COMP6771
Advanced C++ Programming

Week 5
Part One: Exception Handling

2017

www.cse.unsw.edu.au/~cs6771

Memory Management & Exception Handling

1. Part I: Exception Handling
   - Exception objects
   - try-catch blocks
   - Catching exceptions
   - Exception specifications
   - Exception handling
2. Part II: Memory Management
   - The new smart pointers in C++11

What Are Exceptions and Why?

- Exceptions: run-time anomalies
- Exception-handling: a run-time mechanism to communicate exceptions between two unrelated parts
  - A library function detects a run-time error and raises an exception
  - Its user catches the exception if she/he knows how to handles it

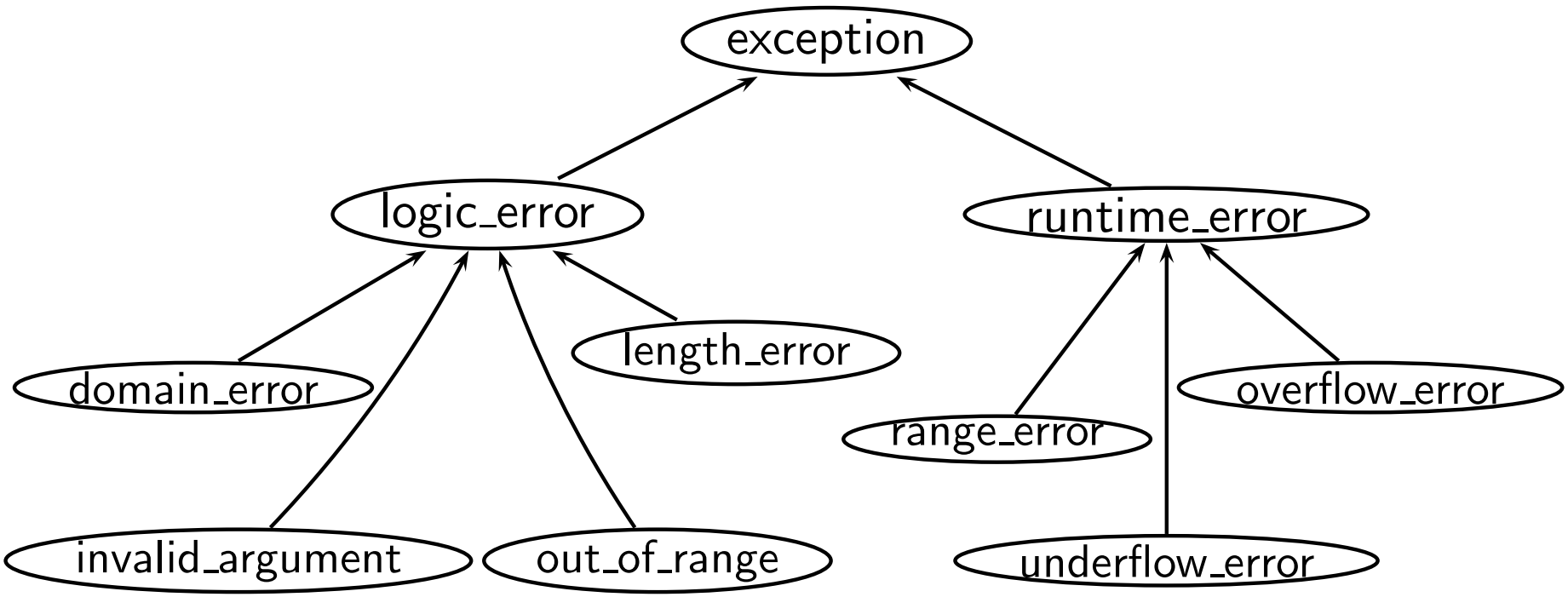
Exception Objects

- In C++, any object can be an exception object:

```
throw 100;
throw true;
throw 'a';
throw "It is a mess here!";
throw 2.1;
throw bad_push{};
throw std::out_of_range("you are dead!");
```

- A class object can communicate more information

Standard Exceptions (#include <stdexcept>)



- The class of your exception objects can be unrelated to this

Throwing an Exception

- The `push()` function (for a bounded stack):

```
1 void Stack::push(const int &item) {
2   if (full()) throw bad_push{currentIndex};
3   stack_[++top_] = item;
4 }
```

- The exception class:

```
1 class bad_push {
2 public:
3   bad_push(int idx) : index {idx} { }
4   value() { return index; }
5 private:
6   int index;
7 };
8 class bad_pop { };
```

The try-catch Block

```
...
Stack s{2};
try {
    s.push(1);
    s.push(2);
    s.push(3);
} catch (bad_push &e) {
  cout << "bab_push: index=" << e.value() << endl;
} catch (bad_pop) {
  cout << "bab_pop caught!" << endl;
}
...
```

The try-catch Block (Cont'd)

```
try {
  throw 100;
} catch (int &i) {
  cout << i << endl;
} catch (...) { } // the catch-all handler
```

- Can throw a value of a primitive type
- The type declaration in a catch phrase is:
  - a built-in type, or
  - a derived type (arrays, classes, etc.)
- Classes are recommended to be used, in general

The type in an exception declaration cannot be an rvalue reference (since an thrown exception is treated always as an lvalue.

