Practical Programming Methodology
(CMPUT-201)

Michael Buro

Lecture 25

- Exceptions Continued
- RAII
- Smart Pointers
- C/C++ Tips

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 ...

Exception Safety Example

```
void bar()
{
  throw MyException();
}

void foo()
{
  int *p = new int[1000];

  bar();

  delete [] p; // not executed -> memory leak
}
```


Exception Safety Paradigm: RAII

Resource deallocation code may not be reached in case of exceptions

Use the RAII scheme:

Resource Allocation Is Initialization

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 (coming up)

Open fstreams with constructor call

- `ofstream os("output.txt");`
- When os goes out of scope, file is closed

Another RAII Application

Locking critical regions in concurrent programs

```
void foo()
{
  XyzLib::Mutex mutex;
  mutex.lock()

  // critical region: only one thread allowed to enter

  do_stuff();

  // when exiting function body mutex.unlock() is
  // automatically called in the destructor of mutex
  // even if an exception is thrown
  //
  // otherwise: program could get dead-locked!
}
```

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 Poiners

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 transfers 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 destroyed here (releasing Foo obj.)
  // even if exception 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 (Boost)

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

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
  // => reference count 0 => object destroyed
}
```


Final Exam

- Wednesday April 26, 2-4pm, here
- Bring OneCard – will be checked
- Closed Book
- Covered material: everything lectures, labs, assignments

REVIEW – C/C++ Programming Tips

"Wisdom and beauty form a very rare combination."
(Petronius Arbiter, Satyricon XCIV)

"With great power comes great responsibility."
(Spiderman's Uncle)

Why ...

C?

- Code is FAST; compiler is FAST; often only little slower than hand-written assembly language code
- Lingua Franca of computer science
- Portable. C compilers are available on all systems
- Compilers/interpreters for new languages are often written in C

C++?

- C + classes + templates: FAST + CONVENIENT
- You are still in total control, unlike Java or C#

From C to C++

Use const and inline instead of `#define`

- Macros are not typesafe
- Macros may have unwanted side effects. Use inline functions instead! (e.g. `#define max(a,b) ((a)>(b)?...)`)

Prefer C++ library I/O over C library I/O

- C's `fprintf` and friends are unsafe and not extensible. Like the syntax? Use boost::format!
- C++ iostream class safe and extensible
- iostream speed is catching up, so speed is hardly a reason anymore for choosing C-library I/O

Prefer C++ style casts

Distinguish between pointers and references

Memory Management

Use the same form in corresponding calls to `new` and `delete`

- `int *p = new Foo;      ... delete p;`
- `int *p = new Foo[100]; ... delete [] p;`

For each new there must be a delete

Delete pointer members in destructors
otherwise you are creating memory leaks

No need for checking the return value of new
It throws an exception if no memory available

`delete p` with p=0 is OK (ignored, no check req.)

The "Big-4"

Define copy constructor and assignment operator when memory is dynamically allocated
 default bit-wise copy is not sufficient in this case

Make destructors virtual in base classes
 otherwise base class pointers can't call the right destr.

Have operator= return reference to `*this`
 for iterated assignments `a = b = c ...`

Assign to all data members in operator=

Check for self assignment in operator=
 `if (this == &rhs) return *this;`

Operators

Never overload `&&` `||` `,`

Distinguish between prefix and postfix forms of ++/--

- they (should) return different types
- `++i` : returns reference to `i`
- `i++` : returns value of temporary object (can be slower!)

Be consistent. E.g. `++` `+=` `prefix++` `postfix++` should have related semantics

Class/Function Design (1)

Guard header files against multiple inclusion
`#ifndef ClassName_H ...`

Strive for complete and minimal interfaces

- complete: users can do anything they need to do
- minimal: as few functions as possible, no overlapping

Minimize compilation dependencies between files

- Consider forward declaration in conjunction with pointers/references to minimize file dependencies
- `class Address;`
  `class Person { ... Adress *address; ... }`
- No need to `#include "Address.h"`

Class/Function Design (2)

Avoid data members in public/protected interfaces
 use inlined get/set functions – more flexible

Use const whenever possible

Pass and return objects by reference
 But don't return references to non-existent objects like local variables!

Avoid returning writable "handles" to internal data from const member functions
 otherwise constant objects can be altered

Inheritance

Make sure public inheritance models "is a"

Never redefine an inherited non-virtual function
- different results for `pBase->f()` and `pDeriv->f()`

Never redefine an inherited default parameter value
- Dirtual functions are dynamically bound
- Default parameters are statically bound

Avoid casting down the inheritance hierarchy
- Use virtual functions instead

Exceptions

Prefer exceptions over C-style error codes

Use destructors to prevent resource leaks
- Say good-bye to pointers that manipulate local resources – use smart pointers

Prevent resource leaks in constructors
- Destructors are only called for fully constructed objects

Prevent exceptions from leaving destructors
- Exceptions within exceptions terminate program
- Special case: exceptions call destructors ...

Catch exceptions by reference
- All alternatives create problems

Efficiency

Choose suitable data structures and efficient algorithms

Consider the 80-20 rule
- 80% of the resources are used by 20% of the code
- Focus your optimization efforts by using profilers

Avoid frequent heap memory allocation

Know how to save space
- bits, bytes, unions, home-brewed memory allocators

Understand costs of virtual functions, multiple inheritance, exception handling, and RTTI

Consider alternative libs. (e.g. iostream vs. stdio)

STL Tips (1)

Choose your containers wisely
- sequence/associative/hash, speed, memory consumption

Careful when storing pointers in containers
- - if the container owns the objects they have to be destroyed before the container
- - possible dangling pointers to vanished objects
- - specify comparison functors

If speed matters, use vectors or hashed associative containers. If speed really matters, don't use STL (for now, but STL implementations are becoming faster)

STL Tips (2)

Make sure destination ranges are big enough

Note which algorithms expect sorted ranges

Have realistic expectations about thread safety of STL containers: YOU need to lock containers

Call `empty()` instead of checking `size()` against 0

Make element copies cheap and correct
- STL copies elements often

Make sure comparison functions implement strict weak ordering

More tips in: S.Meyers: Effective STL