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:

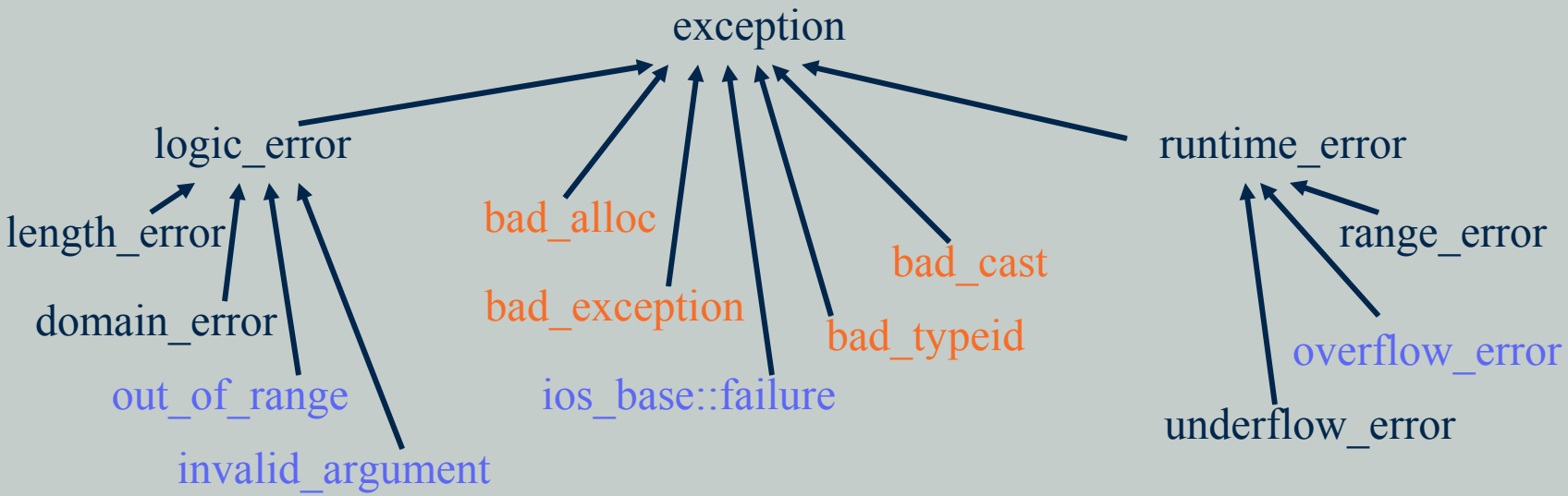
```
void f()
{
 try {
   // something
 }
 catch(Network_err& e) {
   // ...
 }
}

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). Naturally, it is possible to write code that eventually consumes all memory in the system, thus forcing a failure.

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.

Exception Specifications

- 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()*.

Exception Specifications

- The notation:

```
void f() throw (x2, x3)
{
 // stuff
}
```

  is equivalent to:

```
void f()
try
{
 // stuff
}
catch (x2) {throw; } // rethrow
catch (x3) {throw; } // rethrow
catch (...) {
 std::unexpected() ; // unexpected() will not return
}
```

- A function declared without an exception-specification is assumed to throw every exception:

```
int f() ; // can throw any exception
```

- A function that will throw no exceptions can be declared with an empty list:

```
int g() throw () ; // no exception thrown
```

Checking Exception Specifications

- It is not possible to catch every violation of an interface specification at compile time.
- Much compile-time checking is done.
- If any declaration of a function has an exception-specification, every declaration of that function (including the definition) must have an exception-specification with exactly the same set of exception types.

```
int f() throw (std::bad_alloc) ;
int f() // error: exception specification missing
{
 // ...
}
```

- Exceptions-pecifications are not required to be checked exactly across compilation-unit boundaries.

Checking Exception Specifications

- A virtual function may be overridden only by a function that has an exception-specification at least as restrictive as its own (explicit or implicit) exception-specification.

```
class B {
 public:
   virtual void f() ; // can throw anything
   virtual void g() throw(X,Y) ;
   virtual void h() throw(X) ;
};
class D : public B {
 public:
   void f() throw(X) ; // ok
   void g() throw(X) ; // ok: D::g() is more restrictive than B::g()
   void h() throw(X,Y) ; // error: D::h() is less restrictive than B::h()
};
```

- You can assign a pointer to function that has a more restrictive exception-specification to a pointer to function that has a less restrictive exception-specification, but not vice-versa.

```
void f() throw(X) ;
void (*pf1)() throw(X,Y) = &f; // ok
void (*pf2)()throw() = &f; // error: f() is less restrictive than pf2
void g() ; // might throw anything
void (*pf3)() throw(X) = &g; // error: g() less restrictive than pf3
```

- An exception-specification is not part of the type of a function and a *typedef* may not contain one.