The try-catch Block (Cont'd)

```
1 try {
2   throw 100;
3 } catch (double) {
4   // not here!
5 } catch (int) {
6   // but caught here!
7 }
```

```
1 try {
2   throw logic_error{"blah"};
3 } catch (exception) {
4   // caught here!
5 } catch (logic_error) {
6   // not here!
7 }
```

- The catch phrase tried by the first match rule:
- Limited type conversions (unlike parameter passing):
  - nonconst to const
  - derived to base
  - an array to a pointer to the array
- In the case of the hierarchy of exception classes, the derived classes should be listed earlier

Catch Exceptions by `const&` if Possible

- C++ specifies that an object thrown in `throw obj` is always copied (because obj could be a local automatic object!)
- Catch-by-pointer should not be used. Why?
  Who is to delete the pointed-to object? See FAQ 17.8
- Problems with Catch-by-value:
  - inefficient due to object copying
  - causes the slicing problem, and
  - cannot exploit polymorphism and dynamic binding
- Catch-by-reference avoids all these problems
- For details, see Item 13, Meyer's More Effective C++

Catch-by-Value

```
1  #include <iostream>
2
3  class X {
4  public:
5    X() { std::cout << "X constructed" << std::endl; }
6    X(const X &x) { std::cout << "X copy-constructed" << std::endl; }
7    ~X() { std::cout << "X destructed" << std::endl; }
8  };
9
10 void g() {
11   throw X{};
12 }
13
14 void f() {
15   try {
16     g();
17   } catch (X x) {
18     std::cout << "caught in f; rethrow" << std::endl;
19     throw;
20   }
21 }
22
23 int main() {
24   try {
25     f{};
26   } catch (X x) {
27     std::cout << "caught in main" << std::endl;
28   }
29 }
```

How many objects created? A: 1 B: 2 C: 3 D: 4

Catch-by-Value

```
1 OUTPUT:
2 X constructed
3 X copy-constructed
4 caught in f; rethrow
5 X destructed
6 X copy-constructed
7 caught in main
8 X destructed
9 X destructed
```

How a C++ compiler implements exception handling:
http://www.codeproject.com/KB/cpp/exceptionhandler.aspx
http://www.hexblog.com/wp-content/uploads/2012/06/Recon-20

Catch-by-Reference

```
1  #include <iostream>
2
3  class X {
4  public:
5    X() { std::cout << "X constructed" << std::endl; }
6    X(const X &x) { std::cout << "X copy-constructed" << std::endl; }
7    ~X() { std::cout << "X destructed" << std::endl; }
8  };
9
10 void g() {
11   throw X{};
12 }
13
14 void f() {
15   try {
16     g();
17   } catch (X& x) {
18     std::cout << "caught in f; rethrow" << std::endl;
19     throw;
20   }
21 }
22
23 int main() {
24   try {
25     f{};
26   } catch (X& x) {
27     std::cout << "caught in main" << std::endl;
28   }
29 }
```

How many objects created? A: 1 B: 2 C: 3 D: 4

Catch-by-Reference

```
1 OUTPUT:
2 X constructed
3 caught in f; rethrow
4 caught in main
5 X destructed
```

Rethrow

- The syntax:

```
throw;
```

- Rethrowing an exception allows it to be handled by a function further up the list of function calls

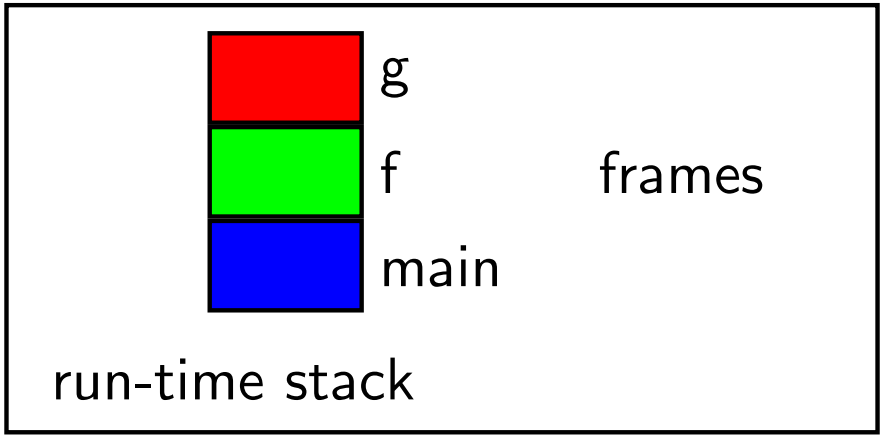
```
catch (T &e1) {
  // modify e1
  throw; // modified e1 rethrown
} catch (T e2) {
  // modify e2
  throw; // the original e2 rethrown
}
```

