# Creative Software Design

## 13– Exception Handling

Yoonsang Lee
Fall 2020

# Final Exam

- Time: **10:00 ~ 12:00 (may end earlier), December 8 (Tue)**
- **Monitored online exam.**
- You MUST
  - Share 2 screens to prevent cheating
    - Display screen: Share the screen you're taking the exam
    - Phone camera: A side view of you, your hands, and the display
  - Find a place to be alone
- In the lab session tomorrow, you'll have time to set this up in advance.
- Scope: Lecture 3 ~ 13
  - Assignments of this scope might be used as exam problems
- **You cannot leave your seat until the exam is over**.

# Questions from the Last Lecture

- Is it possible to use function overloading in member and non-member overloaded operators?

- : Yes. The function that better matches for the arguments is called (just like usual function overloading).

- https://stackoverflow.com/questions/22082608/operator-overloading-member-and-non-member-function-which-one-has-priority

# Today's Topics

- What are Exceptions & How to deal with Exceptions?
- C++ Exceptions: Basics
- try, catch, and throw
- Matching Catch Handlers
- Uncaught Exceptions
- Cleaning Up
- Unwinding the stack
- Course Wrap-up

# Exceptions

- Exceptions are anomalous or ***exceptional situations*** requiring special processing – often changing the normal flow of program execution[wikipedia]
  - Memory allocation error
    - out of memory space
  - Divide by zero

    -
      ```
      double x = 2.;
      double y = -2.;
      double harmonic_mean = 2.0*(x*y)/(x+y);
      ```
  - File IO error
    - Try to open an unavailable file

# How to Deal with Exceptions?

- Ignore them
  - Wrong thing to do for all but demo programs

- Abort processing
  - Detect but don’t try to recover
  -

```
double harmonic_mean(double a, double b){
    if (a == -b)
    {
            std::cout << “wrong arguments\n";
            std::abort();
    }
    return 2.0 * a * b / (a + b);
}
```

```
$ ./harmonic_mean
wrong arguments
Aborted (core dumped)
```

  - A little bit better, but still wrong for all but demo programs

# How to Deal with Exceptions?

```
bool harmonic_mean(double a, double b,
double * ans){
    if (a == -b){
        *ans = DBL_MAX;
        return false;
    }
    else{
        *ans = 2.0 * a * b / (a + b);
        return true;
    }
}
```

- **Returning error values**

```
ret = PerformTask()
If ret is 0 (or some error codes)
    Perform error processing

ret2 = PerformTask2()
If ret2 is 0 (or some error codes)
    Perform error processing
```

  - Difficult to read, modify, maintain and debug
    - Easy to miss a check
  - Impacts performance
    - Constantly spending CPU cycles looking for rare events
  - Traditional approach
    - e.g. malloc(), fopen() of C

# How to Deal with Exceptions?

- **Use C++ Exceptions**

  -
    ```
    try {
       // protected code
    } catch( ExceptionName e1 ) {
       // catch block
    }
    ```
  - More maintainable
  - More efficient: zero-cost model (popular strategy for major compilers):
    - if no exceptions are thrown, there’s NO overhead.
    - if exceptions are thrown, there’s more overhead to process them.
  - Modern approach
    - e.g. new, ifstream::open() of C++

# C++ Exceptions: Basic

```
#include <iostream>
using namespace std;

double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main () {
   int x, y;
   double z;
   cin >> x >> y;

   try {
      z = division(x, y);
      cout << z << endl;
   }
   catch (const char* msg) {
     cerr << msg << endl;
   }

   return 0;
}
```

# C++ Exceptions: Basic

```cpp
#include <iostream>
using namespace std;

double divide(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main () {
   int x, y;
   double z;
   cin >> x >> y;

   try {
      z = divide(x, y);
      cout << z << endl;
   }
   catch (const char* msg) {
      cerr << msg << endl;
   }

   return 0;
}
```

- For a normal case (e.g. y != 0),
  1. All code in the try block is executed.
  2. Catch block is skipped.
  3. Computation resumes after the catch block.

# C++ Exceptions: Basic

