Lecture Material

Exceptions

Grouping of Exceptions

- Often, exceptions fall naturally into families. This implies that inheritance can be useful to structure exceptions and to help exception handling.

```
class Matherr{ };
class Overflow: public Matherr{ };
class Underflow: public Matherr{ };
class Zerodivide: public Matherr{ };
// ...

void f()
{
try   {
      // ...
   }
   catch (Overflow) {
      // handle Overflow or anything derived from Overflow
   }
   catch (Matherr) {
      // handle any Matherr that is not Overflow
   }
}
```

Grouping of Exceptions

- Organizing exceptions into hierarchies can be important for robustness of code.

```
void g()
{
try   {
        // ...
    }
    catch (Overflow) { /* ... */ }
    catch (Underflow) { /* ... */ }
    catch (Zerodivide) { /* ... */ }
}
```

- Without exception grouping
  - A programmer can easily forget to add an exception to the list.
  - If a new exception to the math library were added, every piece of code that tried to handle every math exception would have to be modified.

Derived Exceptions

- In other words, an exception is typically caught by a handler for its base class rather than by a handler for its exact class.
- The semantics for catching and naming an exception are identical to those of a function accepting an argument.
- The formal argument is initialized with the argument value.

```
class Matherr {
 // ...
 virtual void debug_print() const { cerr << "Math error"; }
};
class Int_overflow: public Matherr {
 const char* op;
 int a1, a2;
public:
 Int_overflow(const char* p, int a, int b) { op = p; a1 = a; a2 = b; }
 virtual void debug_print() const { cerr << op <<´(´<< a1 <<´,´<< a2 <<´)´;}
 // ...
};
void f()
{
 try  {
        g() ;
      }
      catch (Matherr m) {
        // ...
      }
}
```

- When the *Matherr* handler is entered, *m* is a *Matherr* object – even if the call to *g()* threw *Int_overflow*. The extra information found in an *Int_overflow* is inaccessible.

Derived Exceptions

- Pointers or references can be used to avoid losing information permanently.

```
int add(int x, int y)
{
 if ( (x>0 && y>0 && x>INT_MAX-y)
   || (x<0 && y<0 && x<INT_MIN-y) )
      throw Int_overflow("+",x,y) ;
 return x+y; // x+y will not overflow
}
void f()
{
 try {
      int i1 = add(1,2) ;
      int i2 = add(INT_MAX,-2);
      int i3 = add(INT_MAX,2) ; // here we go!
}
catch (Matherr& m) {
      // ...
      m.debug_print() ;
   }
}
```

- *Int_overflow::debug_print()* will be invoked.

Composite Exceptions

- Not every grouping of exceptions is a tree structure. Often, an exception belongs to two groups, e.g.:

```
class Netfile_err : public Network_err, public File_system_err{ /* ...*/ };
```

- Such a *Netfile_err* can be caught by functions dealing with network exceptions, and also by functions dealing with file system exceptions:

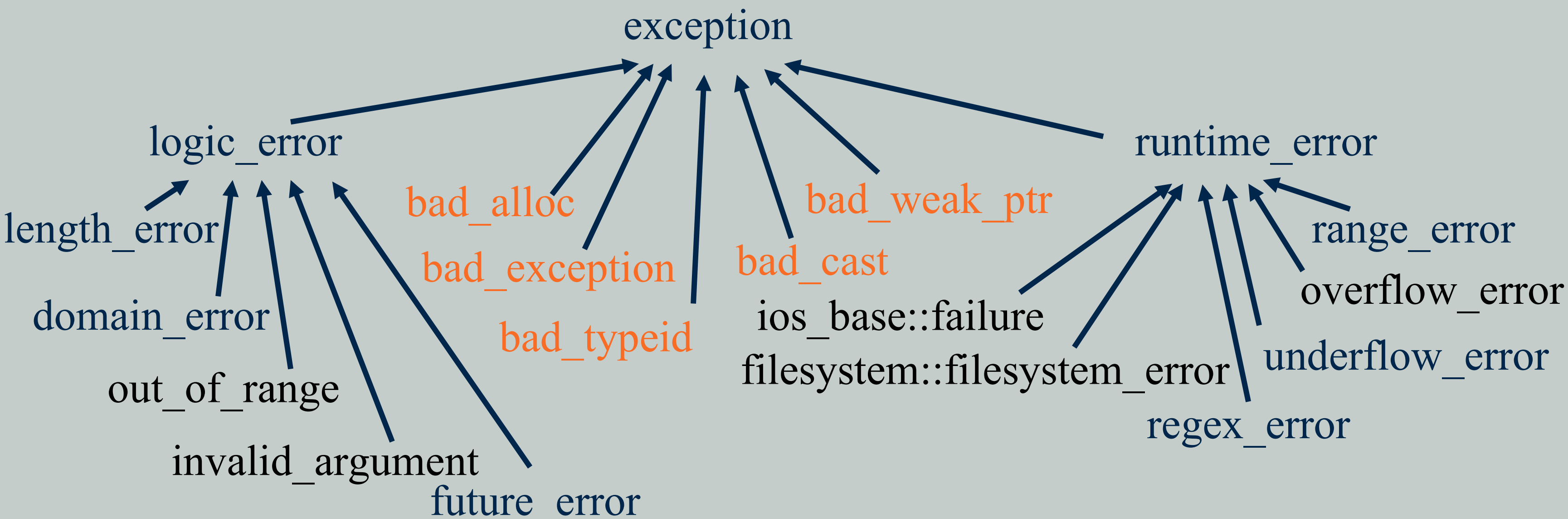
```
void f()
{
 try {
    // something
 }
 catch(Network_err& e) {
    // ...
 }
}

void g()
{
 try {
    // something else
 }
 catch(File_system_err& e) {
    // ...
 }
}
```

