Lecture 11.25

# Exception Handling & Smart Pointer (1)

SE271 Object-Oriented Programming (2020)

Yeseong Kim

Original slides from Prof. Shin at DGIST

And Univ. of Washington, CSE333, Spring 2018

# Short Notice

- Team Project – Guideline will be released by this Friday,
  **but please prepare in advance!**
  - Will have a presentation with a recorded video (4 minutes for each team)
  - Will write a report (3~5 pages)

- Don't forget HW4 preliminary league by this sunday

# Today’s Topic

- Exception Handling
  - Exception Class
- Smart Pointer
  - unique_ptr

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <iomanip>
using namespace std;
int main() {
    string name, pn;
    int age;
    ofstream fout("Phonebook.bin", ios::app|ios::binary);
    getline(cin, name);
    cin >> age;
    cin.ignore(100, '\n');
    getline(cin, pn);

    fout << setw(10) << left << name;
    fout << setw(3) << age;
    fout << setw(13) << pn;
}
```

```
Alice Kim
20
010-111-2222
Bob Lee
21
010-1111-2222
```

```
Alice Kim 20 010-111-2222 Bob Lee   21 010-1111-2222
```

What if Name includes digit?

What if age is set by negative value?

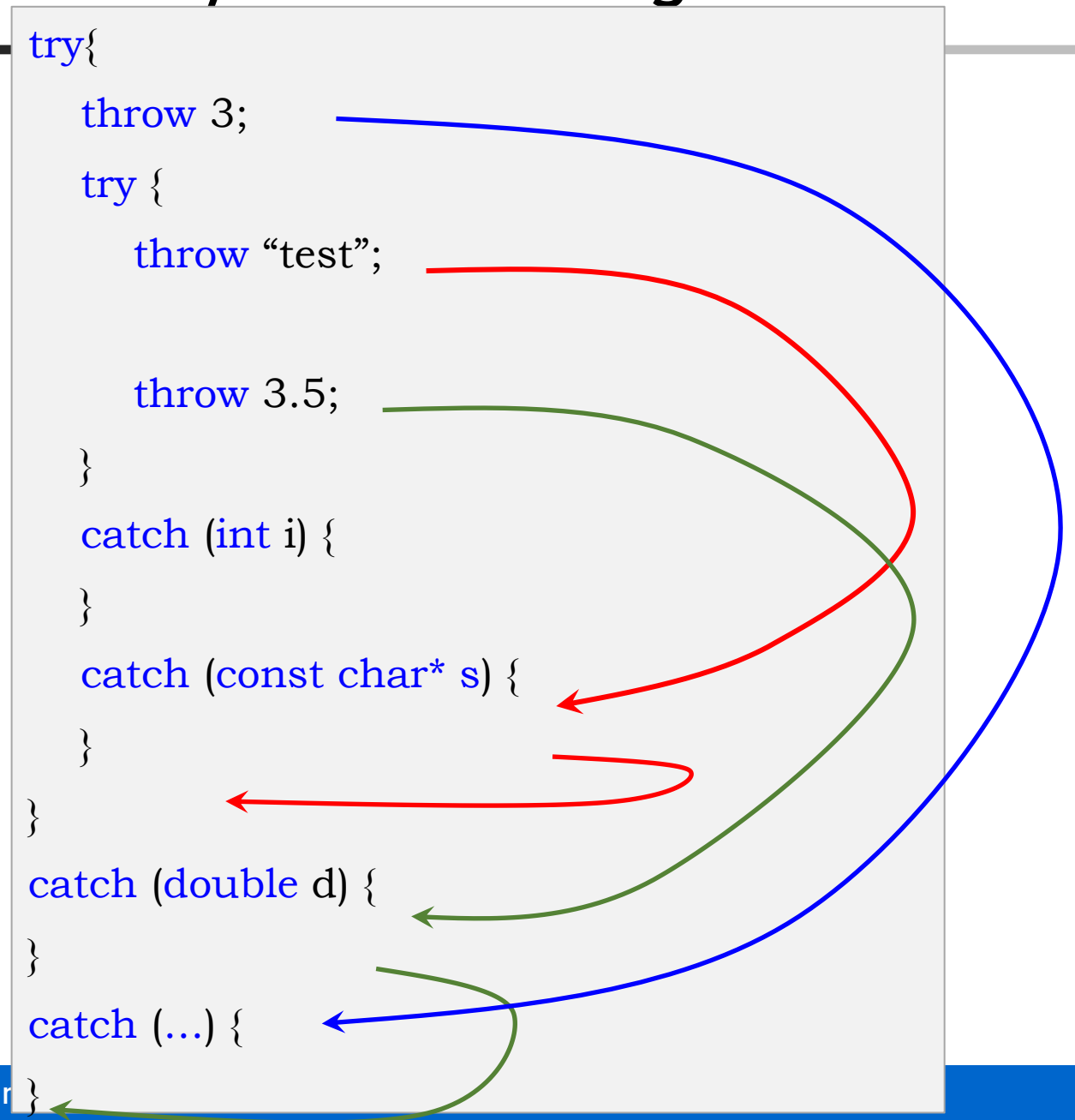
Non-number

What if phone number is strange?

More than 13 words

# Nested Try and Catching