- Modifications propagated for reference parameters only

Stack Unwinding

```
1  #include <iostream>
2
3  void g() {
4    throw std::string("something is wrong!");
5  }
6
7  void f() {
8    g();
9    std::cout << "never executed" << std::endl;;
10 }
11
12 int main() {
13   try {
14     f();
15   } catch (std::string &s) {
16     std::cout << s << std::endl;
17   }
18   std::cout << "execution resumes from here" << std::endl;
19 }
```

Stack Unwinding



- Stack unwinding: the process of exiting the calling frames in the search for an exception handler
- If a new exception is thrown during stack unwinding and not caught in the function that threw it, terminate() is called

Prevent Exceptions from Leaving Destructors

- During stack unwinding, `terminate()` will be called if an exception leaves a destructor
- The resources may not be released properly if an exception leaves a destructor
- All exceptions that occur inside a destructor should be handled inside the destructor
- All STL types guarantee that their destructors are non-throwing

Exception Handling vs. Object Destruction

- In C++, all fully constructed non-heap objects are destroyed automatically (by the compiler-inserted code)
- Automatic objects:
  - destroyed at the end of their defining blocks {}
  - their destructors called for releasing resources

```
void f() {
  std::string s{"C++"};
  throw 100;
  std::cout << "never executed!";}
  --> std::~string() called for s here!
int main() {
  try { f(); }
  catch (int) { }
}
```

Exception Handling vs. Dumb Pointers

```
void f() {
  std::string *s = new std::string{"C++"};
  throw 100;
  std::cout << "never executed!";
}
int main() {
  try { f(); }
  catch (int) { }
}
```

- The heap string object not freed

Exception-Safe Smart Pointers

```
void f() {
  auto s = std::make_unique<std::string>{"C++"};
  throw 100;
  std::cout << "never executed!";
} --> the destructor for s called
```

- Stroustrup: "resource acquisition is initialisation" (RAII idiom):
  - encapsulate resources (e.g., memory and locks.) inside objects,
  - acquire resources using constructors
- Use destructors to prevent resource leaks

Prevent Resource Leaks in Constructors

```
1  class BookEntry {
2  public:
3    BookEntry(std::string&, std::string&, std::string&);
4  private:
5    std::string theName;
6    Image *theImage;
7    AudioClip *theAudioClip;
8  };
9  BookEntry::BookEntry(std::string &name,
10         std::string &imageFileName,
11         std::string &audioClipFileName) :
12       theName{name} {
13       theImage = new Image{imageFileName};
14       theAudioClip = new AudioClip{audioClipFileName};
15 }
```

If an exception occurs during the construction of `theAudioClip`, the fully constructed `theName` will be destroyed but `theImage` leaks.

Prevent Resource Leaks in Ctors

```
1 BookEntry::BookEntry(std::string &name, ... ) {
2     try {
3       theImage = new Image{imageFileName};
4       theAudioClip = new AudioClip{audioClipFileName};
5     } catch(...) {
6       delete theImage;
7       delete theAudioClip;
8     }
9 }
```

This avoids memory leaks but is messy

Prevent Resource Leaks in Ctors Using Smart Pointers

```
1  class BookEntry {
2  public:
3    BookEntry(std::string&, std::string&, std::string&);
4  private:
5    std::string theName;
6    std::unique_ptr<Image> theImage;
7    std::unique_ptr<AudioClip> theAudioClip;
8  };
9  BookEntry::BookEntry(std::string &name,
10                      std::string &imageFileName,
11                      std::string &audioClipFileName)
12       : theName{name},
13       theImage{std::make_unique<Image>(imageFileName)},
14       theAudioClip{std::make_unique<Image>(audioClipFileName)}
15    { }
```

- Using objects to encapsulate resources
- Resources released when destructors called

Exception Specifications: C++11

- An exception specification is part of a function's interface

```
1 void push() noexcept;    // won't throw
2 void push();             // may throw
```

- Promises the callers not to deal with exceptions
- The violation of a spec detected only at run-time

```
1 void f() noexcept {
2   throw exception{};
3 }
```

Compiles, terminate() called and unspecified for stack unwinding

- But allowed to handle the exceptions raised inside:

```
1 void f() noexcept {
2   try {
3     throw exception{};
4   } catch (...) { } // ok handled inside
5 }
```

Exception Specifications: C++11 (Cont'd)

- Enable some compiler optimisations

For example, `std::vector` will call its move constructor to move elements for some operations if it is no-throwing and its copy constructors otherwise.

