Lecture 25

- Exceptions
- Smart pointers

Error Handling

- **Historical Way**
  - Use function return codes to indicate error conditions
  - E.g. int fgetc(FILE *stream);
    - Returns read character (value in 0..255)
    - Or -1 if read error occurred
  - Drawbacks
    - What if function returns full range of values?
    - **Users can ignore errors**
- **Modern Solution: Exceptions**

Exceptions

- Dealing with rare error conditions
- Write code as if nothing can go wrong
- Enclose it in **try-block** which will be exited if some operation fails and **throws** an exception
- Add a **catch-block** to handle exceptions

Syntax

```
function definition:

<type> <func>(<ParamList>) [throw (<ExceptionList>)] {
  ... throw <exception>(<Params>);
}

If <ExceptionList> is empty, <func> can't throw exc.
If throw clause is missing, <func> can throw anything.
Important: if <func> throws an exception not on the
list, function std::unexpected() is called
(terminates)

try-catch block:

try { ... }
catch (<type> [<var>]) { ... }   (can be more than
one)

Or catch (...) { ... } : catches all exceptions

Re-throw in catch blocks (throw;): catch search cont.
```

Example

```
struct MyException {
  int what;
  MyException(int w=0): what(w) { }
};

void foo() {
   ...
   if (error) throw MyException(5); // exceptional case
   ...
}

int main() {

   try { foo(); }

   catch (MyException &e) {
      cout << "caught exception: " << e.what;
   }
}
```


Catching Exceptions

- Once an exception is thrown (can be **any type**!), program execution is stopped
- The runtime system then looks for the next catch statement whose type is compatible with the thrown value:
  - If the exception was thrown in a try block, the following catch statements are checked
  - If no match, the search for an exception handler resumes in the caller (**"stack unwinding"**) after all **local objects have been destroyed**.
  - If no matching catch statement is found, the program is aborted by calling **terminate()**
- If match found, execution resumes there

How to Catch?

- All exception objects are copied in the stack unwinding process, possibly many times
  - Because local temporal objects are destroyed
- Exceptions should be caught **by reference**. E.g.
  - Catch-by-pointer: delete or not delete?
  - Catch-by-value: one additional copy, possible slicing!
- Be aware that catching exceptions is expensive – **exceptions should be rare events!**

Example

```
void foo() {
   ...
   if (error) throw MyException();
      // creates local object
      // while stack is unwound, this object gets copied
      // everytime, because temporal objects are deleted
      // when function is exited
}

int main() {

   try { foo(); }

   catch (MyException &e) {  // no additional copy!
   }
   catch (MyException e) {   // bad: additional copy!
   }
   catch (MyException *e) {  // bad: delete or not?
   }
}
```

Operator new and Exceptions

- **new** throws **std::bad_alloc** in case memory is unavailable
- Thus, checking the result of new (!=0) is a waste of time – it's always != 0
- C++ standard demands that memory is available if new doesn't throw
  - In practice, however, this is **O/S dependent**
  - I.e.: In some O/S's memory allocation always succeeds, and you'll learn that you don't have enough memory later – segfault ...

Code must be Exception Safe

- Resource deallocation code may not be reached in case of exceptions
- Use the **RAII scheme**:
  Resource **A**llocation **I**s **I**nitialization
- Exceptions within constructors must be **handled right away** to free resources (and maybe re-thrown)
  - Destructor is not called on partly constructed objects
- Exceptions must **not leave destructors**
  - If an exception occurs in destructor while unwinding the stack, program terminates
  - Partly completed destructor has not done its job!

RAII Examples

- Say good-bye to using local pointers for memory allocation
  - T *p = new T; .... delete p;
  - delete p may not be executed if exception is thrown!
  - Solution: smart pointers (next slides)
- Open fstreams with constructor call
  - ofstream os("output.txt");
  - When os goes out of scope, file is closed

Smart Pointers

- Objects that look, act, and feel like built-in pointers
- Used for resource management. E.g.
  - Reference counting
  - Solving the pointers & exceptions problem
- Gain control over:
  - Construction and destruction
  - Copying and assignment
  - Dereferencing

Auto Pointers

- Sole owner of objects
- When auto pointers leave scope, the object they point to is destroyed
- Auto pointer assignment p=q transfers ownership
  - lhs object (*p) is destroyed
  - p now points to rhs object (*q)
  - q points to 0
- Dangerous:
  - storing auto pointers in containers - why?
  - passing them by value – transferring ownership!
- Usual meaning of *p and p->

auto_ptr Example

```
#include <memory>
using namespace std;

class Foo { ... };

void foo() {
   auto_ptr<Foo> p = new Foo; // or p(new Foo);
   bar(p);
   ...
   // p is are destroyed here (releasing Foo obj.)
   // even if exceptions is thrown in bar()
}
```


Auto Pointer Implementation

```
template <typename T> class auto_ptr {
public:

   auto_ptr(T *p_ = 0) : p(p_) { }

   ~auto_ptr() { delete p; }     // here's the
magic!
   ...

   T& operator*() const { return *p; }
   T* operator->() const { return p; }
   T get() const { return p; }

private:
   T *p;   // actual pointer
}
```


More Smart Pointers

- In Boost library:
  - scoped_ptr<T>, scoped_array<T>
    - Simple sole ownership of single object or array, resp.
    - Cannot be copied (safeguard)
  - shared_ptr<T>, shared_array<T>
    - Shared, reference counted ownership of single object or array, respectively
    - Can be stored in STL containers
    - Cannot handle cyclic data structures
  - More...
- These template classes will become part of the C++ standard library

Scoped Examples

```
#include <boost/scoped_ptr.hpp>
#include <boost/scoped_array.hpp>
using namespace boost;

void foo() {
   scoped_ptr<Foo> p(new Foo);
   scoped_ptr<Foo> q = p;  // illegal, safeguard!

   p->bar(); ...      // use like regular pointer

   scoped_array<Foo> pa(new Foo[100]);
   scoped_array<Foo> qa = pa;  // illegal

   pa[10].bar();      // use like regular array

   // p destroyed here => destroys Foo object
   // pa destroyed here => destroys Foo array
}
```


Shared Example

```
#include <boost/shared_ptr.hpp>
using namespace boost;

void foo(shared_ptr<Foo> &q) {
   shared_ptr<Foo> p(new Foo); // reference count
1
   q = p;              // copy => reference count
2

   // p destroyed here => reference count 1
   // Foo object not destroyed yet!
}

void main() {
  shared_ptr<Foo> q;

  foo(q); ...

  // q destroyed here
```