```
try{
    throw 3;
    try {
        throw “test”;

        throw 3.5;
    }
    catch (int i) {
    }
    catch (const char* s) {
    }
}
catch (double d) {
}
catch (...) {
}
```

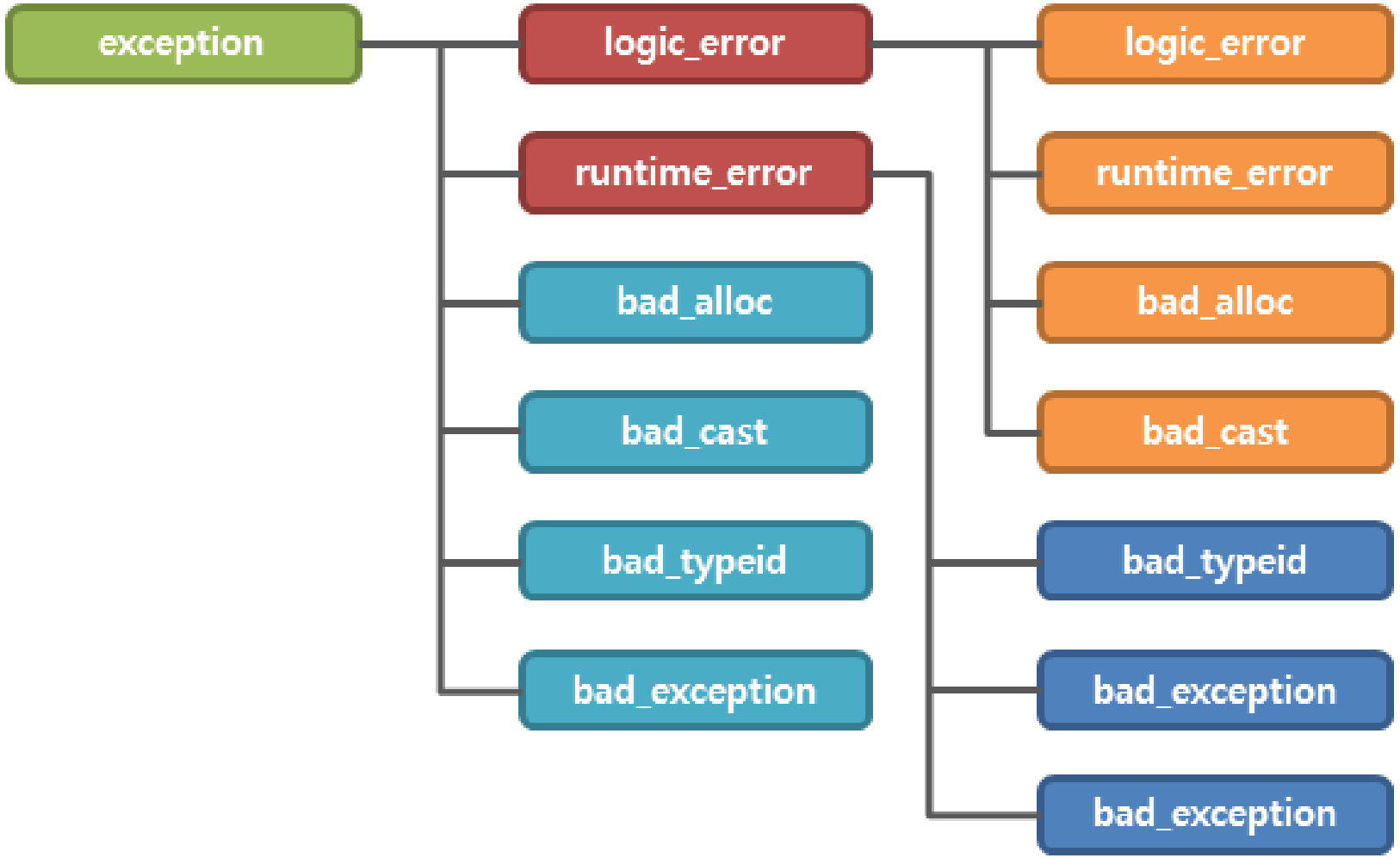


# MyException Class

