601.220 Intermediate Programming

Exceptions

Exceptions

We use exceptions to indicate a *fatal* error has occurred, where there is *no reasonable way to continue from the point of the error*

It might be possible to continue from *somewhere else*, but not from the point of the error

Exceptions

Behold, a bad program:

```
// exceptions1.cpp:
#include <iostream>

using std::cout;  using std::endl;

int main() {
    size_t mem = 1;
    while(true) {
        char *lots_of_mem = new char[mem];
        delete[] lots_of_mem;
        mem *= 2;
    }
    cout << "Forever is a long time" << endl;
    return 0;
}
```

Exceptions

```
$ g++ -c exceptions1.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o exceptions1 exceptions1.o
$ ./exceptions1
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Aborted
```

Exceptions

Keeps allocating bigger arrays until an allocation fails

The exception makes sense:

- Any pointer returned by `new[]` would be unusable; program doesn't necessarily expect that
- Program can signal that it *does* expect that by *catching* the appropriate exception
    - Since we *don't* do so here, the exception crashes the program

Exceptions

```
char *lots_of_mem = new char[mem];
```

Why not have `new[]` return `NULL` on failure, like `malloc`?

- When call stack is deep: `f1()` -> `f2()` -> `f3()` -> ... propagating errors backward requires much coordination
- If any function fails to propagate error back, chain is broken
- Error encoding must be managed (e.g. 1 = success, 2 = out of memory, ...); no standard

Exceptions are more flexible; often less error prone, more concise than manually propagating errors back through the chain of callers

Exceptions

When an exception is thrown, a `std::exception` object is created

Exception types ultimately derive from `std::exception` base class

Exception's type and contents (accessed via `.what()`) describe what went wrong

Exceptions

Looking in documentation for `new`/`new T[n]`, you can see the exception thrown is of type `bad_alloc`

function

operator new[]

<new>

C++98 | C++11

```
  throwing (1)  void* operator new[] (std::size_t size);
   nothrow (2)  void* operator new[] (std::size_t size, const std::nothrow_t& nothrow_value) noexcept;
placement (3)  void* operator new[] (std::size_t size, void* ptr) noexcept;
```

Allocate storage space for array

Default *allocation functions* (array form).

(1) throwing allocation

Allocates *size* bytes of storage, suitably aligned to represent any object of that size, and returns a non-null pointer to the first byte of this block.
On failure, it throws a `bad_alloc` exception.
The default definition allocates memory by calling `operator new`: `::operator new (size)`.
If replaced, both `operator new` and `operator new[]` shall return pointers with identical properties.

(2) nothrow allocation

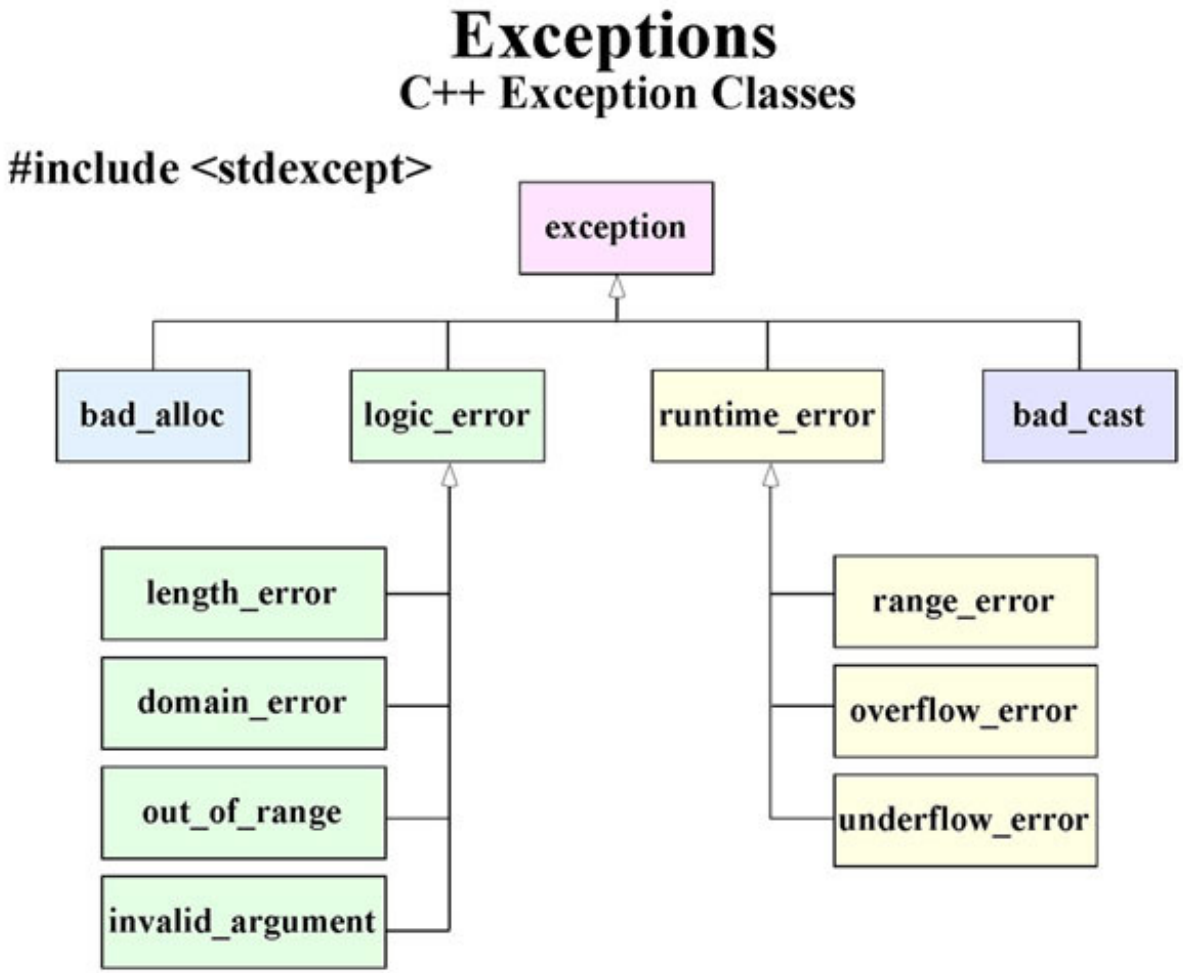
Same as above *(1)*, except that on failure it returns a *null pointer* instead of throwing an exception.