- The noexcept operator:

```
1 void f() noexcept(e);
```

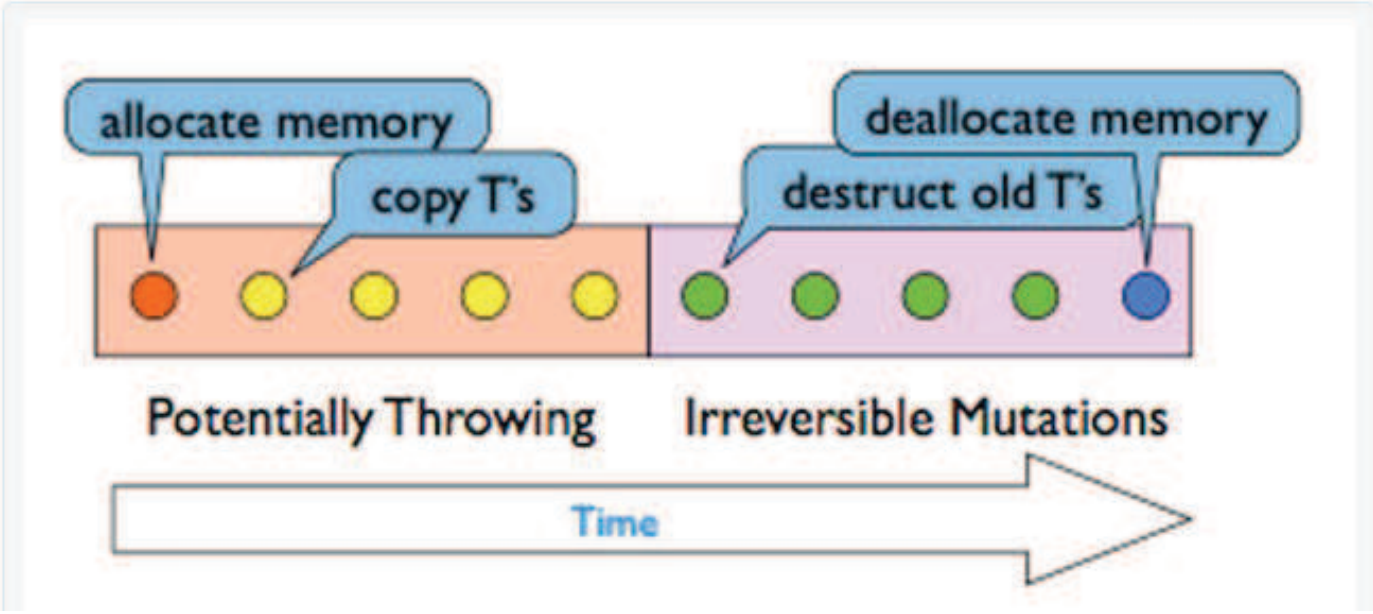
is true if and only if all the functions called by e have nothrowing exceptions and e itself doesn't contain a throw.

```
1 // f has the same exception specifier as g
2 void f() noexcept(noexcept(g()));
```

Common Levels of Exception Safety

- No-throw guarantee, also known as failure transparency: Operations are guaranteed to succeed and satisfy all requirements even in the presence of exceptional situations. If an exception occurs, it will be handled internally and not observed by clients.
- Strong exception safety, also known as commit or rollback semantics: Operations can fail, but failed operations are guaranteed to have no side effects so all data retain original values.
- Basic exception safety, also known as no-leak guarantee: Partial execution of failed operations can cause side effects, but all invariants are preserved and no resources are leaked. Any stored data will contain valid values, even if data has different values now from before the exception.
- No exception safety: No guarantees are made.

Strong-Exception Guarantees



- Operations that may throw, but don't do anything irreversible
- Operations that may be irreversible, but don't throw.

Strong-Exception Guarantees: An Example

```
1  reserve(size_type n) {
2    if (n > this->capacity()) {
3      pointer new_begin = this->allocate( n );
4      size_type s = this->size(), i = 0;
5      try {
6        // copy to new storage: can throw; doesn't modify *this
7        for (; i < s; ++i)
8          new ((void*)(new_begin + i)) value_type( (*this)[i] );
9      } catch(...) {
10      while (i > 0)          // clean up new elements
11        (new_begin + --i)->~value_type();
12      this->deallocate( new_begin );    // release storage
13      throw;
14     }
15     //  -- irreversible mutation starts here --
16     this->deallocate( this->begin_ );
17     this->begin_ = new_begin;
18     this->end_ = new_begin + s;
19     this->cap_ = new_begin + n;
20 }
```