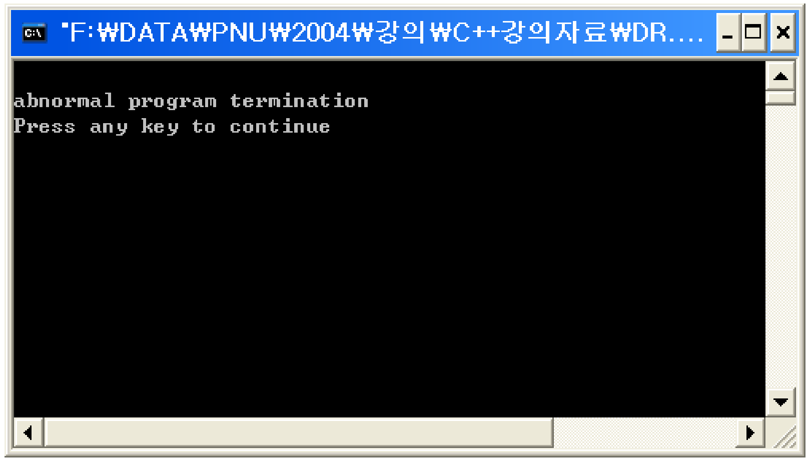
Topic 14 Exception

Exception

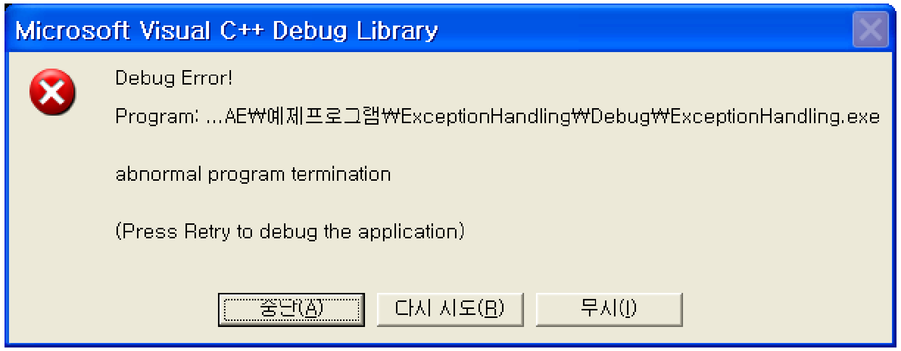
- *An exception is an event caused by some abnormal condition*

```
int divide(int x, int y) { return x / y ; }
int main() {
  int r = divide(10, 0) ;
}
```



"F:₩DATA₩PNU₩2004₩강의₩C++강의자료₩DR....

```
abnormal program termination
Press any key to continue
```



Microsoft Visual C++ Debug Library

Debug Error!

Program: ...AE₩예제프로그램₩ExceptionHandling₩Debug₩ExceptionHandling.exe

abnormal program termination

(Press Retry to debug the application)

Conventional Error Handling

- **include extensive testing of return values**
  - need lots of tests; you just add new code
  - increase the complexity of a program
  - introduce the possibility of new errors

```
int divide(int x, int y) { if ( y == 0 ) abort(-1) ; return x / y ; }
int main() {
  int r = divide(10, 0) ;
}
```

```
int divide(int x, int y) { if ( y == 0 ) return 0 ; return x / y ; }
int main() {
  int r = divide(10, 0) ;
  if ( r == 0 ) // error
```

```
int divide(int x, int y) { return x / y ; }
int main(int x, int y) {
  if ( y != 0 ) { int r = divide(10, 0) ; }
  else { /* error */ }
```

Exception Handling

- Grace solution to handling exception

```
#include <iostream>
using namespace std ;
int divide(int x, int y) {
  if ( y == 0 ) throw (“Error:divide by zero”) ;
  return x / y ;
}
int main() {
  try {
    int r = divide(10, 0) ;
    cout << r << endl ;
  } catch ( const char* const e) {
    cout << e << endl ;
  }
}
```

Exception Handling

```
void do_something() {
  if ( some error ) throw ( object of T ) ;
  //
}

void f() {
  try {
    do_something() ;
  } catch ( T e ) {
  }
}
```

A function that finds a problem that it cannot cope with throws an exception, hoping that its caller can handle the problem

A function that wants to handle that kind of problem can indicate that it is willing to catch that exception

