The University Of Michigan

EECS402 Lecture 18

Andrew M. Morgan

Savitch, Ch. 18
Exceptions
Exception Handling

Current Ways to Handle Errors

- Return a value that signifies an error
  - This value must be checked back in the calling function
  - Every time the function is called, you must check for an error being returned
  - Not only you - but other programmers using your function must remember to do the appropriate checks
- Use assert() Not expecting u to use this
  - Assertions are generally used when a condition simply "will never happen"
  - You generally write your program in a way that won't even handle these situations because you expect them never to happen
  - Instead of making a (usually bad) assumption, use assert()
  - When the condition of an assertion is false, the program aborts immediately, telling you what condition was false, and a line number

Exceptions

- C++ allows you to use a mechanism called exceptions
- Exceptions allow a more attractive method to handling errors
- Exceptions should be used to handle "exceptional circumstances" - and nothing else
- While all errors could be handled with exceptions, you should not stop using other techniques
- Using exceptions allows you to do all your error checking in appropriate places, improving the readability of your proram
- Exceptions allow you to handle an error in a different area of your program from where the error occurred
- There are three keywords associated with C++ exceptions
  - try, throw, and catch

Exception Keywords

- When a function comes across an exceptional situation, it should **throw** an exception
  - An exception that was thrown should then be handled in a specified way, so that the program can recover, or exit gracefully
- When calling a function that throws an exception, the function may complete successfully, or it may fail
  - Therefore, when you call the function, you are going to **try** it and see what happens.
  - We say the function call is placed in a **try** block
- After a try block, an exception might have been thrown
  - In order to correctly handle that exception, we must make an attempt to **catch** the exception that was thrown
  - A **catch** block should come after a try block to handle the exception

Exception Information

EECS 402

- What exactly is "thrown"?
  - Most often, you throw an object
  - You will likely define a class, which is specialized to contain information about an exception
  - Exception objects often contain an error message, and a method to print out information about the exception
  - Often, the type of object being thrown can describe the exception
- What exactly is "caught"?
  - An object of the type that might be thrown should be caught
  - You can handle the exception in whatever way is most appropriate
    - Print out a message and continue (if continuing is a possibility)
    - Print out a message and gracefully end the program
    - Do some default behavior instead
    - Etc...

Example Number 1, Catching an Exception

EECS 402

```
class IndexTooLowClass
{
  public:
    void print()
    {
      cout << "Index too low\n";
    }
};

const int MINARY = 0;
const int MAXARY = 2;
void incrAryElem(int ary[],
                 int i)
{
  if (i < MINARY)
    throw IndexTooLowClass();
  else
    ary[i]++;
}
```

what does the ary mean here

object creation

Note: This else is not necessary, since the function will end if the exception is thrown…

```
int main(void)
{
  int myAry[MAXARY + 1] = {1,5,7};
  int i;
  for (i = -1; i <= MAXARY; i++)
  {
    try
    {
      incrAryElem(myAry, i);
    }
    catch (IndexTooLowClass itl)
    {
      itl.print();
    }
    cout << "Continuing!" << endl;
  }
  return 0;
}
```

if we did not catch it successfully, the program will crash

```
Index too low
Continuing!
Continuing!
Continuing!
Continuing!
```

Catching Multiple Types Of Exceptions

- Some of the power of exceptions results from the fact that any number of user-defined exception types can be caught and handled appropriately
- There are two ways to handle multiple types

1. If the exception types are handled differently, multiple catch blocks are included after the single try block
2. If all exception types are handled the same way, there is no need to handle each individual type
   - Instead, "catch (…)" is used to indicate any type of exception thrown should be caught and handled within that catch block

Handling Multiple Types, Example 1


```
class IndexTooLowClass
{
  public:
    void print()
    {cout << "Index too low\n";}
};
class IndexTooHighClass
{
  public:
    void print()
    {cout << "Index too high\n";}
};
void incrAryElem(int ary[],
                 int i)
{
  if (i < MINARY)
    throw IndexTooLowClass();
  if (i > MAXARY)
    throw IndexTooHighClass();

  ary[i]++;
}
```

```
int main(void)
{
  int myAry[MAXARY + 1] = {1,5,7};
  int i;

  for (i = -1; i <= MAXARY + 1; i++)
  {
    try
    {
      incrAryElem(myAry, i);
    }
    catch (IndexTooLowClass itl)
    {
      itl.print();
    }
    catch (IndexTooHighClass ith)
    {
      ith.print();
    }
    cout << "Continuing!" << endl;
  }
  return 0;
}
```

```
Index too low
Continuing!
Continuing!
Continuing!
Continuing!
Index too high
Continuing!
```

Handling Events With The Catch-All


```
class IndexTooLowClass
{
  public:
    void print()
    {cout << "Index too low\n";}
};
class IndexTooHighClass
{
  public:
    void print()
    {cout << "Index too high\n";}
};
void incrAryElem(int ary[],
                 int i)
{
  if (i < MINARY)
    throw IndexTooLowClass();
  if (i > MAXARY)
    throw IndexTooHighClass();

  ary[i]++;
}
```

```
int main(void)
{
  int myAry[MAXARY + 1] = {1,5,7};
  int i;

  for (i = -1; i <= MAXARY + 1; i++)
  {
    try
    {
      incrAryElem(myAry, i);
    }
    catch (...)
    {
      cout << "Caught exception" <<
              endl;
    }
    cout << "Continuing!" << endl;
  }
  return 0;
}
```

```
Caught exception
Continuing!
Continuing!
Continuing!
Continuing!
Caught exception
Continuing!
```