```
#include <iostream>
using namespace std;

double divide(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main () {
   int x, y;
   double z;
   cin >> x >> y;

   try {
      z = divide(x, y);
      cout << z << endl;
   }
   catch (const char* msg) {
      cerr << msg << endl;
   }

   return 0;
}
```

- For an exceptional case (e.g. y==0),
  1. **"Throw"** an exception.
  2. Remaining code in the try block is **skipped**.
  3. **Based on the type of the exception**, the matching catch block is executed, if found.
  4. Computation resumes after the last catch block.

# C++ Exceptions: Basic

```
void someFunc1(){
    …
    throw SomeException();  // when an exception occurs
    …
}

void someFunc2() {
    try {
       // some code that may throw an exception
        someFunc1();
    }
    catch(SomeException &e) {
       // some processing to attempt to recover from error
    }
}
```

# try, catch, and throw

```
#include <iostream>
using namespace std;

double divide(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main () {
   int x, y;
   double z;
   cin >> x >> y;

   try {
      z = divide(x, y);
      cout << z << endl;
   }
   catch (const char* msg) {
      cerr << msg << endl;
   }

   return 0;
}
```

- **try {…}:**
  - Consists of codes that may “throw” exceptions
  - Groups one or more statements (that may throw exceptions) with one or more catch blocks

# try, catch, and throw

```cpp
#include <iostream>
using namespace std;

double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main () {
   int x, y;
   double z;
   cin >> x >> y;

   try {
      z = division(x, y);
      cout << z << endl;
   }
   catch (const char* msg) {
     cerr << msg << endl;
   }

   return 0;
}
```

- **catch(E e) {…}:**
  - Catchs the exception of the given type, thrown from a *throw* statement inside try block
  - Exception type can be any built-in type or user-defined class
  - Exceptions are handled inside the catch block

# try, catch, and throw

```
#include <iostream>
using namespace std;

double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}

int main () {
   int x, y;
   double z;
   cin >> x >> y;

   try {
      z = division(x, y);
      cout << z << endl;
   }
   catch (const char* msg) {
     cerr << msg << endl;
   }

   return 0;
}
```

- **throw e:**
  - "Throw" an exception
  - Exception type can be any built-in type or user-defined class
  - Program immediately jumps to the matching catch block

# Matching Catch Handlers

- A catch handler matches an exception based on its type.

- A try block can be followed by multiple catch blocks.
  - Matching attempts are performed **in the order of catch handler declaration.**

```
try {
              // some code that may throw an exception
 }
 catch(T1 t1) {
             // processing for type T1
 }
 catch(T2 t2) {
             // processing for type T2
 }
```

```
#include <iostream>
#include <string>
using namespace std;

double divide(int a, int b) {
    if( b == 0 ) {
        throw -1;                           // "catch int"
        //throw "exception";                // "catch const char*"
        //throw string("exception");        // "catch string&"
    }
    return (a/b);
}

int main () {
    int x=2, y=0;
    double z;

    try {
        z = divide(x, y);
        cout << z << endl;
    }
    catch (int e) {
        cout << "catch int " << e << endl;
    }
    catch (const char* e) {
        cout << "catch const char* " << e << endl;
    }
    catch (string& e) {
        cout << "catch string& " << e << endl;
    }
    return 0;
}
```

# Matching Catch Handlers

- The conventional way to throw and catch exceptions is:
  - throw an exception **object**
  - catch it by **reference** (or const reference)

- A **derived class object** can be caught by **base class reference**.
  - But the opposite does not work.
  - Caution: If a derived class object is passed **by value of base class type**, *object slicing* occurs.

# Matching Catch Handlers

- **std::exception** : Base class for standard exceptions.
  - All exceptions thrown by C++ standard library are derived from this class.
  - Therefore, all standard exceptions can be caught by catching this ty pe by reference ( `catch(std::exception& e)` ).

```
#include <iostream>
using namespace std;
class ExceptionA: public std::exception { };
class ExceptionB: public ExceptionA { };

double divide(int a, int b) {
    if( b == 0 ) {
        throw ExceptionA();         // "catch ExceptionA&"
        //throw ExceptionB();       // "catch ExceptionA&"
        //throw std::exception();   // "catch std::exception&"
    }
    return (a/b);
}

int main () {
    int x=2, y=0;
    double z;

    try {
        z = divide(x, y);
        cout << z << endl;
    }
    catch (ExceptionA& e) {
        cout << "catch ExceptionA&" << endl;
    }
    catch (std::exception& e) {
        cout << "catch std::exception&" << endl;
    }

    return 0;
}
```

# Quiz #1

- Go to https://www.slido.com/
- Join #csd-hyu
- Click "Polls"

- Submit your answer in the following format:
  - **Student ID: Your answer**
  - **e.g. 2017123456: 4)**

- Note that you must submit all quiz answers in the above format to be checked as "attendance".

