch it

```
#include <stdexcept>
#include <iostream>

int main() {
 int* p[100000];
 try {
   for (int i = 0; i < 100000; i++) {
      p[i] = new int[10000000];
   }
  } catch(std::bad_alloc) {
    cout << “insufficient memory"
            << endl;
  }
  return 0;
}
```

Function and throw

- When declaring functions it is possible to use the **throw** keyword to give indication on the exception that the function will throw
  - void f() throw (); // means that f throws no exception
  - void f() throw (ExceptionA, ExceptionB); // means that f throws only the exceptions ExceptionA and ExceptionB
  - void f(); // means that f can throw any type of exceptions (it can of course throw nothing)

Function and throw

- Problem with exception specification: if a function declares the type of the exception it throws but throws a different type then this will terminate the program (call terminate() then abort())
- It is not possible for compilers to enforce that a function is not throwing an other type than what it declares (because of template among others)