C++

Exceptions

Exception

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

Introduction

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

Exception in C++

- In C++ exception handling is done by:

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

Try block

- Statements and functions that may raise an exception should be put in a **try** block
- A try block is associated with the exception handler following immediately
- Try blocks can be nested

Throw statement

- A throw statement is used to raise an exception
- Syntax is: **throw object** where the type of the object gives the type of the exception to be thrown
- Example 1:
  class E {};
  // …
  throw E(); // it will throw an exception of type E
- Example 2:
  throw 1.0; // it will throw an exception of type double

Catch clause

- Catch clauses are containing the code that will handle the exception
- It must immediately follow a try block
- Syntax is: **catch (type e) { *// exception handler* }**
  or **catch (…) { *// exception handler* }**
- In case of “catch (…)” all exceptions not yet handled will be handled
- In case of “catch (type e)” exceptions of type “type” not yet handled will be handled
- C++ allows a variety of options for catching: by value, by reference, or by pointer
- In a succession of catch() the parameter of each catch must be unique
- No automatic type conversion is performed

How to catch

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

How to catch

- The recommendation is to catch by reference
- Why ?
  - Catch by value is expensive because it makes two copies 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

- 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 will be called by C++)
- But: dynamically allocated resources are not released. For example: pointers are not deleted, file handler are not closed, etc

Example 1

```
class E;
class B;

void f() {
 B b;
 throw E;
}

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

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

Example 2: cleaning allocated resources

```
void f() {
 // some code
 throw E;
}

void g() {
  B* b = new B;
  try {
    f();
  } catch (…) {     // catch all exceptions
    delete b;       // avoid resource leak when exception thrown
    throw;          // propagate exception to caller
  }
  delete b;         // avoid resource leak when no except. thrown
}
```

Handling failure in constructors with exception

- Constructors are not returning so it is not possible to return an error code
- Only possibility is to throw an exception
- Be careful that if the constructor of an object is throwing an exception, the object's destructor is not run.
  - You have to find a way to have data members inside the object to remember what needs to be undone
  - Example: if you need to use pointers, you may want to consider smart pointer instead (hint for curious: std::auto_ptr in the standard library)

Handling failure in destructor

- You should never throw an exception in a destructor
- Instead failure inside a destructor can be handled (for example) by writing in a log file for example
- The reason for not throwing exception in C++ 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 ?

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
- Usually it is good to have this class derives from a standard exception (i.e. exception 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;
  }
}
```

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 problem (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)