```
class MyException {
    int errNo;
    string errFunc, errMsg;
public:
    MyException(int n, string f, string m):
        errNo{ n }, errFunc{ f }, errMsg{ m }{}
    virtual ~MyException() {}
    void what(){
        cout << "Error[" << errNo << "] : "
<< errMsg << " at " << errFunc << endl;
    }
};
class MyDivideByZero : public MyException{
public:
    MyDivideByZero(string f) :
    MyException(100, f, "Divide by Zero") {} };
```

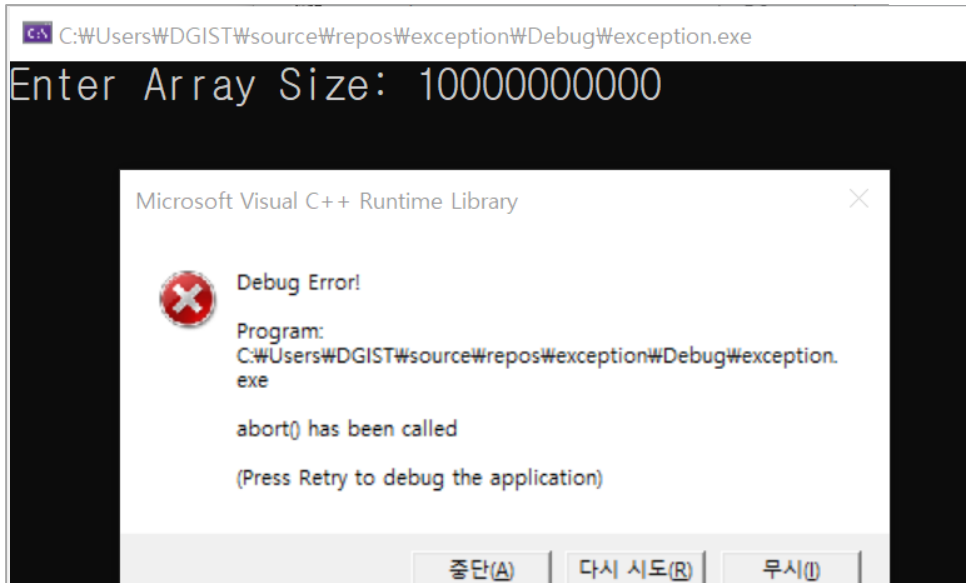
```
int main() {
    int n1{ 10 }, n2{ 0 };
    cin >> n2;