```
typedef void (*PF)() throw(X) ; // error
```

Unexpected Exceptions

- Careless exception specifications can lead to calls to *unexpected()*.
  - Such calls can be avoided through careful organization of exceptions and specification of interfaces.
  - Alternatively, calls to *unexpected()* can be intercepted and rendered harmless.
- A well-defined subsystem *Y* will often have all its exceptions derived from a class *Yerr*. For example, given

```
class Some_Yerr : public Yerr{ /* ... */ };
void f() throw (Xerr, Yerr, exception) ;
```

function *f()* will pass any *Yerr* on to its caller. In particular, *f()* would handle a *Some_Yerr* by passing it on to its caller. Thus, no *Yerr* in *f()* will trigger *unexpected()*.

Mapping Exceptions

- Occasionally, the policy of terminating a program upon encountering an unexpected exception is too Draconian. In such cases, the behavior of *unexpected()* must be modified into something acceptable.
- The simplest way of achieving that is to add the standard library exception *std::bad_exception* to an exception-specification. In that case, *unexpected()* will simply throw *bad_exception* and the program will not be terminated.

```
class X{ };
class Y{ };
void f() throw(X, std::bad_exception)
{
 // ...
 throw Y() ; // throw ``bad'' exception
}
```

User Mapping of Exceptions

- Consider a function *g()* written for a nonnetworked environment. Assume further that *g()* has been declared with an exception-specification, so that it will throw only exceptions related to its "subsystem Y":

```
void g() throw(Yerr) ;
```

- Now assume that we need to call *g()* in a networked environment.
  - *g()* will not know about network exceptions and will invoke *unexpected()* when it encounters one.
  - To use *g()* in a distributed environment, we must either provide code that handles network exceptions or rewrite *g()*.
  - Assuming a rewrite is infeasible or undesirable, we can handle the problem by redefining the meaning of *unexpected()*.

User Mapping of Exceptions

- The response to an unexpected exception is determined by an *_unexpected_handler* set by *std::set_unexpected()* from <*exception*>:

```
typedef void(*unexpected_handler)() ;
unexpected_handler set_unexpected(unexpected_handler) ;
```

- To handle unexpected exceptions well, we first define a class to allow us to use the "resource acquisition is initialization" technique for *unexpected()* functions:

```
class STC{ // store and reset class
 unexpected_handler old;
public:
 STC(unexpected_handler f) { old = set_unexpected(f) ; }
 ~STC() { set_unexpected(old) ; }
};
```

- Then, we define a function with the meaning we want for *unexpected()* in this case:

```
class Yunexpected : Yerr{ };
void throwY() throw(Yunexpected) { throw Yunexpected() ; }
```

- Used as an *unexpected()* function, *throwY()* maps any unexpected exception into *Yunexpected* .
- Finally, we provide a version of *g()* to be used in the networked environment:

```
void networked_g() throw(Yerr)
{
 STC xx(&throwY) ; // now unexpected() throws Yunexpected
 g() ;
}
```

- In this way, the exception-specification for *g()* is not violated.

Recovering the Type of an Exception

- Mapping unexpected exceptions to *Yunexpected* would allow a user of *networked_g()* to know that an unexpected exception had been mapped into *Yunexpected*. However, such a user wouldn't know which exception had been mapped. That information was lost in *throwY()*. A simple technique allows that information to be recorded and passed on:

```
class Yunexpected : public Yerr {
 public:
   Network_exception* pe;
   Yunexpected(Network_exception* p) :pe(p) { }
};
void throwY() throw(Yunexpected)
{
 try {
   throw; // rethrow to be caught immediately!
 }
 catch(Network_exception& p) {
   throw Yunexpected(&p) ; // throw mapped exception
 }
 catch(...) {
   throw Yunexpected(0) ;
 }
}
```

- It is not possible for an *unexpected()* function to ignore the exception and return. If it tries to, *unexpected()* itself will throw a *bad_exception*.

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 memeory, 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) throw() ;
void* operator new(size_t) throw(bad_alloc) ;
void operator delete(void*) throw() ;
void* operator new(size_t, const nothrow_t&) throw() ;
void operator delete(void*, const nothrow_t&) throw() ;
void* operator new[](size_t) throw(bad_alloc) ;
void operator delete[](void*) throw() ;
void* operator new[](size_t, const nothrow_t&) throw() ;
void operator delete[](void*, const nothrow_t&) throw() ;
void* operator new (size_t, void* p) throw() { return p; } // placement
void operator delete (void* p, void*) throw() { }
void* operator new[](size_t, void* p) throw() { return p; }
void operator delete[](void* p, void*) throw() { }
```

```
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
 }
}
```