Standard Exceptions

- The library exceptions are part of a class hierarchy rooted in the standard library exception class exception presented in *<exception>*:

```
class exception {
 public:
   exception() throw() ;
   exception(const exception&) throw() ;
   exception& operator=(const exception&) throw() ;
   virtual ~exception() throw() ;
   virtual const char* what() const throw() ;
 private:
 // ...
};
```



Standard Exceptions

- Logic errors are errors that in principle could be caught either before the program starts executing or by tests of arguments to functions and constructors.
- Runtime errors are all other errors.
- The standard library exception classes define the required virtual functions appropriately.

```
void f()
try {
 // use standard library
}
catch (exception& e) {
 cout<< "standard library exception" << e.what() << ´\n´; // well, maybe
 // ...
}
catch (...) {
 cout << "other exception\n";
 // ...
}
```

- *exception* operations do not themselves throw exceptions. In particular, this implies that throwing a standard library exception doesn't cause a *bad_alloc* exception. The exception-handling mechanism keeps a bit of memory to itself for holding exceptions (possibly on the stack).

Catching Exceptions

- Consider the example:

```
void f()
{
 try {
    throw E() ;
 }
 catch(H) {
    // when do we get here?
 }
}
```

- The handler is invoked:

  1. If H is the same type as E.
  2. If H is an unambiguous public base of E.
  3. If H and E are pointer types and [1] or [2] holds for the types to which they refer.
  4. If H is a reference and [1] or [2] holds for the type to which H refers.

- An exception is copied when it is thrown, so the handler gets hold of a copy of the original exception.
- An exception may be copied several times before it is caught.
- We cannot throw an exception that cannot be copied.
- The implementation may apply a wide variety of strategies for storing and transmitting exceptions. It is guaranteed, however, that there is sufficient memory to allow new to throw the standard out of memory exception, *bad_alloc*.

Re-Throw

- Having caught an exception, it is common for a handler to decide that it can't completely handle the error.
- In that case, the handler typically does what can be done locally and then throws the exception again.
- The recovery action can be distributed over several handlers.

```
void h()
{
 try {
    // code that might throw Math errors
 }
 catch (Matherr) {
    if (can_handle_it_completely) {
       // handle the Matherr
       return;
    }
    else {
       // do what can be done here
       throw; // rethrow the exception
    }
 }
}
```

- A rethrow is indicated by a throw without an operand.
- If a rethrow is attempted when there is no exception to re-throw, *terminate()* will be called.
- The exception rethrown is the original exception caught and not just the part of it that was accessible as a *Matherr*.

Catch Every Exception

- As for functions, where the ellipsis ... indicates "any argument", *catch(...)* means "catch any exception", e.g.:

```
void m()
{
 try {
   // something
 }
 catch (...) { // handle every exception
   // cleanup
   throw;
 }
}
```

Order of Handlers

- Because a derived exception can be caught by handlers for more than one exception type, the order in which the handlers are written in a try statement is significant. The handlers are tried in order. For example:

```
void f()
{
 try {
   // ...
 }
 catch (std::ios_base::failure) {
   // handle any stream io error
 }
 catch (std::exception& e) {
   // handle any standard library exception
 }
 catch (...) {
   //  handle any other exception
 }
}
```

Order of Handlers

- Because the compiler knows the class hierarchy, it can catch many logical mistakes. For example:

```
void g()
{
 try {
   // ...
 }
 catch (...) {
   // handle every exception
 }
 catch (std::exception& e) {
   // handle any standard library exception
 }
 catch (std::bad_cast) {
   // handle dynamic_cast failure
 }
}
```

- Here, the *exception* will never be considered. Even if we removed the "catchall" handler, *bad_cast* wouldn't be considered because it is derived from *exception*.