    try {
        if (n2 == 0)
            throw MyException(100, "main()", "Zero");
    }              MyDivideByZero("main()");
    catch (MyException & e) {
        e.what();
    }
}
```

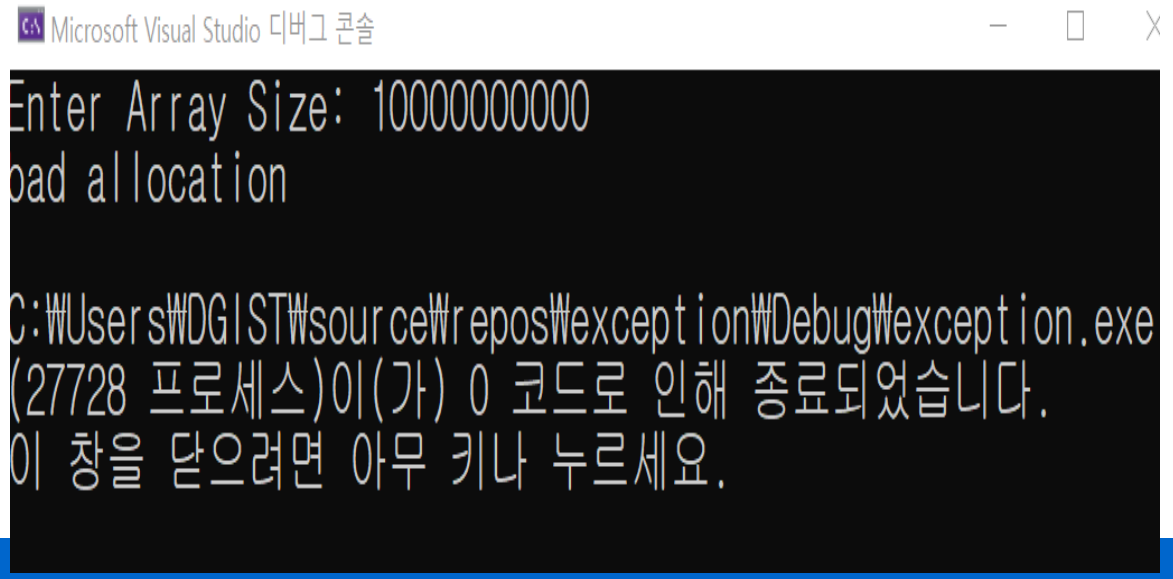
# Exception Class



# Example: Exception Class

```
#include<exception>
int main() {
    int nSize;
    char* arr;
cout << "Enter Array Size: ";
    cin >> nSize;
    try {
        arr = new char[nSize];
        cout << "Array (" << _msize(arr) << ") is
created.";
    delete[] arr;
    }
    catch (bad_alloc & e) {
        cout << e.what() << endl;
    }
}
```



# Inherit from Exception Class & noexcept

```
class  exception {
public:
    exception() noexcept;
    exception(const exception &) noexcept;
    virtual ~exception();
    virtual const char * what() const noexcept;
};
```

```
class MyException : public exception {
public:
    virtual const char* what() const noexcept {
        return “my exception”;
    }
};

try {
    throw MyException{};
}
catch (exception & e) {
    cout << e.what();
}
```

# Uncatched Exception

```
#include <iostream>
#include <exception>
void MyErrorHandler() {
    std::cout << "Error anyway";
    exit(-1);
}
int main() {
    set_terminate(MyErrorHandler);
    try {
        throw 3.14;
    }
    catch (int i) {
    }
}
```

# `std::unique_ptr`

- A `unique_ptr` *takes ownership* of a pointer
  - Part of C++'s standard library (C++11)
  - Its destructor invokes `delete` on the owned pointer
    - Invoked when `unique_ptr` object is `delete`'d or falls out of scope

# Using `unique_ptr`

```cpp
#include <iostream>  // for std::cout, std::endl
#include <memory>    // for std::unique_ptr
#include <cstdlib>   // for EXIT_SUCCESS

void Leaky() {
  int *x = new int(5);  // heap-allocated
  (*x)++;
  std::cout << *x << std::endl;
}  // never used delete, therefore leak

void NotLeaky() {
  std::unique_ptr<int> x(new int(5));  // wrapped, heap-allocated
  (*x)++;
  std::cout << *x << std::endl;
}  // never used delete, but no leak

int main(int argc, char **argv) {
  Leaky();
  NotLeaky();
  return EXIT_SUCCESS;
}
```

# Why are `unique_ptrs` useful?

- If you have many potential exits out of a function, it's easy to forget to call `delete` on all of them
  - `unique_ptr` will `delete` its pointer when it falls out of scope
  - Thus, a `unique_ptr` also helps with *exception safety*

```
void NotLeaky() {
  std::unique_ptr<int> x(new int(5));
  ...
  // lots of code, including several returns
  // lots of code, including potential exception throws
  ...
}
```

# `unique_ptr` Operations

```cpp
#include <memory>   // for std::unique_ptr
#include <cstdlib>  // for EXIT_SUCCESS

using namespace std;
typedef struct { int a, b; } IntPair;

int main(int argc, char **argv) {
  unique_ptr<int> x(new int(5));

  int *ptr = x.get(); // Return a pointer to pointed-to object
  int val = *x;       // Return the value of pointed-to object