C++98 | C++11

The default definition allocates memory by calling the `nothrow` version of `operator new`: `::operator new (size,nothrow)`.

Exceptions

Standard exceptions:



Exceptions

```cpp
// exceptions2.cpp:
#include <iostream>
#include <new> // bad_alloc defined here

using std::cout;  using std::endl;

int main() {
    size_t mem = 1;
    char *lots_of_mem;
    try {
        while(true) {
            lots_of_mem = new char[mem];
            delete[] lots_of_mem;
            mem *= 2;
        }
    }
    catch(const std::bad_alloc& ex) {
        cout << "Got a bad_alloc!" << endl
             << ex.what() << endl;
    }
    cout << "Forever is a long time" << endl;
    return 0;
}
```

Exceptions

```
$ g++ -c exceptions2.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o exceptions2 exceptions2.o
$ ./exceptions2
Got a bad_alloc!
std::bad_alloc
Forever is a long time
```

Exceptions

Another example:

```
// exceptions3.cpp:
#include <iostream>
#include <vector>
#include <stdexcept> // standard exception classes defined

using std::cout;   using std::endl;
using std::vector;

int main() {
    vector<int> vec = {1, 2, 3};
    try {
        cout << vec.at(3) << endl;
    } catch(const std::out_of_range& e) {
        cout << "Exception: " << endl << e.what() << endl;
    }
    return 0;
}
```

Exceptions

```
$ g++ -c exceptions3.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o exceptions3 exceptions3.o
$ ./exceptions3
Exception:
vector::_M_range_check: __n (which is 3) >= this->size() (which is 3)
```

Exceptions

`try` marks block of code where an exception might be thrown

```
try {
    while(true) {
        lots_of_mem = new char[mem]; // !
        delete[] lots_of_mem;
        mem *= 2;
    }
}
```

Tells C++ "exceptions might be thrown, and I'm ready to handle some or all of them"

Exceptions

`catch` block, immediately after `try` block, says what to do in the event of a particular exception

```
catch(const bad_alloc& ex) {
    cout << "Yep, got a bad_alloc" << endl;
}
```

Exceptions

The point in the program where the exception is actually thrown is the *throw point*

When exception is thrown, we *don't* proceed to the next statement

Instead we follow a process of "unwinding"

Exceptions

Unwinding: keep moving "up" to wider enclosing scopes; stop at `try` block with relevant `catch` clause

```
if(a == b) {
    try {
        while(c < 10) {
            try {
                if(d % 3 == 1) {
                    throw std::runtime_error("!");
                }
            }
            catch(const bad_alloc &e) {
                ...
            }
        }
    }
    catch(const runtime_error &e) {
        // after throw, control moves here
        ...
    }
}
```