```
class ExceptionA : public std::exception {
        …
};
class ExceptionB : public ExceptionA {
        …
};
```

```
int main() {
  try {
    // This may throw
    // ...
  } catch (ExceptionB & e) {
    // ...
  } catch (ExceptionA& e) {
    // ...
  } catch (std::exception& e) {
    // ...
  }
  return 0;
}
```

To catch each exception types in a hierarchy:

- Most-derived type should be caught first
- Most-base type should be caught last

# Nested Try Blocks

- Try blocks can be nested.
- If a throw occurs in an inner try block, the exception moves outward through the nested try blocks until the first matching catch block is found.
  - If one of the inner catch blocks catches the exception, it will not get caught by the outer catch blocks.
  - else, it will try to find a matching one in the outer catch blocks.

```
#include <iostream>
using namespace std;
class ExceptionA: public std::exception { };
class ExceptionB: public ExceptionA { };

double divide(int a, int b) {
    if( b == 0 ) {
        throw ExceptionA();     // "catch std::exception&"
        //throw ExceptionB();     // "catch ExceptionB&"
    }
    return (a/b);
}

int main () {
    int x=2, y=0;
    double z;
    try {
        try{
            z = divide(x, y);
        }
        catch (ExceptionB& e) {
            cout << "catch ExceptionB&" << endl;
        }
        cout << z << endl;
    }
    catch (std::exception& e) {
        cout << "catch std::exception&" << endl;
    }

    return 0;
}
```

# Re-throw Exceptions

- If your catch handler does not completely handle an exception,
- you may **re-throw** it to the next outer catch blocks.

```
catch(E e)
{
  // if the processing to handle e is incomplete,
  throw;
}
```

```
#include <iostream>
using namespace std;
class ExceptionA: public std::exception { };
class ExceptionB: public ExceptionA { };

double division(int a, int b) {
    if( b == 0 ) {
        throw ExceptionB();     // "catch ExceptionB&", "catch 
std::exception&"
    }
    return (a/b);
}
int main () {
    int x=2, y=0;
    double z;

    try {
        try{
            z = division(x, y);
        }
        catch (ExceptionB& e) {
            cout << "catch ExceptionB&" << endl;
            throw;
        }
        cout << z << endl;
    }
    catch (std::exception& e) {
        cout << "catch std::exception&" << endl;
    }
    return 0;
}
```

# Uncaught Exceptions

- If there is ***no matching catch handler*** in all of the nested try block,
  - Exception is ***uncaught***
  - If an exception is uncaught, the special function **terminate**() is called

```
$ ./test
terminate called after throwing an instance of 'std::exception'
  what():  std::exception
Aborted (core dumped)
```

- Use "**catch(...)**", an *ellipsis* handler, to avoid uncaught exceptions.
  - It catches any exception not caught earlier.

# Uncaught Exceptions: Example

- If none of the catch handlers matches,
  - Exception moves to the next enclosing try block

```
void ThrowsException() {
  throw string("Exception!");
}
void CallsOne() {
  ThrowsException();
}
void CallsTwo() {
  try {
    CallsOne();
  } catch (const char* e) {
    cout << "Caught in CallsTwo\n";
  }
}
```

```
int main() {
  try {
    CallsTwo();
  }
  catch (string e) {
    cout << "Caught an exception in
main\n";
  }
  return 0;
}
```

**Output:**

```
Caught an exception in main
```

# Uncaught Exceptions: Example

- If an exception is uncaught,
  - The special function **terminate**() is called

```
void ThrowsException() {
  throw string("Exception!");
}
void CallsOne() {
  ThrowsException();
}
void CallsTwo() {
  try {
    CallsOne();
  } catch (const char* e) {
    cout << "Caught in CallsTwo\n";
  }
}
```

```
int main() {
  try {
    CallsTwo();
  }
  catch (const char* e) {
    cout << "Caught an exception in
main\n";
  }
  return 0;
}
```

**Output:**

```
terminate called after throwing an instance
of 'std::string'
```

# Uncaught Exceptions: Example

- An ellipsis handler catches all uncaught exceptions