Exceptions in Destructors

- From the point of view of exception handling, a destructor can be called in one of two ways:
  - Normal call: As the result of a normal exit from a scope, a *delete*, etc.
  - Call during exception handling: During stack unwinding, the exception-handling mechanism exits a scope containing an object with a destructor.
- In the latter case, an exception may not escape from the destructor itself. If it does, it is considered a failure of the exception-handling mechanism and *std::terminate()* is called.

Exceptions in Destructors

- If a destructor calls functions that may throw exceptions, it can protect itself. For example:

```
X::~X()
try {
 f() ; // might throw
}
 catch (...) {
   // do something
}
```

- The standard library function *uncaught_exception()* returns true if an exception has been thrown but hasn't yet been caught. This allows the programmer to specify different actions in a destructor depending on whether an object is destroyed normally or as part of stack unwinding.

Exceptions That Are Not Errors

- One might think of the exception-handling mechanisms as simply another control structure, e.g.:

```
void f(Queue<X>& q)
{
 try {
    for (;;) {
       X m = q.get() ; // throws 'Empty' if queue is empty
       // ...
    }
 }
 catch (Queue<X>::Empty) {
    return;
 }
}
```

- Exception handling is a less structured mechanism than local control structures such as if and for and is often less efficient when an exception is actually thrown.
- Exceptions should be used only where the more traditional control structures are inelegant or impossible to use.

Exceptions That Are Not Errors

- Using exceptions as alternate returns can be an elegant technique for terminating search functions – especially highly recursive search functions such as a lookup in a tree.

```
void fnd(Tree* p, const string& s)
{
 if (s == p->str) throw p; // found s
 if (p->left) fnd(p->left,s) ;
 if (p->right) fnd(p->right,s) ;
}
Tree* find(Tree* p, const string& s)
{
 try {
   fnd(p,s) ;
 }
 catch (Tree* q) { // q->str==s
   return q;
 }
 return 0;
}
```

- Such use of exceptions can easily be overused and lead to obscure code.
- Whenever reasonable, one should stick to the "exception handling is error handling" view.

noexcept Functions

- Some functions don't throw exceptions and some really shouldn't. To indicate that, we can declare such a function noexcept:

```
double compute(double) noexcept;  // may not throw an exception
```

- Now no exception will come out of *compute()*. But we can try:

```
double compute(double x) noexcept
{
     string s = "A string";
     vector<double> tmp(10);
     // throw an exception here
}
```

- If an exception is thrown inside *compute()* and not handled internally, the program terminates. It terminates unconditionally by invoking *std::terminate()*. It does not invoke destructors from calling functions.
  - It is implementation-defined whether destructors from scopes between the throw and the noexcept (e.g., for *s* in *compute()*) are invoked.
- All destructors are implicitly *noexcept*.

The *noexcept* Operator

- It is possible to declare a function to be conditionally *noexcept*:

```
template<typename T>
void my_fct(T& x) noexcept(Is_pod<T>());
```

- The *noexcept(Is_pod<T>())* means that *my_fct* may not throw if the predicate *Is_pod<T>()* is true but may throw if it is false.
- The predicate in a *noexcept()* specification must be a constant expression. Plain *noexcept* means *noexcept(true)*.
- The *noexcept()* operator takes an expression as its argument and returns *true* if the compiler "knows" that it cannot throw and *false* otherwise.

```
template<typename T>
void call_f(vector<T>& v) noexcept(noexcept(f(v[0]))
{
      for (auto x : v)
            f(x);
}
```

- The operand of *noexcept()* is not evaluated, so in the example we do not get a run-time error if we pass *call_f()* with an empty *vector*.

Exception Specifications - Deprecated

- It is possible to specify the set of exceptions that might be thrown as part of the function declaration. For example:

```
void f(int a) throw (x2, x3) ;
```

- This specifies that *f()* may throw only exceptions *x2*, *x3*, and exceptions derived from these types, but no others.
- If *f()* tries to throw some other exception, the attempt will be transformed into a call of *std::unexpected()*. The default meaning of *unexpected()* is *std::terminate()*, which in turn normally calls *abort()*.
- *throw()* specification is equivalent to *noexcept*.