Exceptions

If we unwind all the way to the point where our scope is an entire function, we jump back to the caller and continue the unwinding

Exceptions

```
void fun2() { // (called by fun1)
    while(...) {
        try {
            // unwinding from here...
            throw std::runtime_error("whoa");
        } catch(const bad_alloc& e) {
            // only catches bad_alloc, not runtime_error
            ...
        }
    }
}

void fun1() {
    try {
        fun2();
    } catch(const runtime_error& e) {
        // ends up here...
        ...
    }
}
```

Exceptions

If exception is never caught – i.e. we unwind all the way through `main` – exception info is printed to console & program exits

That's what happened in the case of our `bad_alloc` example

Exceptions

```cpp
// exceptions1.cpp:
#include <iostream>

using std::cout;  using std::endl;

int main() {
    size_t mem = 1;
    while(true) {
        char *lots_of_mem = new char[mem];
        delete[] lots_of_mem;
        mem *= 2;
    }
    cout << "Forever is a long time" << endl;
    return 0;
}
```

```
$ g++ -o exceptions1 exceptions1.cpp -std=c++11 -pedantic -Wall
$ ./exceptions1
terminate called after throwing an instance of 'std::bad_alloc'
  what():  std::bad_alloc
Aborted
```

Exceptions

```
// except_unwind.cpp:
#include <iostream>
#include <stdexcept>

using std::cout;  using std::endl;

void fun2() {
    cout << "fun2: top" << endl;
    throw std::runtime_error("runtime_error in fun2");
    cout << "fun2: bottom" << endl;
}

void fun1() {
    cout << "fun1: top" << endl;
    fun2();
    cout << "fun1: bottom" << endl;
}

int main() {
    try {
        cout << "main: try top" << endl;
        fun1();
        cout << "main: try bottom" << endl;
    } catch(const std::runtime_error &error) {
        cout << "Exception handled in main: "
             << error.what() << endl;
    }
    cout << "main: bottom" << endl;
    return 0;
}
```

Exceptions

```
$ g++ -c except_unwind.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o except_unwind except_unwind.o
$ ./except_unwind
main: try top
fun1: top
fun2: top
Exception handled in main: runtime_error in fun2
main: bottom
```

Exceptions

Unwinding causes local variables to go out of scope

Destructors always called when object goes out of scope, regardless of whether scope is exited because of reaching end, `return`, `break`, `continue`, exception, ...

Exceptions

```
// hello_goodbye.h:
#include <iostream>
#include <string>

// Prints messages upon construction and destruction
class HelloGoodbye {
public:
    HelloGoodbye(const std::string& nm) : name(nm) {
        std::cout << name << ": hello" << std::endl;
    }

    ~HelloGoodbye() {
        std::cout << name << ": goodbye" << std::endl;
    }
private:
    std::string name;
};
```

Exceptions

```
// except_unwind2.cpp:
#include <iostream>
#include <stdexcept>
#include "hello_goodbye.h"

using std::cout;  using std::endl;

void fun2() {
    HelloGoodbye fun2_top("fun2_top");
    throw std::runtime_error("runtime_error in fun2");
    HelloGoodbye fun2_bottom("fun2_bottom");
}

void fun1() {
    HelloGoodbye fun1_top("fun1_top");
    fun2();
    HelloGoodbye fun1_bottom("fun1_bottom");
}

int main() {
    try {
        HelloGoodbye main_top("main_top");
        fun1();
        HelloGoodbye main_bottom("main_bottom");
    }
    catch(const std::runtime_error &error) {
        cout << "Exception handled in main: "
             << error.what() << endl;
    }
    return 0;
}
```

Exceptions

```
$ g++ -c except_unwind2.cpp -std=c++11 -pedantic -Wall -Wextra
$ g++ -o except_unwind2 except_unwind2.o
$ ./except_unwind2
main_top: hello
fun1_top: hello
fun2_top: hello
fun2_top: goodbye
fun1_top: goodbye
main_top: goodbye
Exception handled in main: runtime_error in fun2
```

Quiz!

What output is printed by the following program?

```
#include <iostream>
#include <vector>

int main(void) {
  std::vector<int> v = {1, 2, 3};
  try {
    std::cout << 'A' << ' ';
    std::cout << v[3] << ' ';
    std::cout << 'B' << ' ';
  } catch (const std::logic_error &e) {
    std::cout << "exception!" << std::endl;
  }
  return 0;
}
```

A. `A exception!`

B. `A 3 B`

C. `A 0 B`

D. Output is impossible to predict

E. None of the above