```
void ThrowsException() {
  throw string("Exception!");
}
void CallsOne() {
  ThrowsException();
}
void CallsTwo() {
  try {
    CallsOne();
  } catch (const char* e) {
    cout << "Caught in CallsTwo\n";
  }
}
```

```
int main() {
  try {
    try {
      CallsTwo();
    }
    catch (const char* e) {
      cout << "Caught an exception in main\n";
    }
  catch(...) { cout << "An ellipsis handler catches all
uncaught exceptions" << endl; }
  return 0;
}
```

**Output:**

```
An ellipsis handler catches all uncaught
exceptions
```

# Cleaning Up

- When an exception is thrown and leaves a scope, ***destructors*** of all the objects in that scope will be called.

- Make sure that all allocated members in each object should be deallocated in its destructors.

# Cleaning Up: Example

```
class CleaningUp{
  private:
    string word;
  public:
    CleaningUp (const string & str) {
      word = str;
      cout<< "Created word:" << word <<endl;
    }
   ~CleaningUp() {
      cout<< "Destroyed word:" << word <<endl;
   }
};
void ThrowsException() {
  CleaningUp hi("HI");
  int* pi = new int;
  throw "Exception";
  delete pi;  // memory leak
  CleaningUp bye("BYE");
}
```

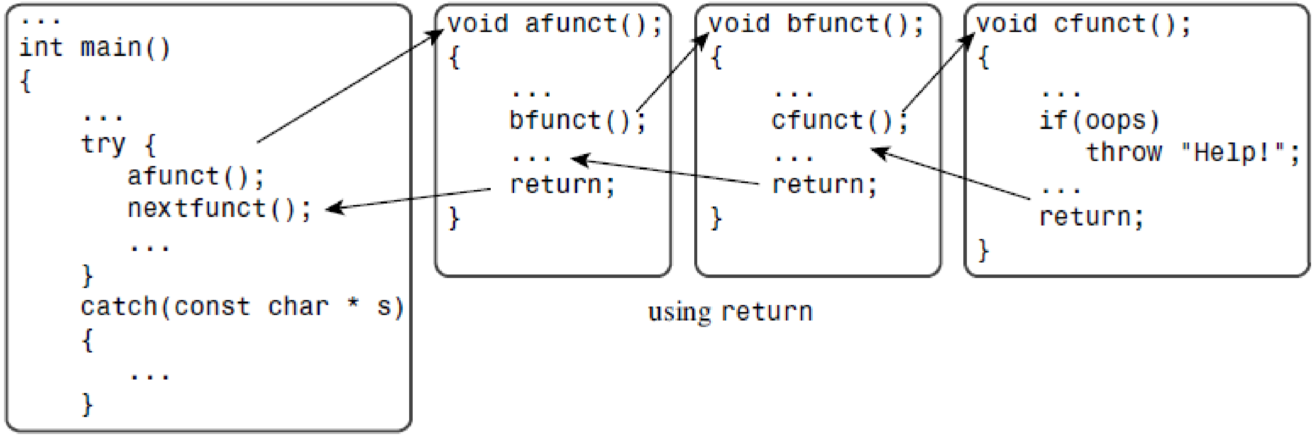
```
int main() {
    try {
      ThrowsException();
    }
    catch (const char* e) {
       cout << "Caught an exception"<<
endl;
    }
    return 0;
}
```

**Output:**