Combining Exceptions Into One Class

- Since user-defined exception types are classes, they may have data and/or function members
- Allowing exception types to have descriptive attributes means that the exceptions shown previously can be combined into a single class
- Common exception class attributes include:
  - An identifier for the specific type of exception an object represents
  - A string containing the name of the function which threw the exception
  - Other information describing a specific exception object thrown

Using A Single Index Exception Class, Example


```
const int TOO_LOW_TYPE = 1;
const int TOO_HIGH_TYPE = 2;
class IndexExcepClass
{
  public:
    IndexExcepClass(int inType)
    {
       excepType = inType;
    }
    void print()
    {
      if (excepType == TOO_LOW_TYPE)
      {
        cout << "Index too low!";
      }
      else if (excepType == TOO_HIGH_TYPE)
      {
        cout << "Index too high!";
      }
    }
   private:
    IndexExcepClass()
    { ; }

    int excepType;
};
```

```
void decrAryElem(int ary[], int ind)
{
  if (ind < MINARY)
    throw IndexExcepClass(TOO_LOW_TYPE);
  if (ind > MAXARY)
    throw IndexExcepClass(TOO_HIGH_TYPE);

  ary[ind]--;
}
```

From main:

```
  for (i = -1; i <= MAXARY + 1; i++)
  {
    try
    {
      decrAryElem(myAry, i);
    }
    catch (IndexExcepClass iex)
    {
      iex.print();
      cout << endl;
    }
    cout << "Continuing!" << endl;
  }
```

```
Index too low!
Continuing!
Continuing!
Continuing!
Continuing!
Index too high!
Continuing!
```


More About Exceptions

- Exceptions will propagate upwards until a handler is found
- When a function throws an exceptions
  - It returns immediately to the calling program and looks for an appropriate catch block
  - If no catch block is found, then THAT function returns immediately, looking for an appropriate catch block
  - If no catch block is found ... ... and so on until:
    - a) An appropriate catch block is found
    - b) main() is reached and no catch block is found
      - This results in an unhandled exception error and the program aborts immediately
- This would allow you to do all your error checking in main(), with all your code in one big try block
  - If that is appropriate for your program

Example 4, Exception Propagation


```
void doOper(int ary[])
{
  try
  {
    //recall decrAryElem may
    //throw an IndexExcepClass
    decrAryElem(ary, -1);

    //recall incrAryElem may
    //throw IndexTooLowClass
    //or IndexTooHighClass
    incrAryElem(ary, 7);
  }
  catch (IndexTooHighClass ith)
  {
    cout << "From doOper: ";
    ith.print();
  }
}
```

```
int main(void)
{
  int myAry[MAXARY + 1] = {1,5,7};
  int i;

  try
  {
    doOper(myAry);
    cout << "Try complete!" << endl;
  }
  catch (IndexExcepClass iex)
  {
    cout << "From main: ";
    iex.print();
  }
  cout << "Main complete!" << endl;
  return 0;
}
```

What is the output of the above program??

Propagated Exception Output

- The output is:

  From main: Index too low!
  Main complete!

- Why is the output NOT as follows??

  From main: Index too low!
  From doOper: Index too high
  Try complete!
  Main complete!

- The reason is that when an exception is thrown, the function returns *immediately* and ceases execution, so the call to incrAryElem() is never even reached!

How Exceptions Help

- Exceptions can improve readability and understandability of your program
  - –removes the clutter of error checking in the flow your code
- Consider the code shown here, which does not utilize exceptions
  - –Compare to code to do the same thing, but using exceptions, on the next slide

```
success = myData.loadDataFromFile(inFname);
if (!success)
{
  cout << "Unable to load data from file" << endl;
}
else
{
  success = myData.convertDataFormat();
  if (!success)
  {
    cout << "Error converting data" << endl;
    cout << "  --Data: " << myData << endl;
  }
  else
  {
    success = mergeWithDatabase(myData, database);
    if (!success)
    {
      cout << "Error merging data into database" << endl;
    }
    else
    {
      success = database.doStatisticalAnalysis(stats);
      if (!success)
      {
        cout << "Statistical analysis failed!" << endl;
      }
      else
      {
        success = stats.outputToFile(outFname);
        if (!success)
        {
          cout << "Could not write output!" << endl;
        }
      }
    }
  }
}
```


How Exceptions Help, p2

- The code using exceptions isn't cluttered up with error handling
- The actual flow of useful statements is now easy to see
- "stats" can now be a return value, rather than a pass-by-reference parameter since we no longer need to return success

```
try
{
  myData.loadDataFromFile(inFname);
  myData.convertDataFormat();
  mergeWithDatabase(myData, database);
  stats = database.doStatisticalAnalysis();
  stats.outputToFile(outFname);
}
catch (SevereException ge)
{
  ge.print();
}
```