Exception safety
Exceptions

- Runtime errors in low-level functions should not handle it themselves, but rather notify the caller.

- This is accomplished in C++ by “throwing” exceptions.
Why care?

• “It’s pretty near impossible to write robust code in modern C++ without knowing about exception safety issues. Period.”
  — Herb Sutter, More exceptional C++
Nitty gritty details

• Exceptions are passed using objects

• These objects are allocated on a special stack and stick around until their handler (catch block) terminates.

• As a rule, pass exceptions by reference:
  – Catch (MyException &)
Pop quiz

- Is this piece of code correct C++?

```cpp
int main () {
    doAlloc(blah); // will throw
    catch (myException &)
        cout << “blah blah” << endl;
}
```
Overview

• Code is said to be “exception-safe” if run-time failures will not produce ill-effects (memory leaks, garbled data, etc.)

• As discussed previously, errors are handled in C++ with try/catch blocks and thrown exceptions.
Different levels exist

• “no throw guarantee”
  – Operations are guaranteed to succeed, even in exceptional situations
  – Will not throw an exception further up in the code

• “no change guarantee”
  – Operations fail, but such operations have no side effects on other data
More types

• Basic exception safety
  – Some side effects, but stored data will contain valid values even if data is different than before the exception.

• No-leak guarantee
  – Failed operations may store invalid data, but no resource leak

• No exception safety
An example

• Consider adding an element to a C++ `std::vector` that requires new memory allocation.

• This memory allocation may fail and throw an exception. So “no throw guarantee” is impossible.

• The vector could offer the “no change” guarantee fairly easily; in this case, the insertion succeeds or v remains unchanged.
Example (cont.)

• With the basic exception safety guarantee, if the insertion fails, v may or may not contain x, but at least it will remain in a consistent state.

• If the vector makes only the minimal guarantee, it’s possible that the vector may be invalid. For instance, perhaps the size field of v was incremented but x wasn’t inserted, making the state inconsistent.

• With no guarantee, the program may crash.
Golden rules to remember

• Constructors (default, non-default, copy) should provide basic (no leaks) safety.

• Copy assignment (operator=) should provide “no change” guarantee

• Deconstructor should never throw.
Some C++ “urban legends”

- Interactions between templates and exceptions are not well understood
  - Answer: As many of you know, g++ and templates aren’t the best of friends but once you have template specializations, they are like any other function
More myths

• Dealing with exceptions will “slow code down”
  – Answer: Given the special stack, if no exceptions are thrown, code runs as fast as if there were no error checking.
  – Throwing exceptions has the same overhead as calling a function.
  – Further, the structure of try/catch blocks leads to compiler optimization that is not possible with if/then blocks (or similar), speeding up this slightly.
Another pop quiz

- What does the code below do? Where can things go wrong?

Class C : public A {
    B b;
}
Deeper questions

• Consider this line of code:
  – {parrot p;}

• When do you think this object is born?

• When exactly does this object die?
Throwing exceptions in constructors

• If a constructor fails, an object never existed

• If there is no object, no deconstructor runs. This makes absorbing exceptions interesting.
Details

• In the C++ standard, if the handler does not exit by throwing an exception (a throw statement), and control reaches the end of a block, the exception is “rethrown”

• This implies that a constructor ** must ** emit some exception, either the same one or a different one
Why are try blocks useless in deconstructors?

• Deconstructors must never be allowed to emit an exception.

• If it does, it can cause a bunch of (pretty) bad things to happen so we’ll stay away from them

• Short version: Don’t throw in deconstructors!!
How?

• To ensure deconstructors don’t throw, make sure you catch any exceptions like this:

```cpp
T::~T() {
    try {
        close();
    }
    catch () {}
}
```

T::close();  // can throw