```
Created word:HI
Destroyed word:HI
Caught an exception
```

# Unwinding the stack

- return vs. throw



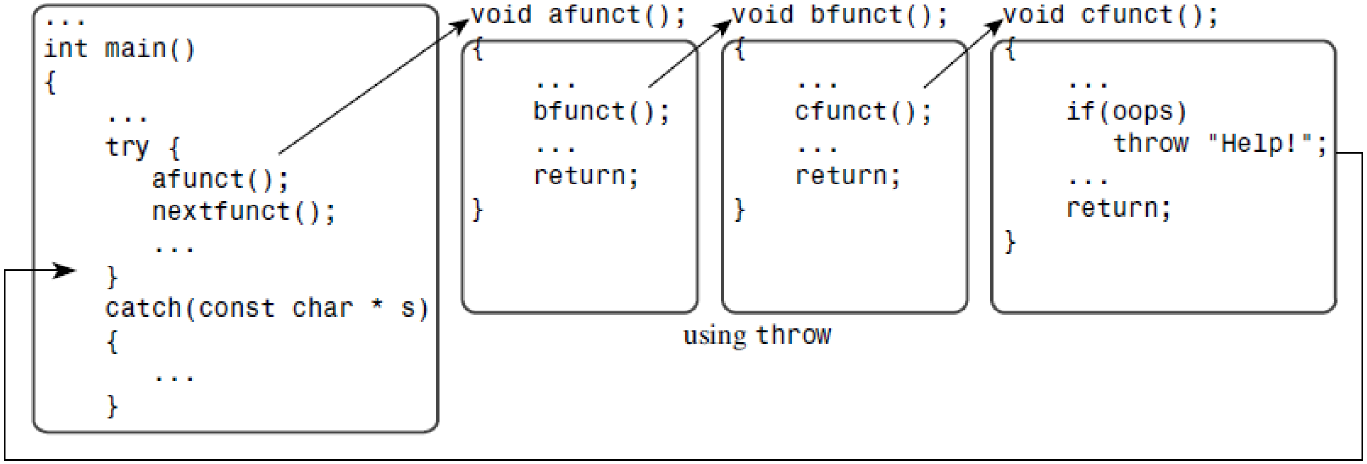
```
...
int main()
{
    ...
    try {
       afunct();
       nextfunct();
       ...
    }
    catch(const char * s)
    {
       ...
    }
```

```
void afunct();
{
    ...
    bfunct();
    ...
    return;
}
```

```
void bfunct();
{
    ...
    cfunct();
    ...
    return;
}
```

```
void cfunct();
{
    ...
    if(oops)
       throw "Help!";
    ...
    return;
}
```

using return



```
...
int main()
{
    ...
    try {
       afunct();
       nextfunct();
       ...
    }
    catch(const char * s)
    {
       ...
    }
```

```
void afunct();
{
    ...
    bfunct();
    ...
    return;
}
```

```
void bfunct();
{
    ...
    cfunct();
    ...
    return;
}
```

```
void cfunct();
{
    ...
    if(oops)
       throw "Help!";
    ...
    return;
}
```

using throw

# Unwinding the stack

- Exceptions can be propagated through several levels of function calls if there is no try-catch block

```
void ThrowsException() {
  throw string("Exception!");
}

void DoSomething() {
  cout << "DoSomething called.\n";
  ThrowsException();
  cout << “DoSomething finished\n”;
}

void DoSomethingMore() {
  cout << "DoSomethingMore called.\n";
  DoSomething();
  cout << “error in DoSomethingMore\n";
  throw string(“error");
  cout << "DoSomethingMore finished.\n";
}
```

```
int main() {
  try {
    DoSomethingMore();
  } catch (string s) {
    cout << "Caught an exception “  << s << "'" <<
endl;
  }
  cout << "All done." << endl;
  return 0;
}
```

**Output:**

```
DoSomethingMore called.
DoSomething called.
Caught an exception 'Exception!'
All done.
```

# Course Wrap-up

# Topics we covered...

- 1 - Course Intro
  - 1 - Lab1 - Environment Setting,1 - Lab2 - G++, Make, GDB
- 2 - Review of C Pointer, Const and Structure
- 3 - Differences Between C and C++
- 4 - Dynamic Memory Allocation, References
- 5 - Compilation and Linkage, CMD Args
- 6 - Class
- 7 - Standard Template Library (STL)
- 8 - Inheritance, Const & Class
- 9 - Polymorphism 1
- 10 - Polymorphism 2
- 11 – Copy Constructor, Operator Overloading
- 12 - Template
- 13 - Exception Handling

# Ending the class...

- We covered a large amount of complex C ++ content.
- I applaud your effort for all this hard work.
- Perhaps the programming language you will encounter will be easier to learn. Now, you can be proud of yourself.
- I recommend you work on larger projects with your own topics, that use 3rd-party libraries in more diverse environments.
- I hope you will continue to **enjoy** programming.

# Announcement

- **All students MUST attend the lab session tomorrow, set up for the online test, and be confirmed by the TA.**
  - If additional time is needed to set up, it will continue in the lab session on the day after tomorrow.

- If you can't make the settings right because you don't attend the lab, it will be almost impossible to take the online exam and you'll get very poor grades.

- No assignments for lecture 13. Study hard for the final exam!

Thanks for being a great class!