  // Access a field or function of a pointed-to object
  unique_ptr<IntPair> ip(new IntPair);
  ip->a = 100;

  // Deallocate current pointed-to object and store new pointer
  x.reset(new int(1));

  ptr = x.release();  // Release responsibility for freeing
  delete ptr;
  return EXIT_SUCCESS;
}
```

# `unique_ptrs` Cannot Be Copied

- `std::unique_ptr` has disabled its copy constructor and assignment operator
  - You cannot copy a `unique_ptr`, helping maintain "uniqueness" or "ownership"

```
#include <memory>   // for std::unique_ptr
#include <cstdlib>  // for EXIT_SUCCESS

int main(int argc, char **argv) {
  std::unique_ptr<int> x(new int(5));  //

  std::unique_ptr<int> y(x);           //

  std::unique_ptr<int> z;              //

  z = x;                               //

  return EXIT_SUCCESS;
}
```

# Transferring Ownership

- Use **reset**() and **release**() to transfer ownership
  - **release** returns the pointer, sets wrapper's pointer to NULL
  - **reset** delete's the current pointer and stores a new one

```
int main(int argc, char **argv) {
  unique_ptr<int> x(new int(5));
  cout << "x: " << x.get() << endl;

  unique_ptr<int> y(x.release());  // x abdicates ownership to y
  cout << "x: " << x.get() << endl;
  cout << "y: " << y.get() << endl;

  unique_ptr<int> z(new int(10));

  // y transfers ownership of its pointer to z.
  // z's old pointer was delete'd in the process.
  z.reset(y.release());

  return EXIT_SUCCESS;
}
```

# `unique_ptr` and STL

- `unique_ptr`s *can* be stored in STL containers
  - Wait, what? STL containers like to make lots of copies of stored objects and `unique_ptr`s cannot be copied…

- Move semantics to the rescue!
  - When supported, STL containers will *move* rather than *copy*
    - `unique_ptr`s support move semantics

# Aside: Copy Semantics

- Assigning values typically means making a copy
  - Sometimes this is what you want
    - *e.g.* assigning a string to another makes a copy of its value
  - Sometimes this is wasteful
    - *e.g.* assigning a returned string goes through a temporary copy

```
std::string ReturnFoo(void) {
  std::string x("foo");
  return x;  // this return might copy
}

int main(int argc, char **argv) {
  std::string a("hello");
  std::string b(a);  // copy a into b

  b = ReturnFoo();   // copy return value into b

  return EXIT_SUCCESS;
}
```

# Transferring Ownership via Move

- `unique_ptr` supports move semantics
  - Can "move" ownership from one `unique_ptr` to another
    - Behavior is equivalent to the "release-and-reset" combination

```
int main(int argc, char **argv) {
  unique_ptr<int> x(new int(5));
  cout << "x: " << x.get() << endl;

  unique_ptr<int> y = std::move(x); // x abdicates ownership to y
  cout << "x: " << x.get() << endl;
  cout << "y: " << y.get() << endl;

  unique_ptr<int> z(new int(10));

  // y transfers ownership of its pointer to z.
  // z's old pointer was delete'd in the process.
  z = std::move(y);

  return EXIT_SUCCESS;
}
```

# `unique_ptr` and STL Example

uniquevec.cc

```
int main(int argc, char **argv) {
  std::vector<std::unique_ptr<int> > vec;

  vec.push_back(std::unique_ptr<int>(new int(9)));
  vec.push_back(std::unique_ptr<int>(new int(5)));
  vec.push_back(std::unique_ptr<int>(new int(7)));

  //
  int z = *vec[1];
  std::cout << "z is: " << z << std::endl;

  //
  std::unique_ptr<int> copied = vec[1];

  //
  std::unique_ptr<int> moved = std::move(vec[1]);
  std::cout << "*moved: " << *moved << std::endl;
  std::cout << "vec[1].get(): " << vec[1].get() << std::endl;

  return EXIT_SUCCESS;
}
```

# References

- Learn c++
  - https://www.learncpp.com/
  - Chapter : 14


# ANY QUESTIONS?