Uncaught Exceptions

- If an exception is thrown but not caught, the function *std::terminate()* will be called.
- The *terminate()* function will also be called when the exception-handling mechanism finds the stack corrupted and when a destructor called during stack unwinding caused by an exception tries to exit using an exception.
- The response to an uncaught exception is determined by an *_uncaught_handler* set by *std::set_terminate()* from <*exception*>:

```
typedef void(*terminate_handler)() ;
terminate_handler set_terminate(terminate_handler) ;
```

- The return value is the previous function given to *set_terminate()*.
- By default, *terminate()* will call *abort()*.
- An *_uncaught_handler* is assumed not to return to its caller. If it tries to, *terminate()* will call *abort()*.

Uncaught Exceptions

- It is implementation-defined whether destructors are invoked when a program is terminated because of an uncaught exception.
- If you want to ensure cleanup when an uncaught exception happens, you can add a catchall handler to *main()* in addition to handlers for exceptions you really care about.

```
int main()
try {
 // ...
}
catch (std::range_error)
{
 cerr << "range error: Not again!\n";
}
catch (std::bad_alloc)
{
 cerr << "new ran out of memory\n";
}
catch (...) {
 // ...
}
```

- This will catch every exception, except those thrown by construction and destruction of global variables. There is no way of catching exceptions thrown during initialization of global variables.

operator new and Exceptions

- There are versions of *operator new*, which do not throw exceptions when they are out of memory, but return 0 instead. They accept an additional argument *nothrow*.

```
class bad_alloc : public exception{ /* ... */ };
struct nothrow_t {};
extern const nothrow_t nothrow; // indicator for allocation that
                                // doesn't throw exceptions
typedef void (*new_handler)() ;
new_handler set_new_handler(new_handler new_p) noexcept ;
void* operator new(size_t);
void operator delete(void*) noexcept ;
void* operator new(size_t, const nothrow_t&) noexcept ;
void operator delete(void*, const nothrow_t&) noexcept ;
void* operator new[](size_t) ;
void operator delete[](void*) noexcept ;
void* operator new[](size_t, const nothrow_t&) noexcept ;
void operator delete[](void*, const nothrow_t&) noexcept ;
void* operator new (size_t, void* p) noexcept { return p; } // placement
void operator delete (void* p, void*) noexcept { }
void* operator new[](size_t, void* p) noexcept { return p; }
void operator delete[](void* p, void*) noexcept { }
```

```
void f()
{
 int* p = new int[100000] ; // may throw bad_alloc
 if (int* q = new(nothrow) int[100000]) { // will not throw exception
   // allocation succeeded
 }
 else {
   // allocation failed
 }
}
```

Exception Guarantees

- The C++ standard library provides one of the following guarantees for every library operation:
  - The **basic guarantee** for all operations: The basic invariants of all objects are maintained, and no resources, such as memory, are leaked.
    - In particular, the basic invariants of every built-in and standard-library type guarantee that you can destroy an object or assign to it after every standard-library operation
  - The **strong guarantee** for key operations: in addition to providing the basic guarantee, either the operation succeeds, or it has no effect. This guarantee is provided for key operations, such as *push_back()*, single-element *insert()* on a list, and *uninitialized_copy()*
  - The **nothrow guarantee** for some operations: in addition to providing the basic guarantee, some operations are guaranteed not to throw an exception. This guarantee is provided for a few simple operations, such as *swap()* of two containers and *pop_back()*.

Exception Guarantees

- Both the basic guarantee and the strong guarantee are provided on the condition that
  - user-supplied operations (such as assignments and *swap()* functions) do not leave container elements in invalid states
  - user-supplied operations do not leak resources, and
  - destructors do not throw exceptions
- Violating a standard-library requirement, such as having a destructor exit by throwing an exception, is logically equivalent to violating a fundamental language rule, such as dereferencing a null pointer. The practical effects are also equivalent and often disastrous.

Catching Exceptions from Other Threads

- The exceptions thrown in other threads cannot be caught in the main thread
- If you want to get the result of operations in other threads you can use *futures*.

```
include <iostream>
#include <string>
#include <thread>
#include <future>
#include <exception>

std::string fun() {
 std::this_thread::sleep_for (std::chrono::seconds(5));
 throw std::runtime_error("Exception from future");
 return std::string ("Hello from future!");
}

int main(int argc, const char * argv[])
{
 std::future<std::string> fut = std::async(&fun);
          // asynchronously: starts now or later
 try {
  std::string str = fut.get(); // but will be finished here
  std::cout << str << std::endl;
 } catch(std::exception& ex) {
  std::cout << ex.what() << std::endl;
 }
 return 0;
}
```