Exception Handler

- Whenever an exception of type T is thrown, the catch block for type T is invoked

```
const int E_DIVIDEBYZERO = 10 ;
void doSomething() {
  if ( some error ) throw ( “Error” ) ;
  if ( another error ) throw (E_DIVIDEBYZERO) ;
}
int main() {
  try {
    do_something() ;
  } catch ( const char* const e ) { cout << e << endl ;
  } catch ( const int e ) { cout << e << endl ;
  }
}
```

Call Chain and Exception Handling

- The call chain is searched for a matching exception handler from the throw point up through its callers

```
int divide(int x, int y) {
   if ( y == 0 ) throw ( “Error:divide by zero”) ;
   return x / y ;
}
void f(int x, int y) {
   try { divide(x, y) ; }
   catch ( const int e ) { cout << e << endl ; }
}
int main() {
   try { f(10, 0) ; }
   catch ( const char* const e ) { cout << e ; }
}
```

Exercise

❖ What is the output of the following program ?

```
int divide(int x, int y) {
  if ( y == 0 ) throw ( “Error:divide by zero”) ;
  if ( x == 0 ) throw (-1) ;
  return x / y ;
}
void f(int x, int y) {
  try { divide(x, y) ; }
  catch ( const char* e ) { cout << e << endl ; divide(0, -1) ; }
}
int main() {
  try { f(10, 0) ; }
  catch ( const int e) { cout << e << endl ; }
}
```

Exception Class

- An object of a class can be thrown and caught.

```
# include <iostream>
using namespace std ;
class MyException {
  public:
    void print() const { cout << "Error: divide by zero₩n" ; }
} ;
int divide(int x, int y) {
  if ( y == 0 ) throw MyException() ;
  return x / y ;
}
int main() {
  try {
    int r = divide(10, 0) ;
    cout << r << endl ;
  } catch (const MyException& e) { e.print() ; }
}
```

Exception Handling in Constructor

- Exception can be used to notify errors in a constructor

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

class RangeException {
      const int value ;
      const string msg ;
   public:
      RangeException(const string& _msg, int _value) : msg(_msg), value(_value) {}
      void print() const { cout << value << " " << msg << endl ; }
} ;
```

```
class Date {
      int month, day, year ;
   public:
      Date(int m, int d, int y) {
         if ( m <= 0 || m > 12 ) throw RangeException("Invalid Month !", m) ;
         if ( d <= 0 || d > 31 ) throw RangeException("Invalid Day!", d) ;
         if ( y <= 0 ) throw RangeException("Negative year not allowed!", y) ;
         month = m ; day = d ; year = y ;
      }
      void print() const { cout << month << '.' << day << '.' << year << endl ; }
} ;

int main() {
   try {
      Date d1(5, 30, 2009) ; // no exception
      Date d2(5, 30, -10) ; // negative year
      Date d3(5, 35, 2009) ; // invalid day
      Date d4(13, 35, 2009) ; // invalid month
   }
   catch ( const RangeException& e) { e.print() ; }
}
```

Exception Class

❖ Exception object can be passed by pointer

```
class MyException {
      string msg ;
   public:
      MyException(const string& _msg) : msg(_msg) {}
      void print() const { cout << msg << endl ; }
} ;
int find(int data[], int size, int v) {
   if ( data == '₩0' ) throw MyException("The array pointer is null") ;
   if ( size <= 0 ) throw new MyException("The array size is invalid") ;
   for ( int i = 0 ;  i < size ; i ++ ) if ( data[i] == v ) return i ;
   throw MyException("Not found") ;
}
int main() {
   int iValues[] = {10, 20, 30, 40, 50} ;
   try {
      int index = find(iValues, 5, 50) ;
      index = find(iValues, 5, 60) ;
      index = find(iValues, 0, 50) ;
   }
   catch (const MyException& e) { e.print() ; }
   catch (const MyException* const e) { e->print() ; delete e ; }
}
```

Template Exception Class

```
template <class T>
class MyException {
      string msg ;
      T v ;
   public:
      MyException(const string& _msg, T _v=T()) : msg(_msg), v(_v) {}
      void print() const { cout << msg << " " << v << endl ; }
} ;
template <class T>
int find(T data[], int size, T v) {
   if ( data == '₩0' ) throw MyException<T>("The array pointer is null") ;
   if ( size <= 0 ) throw new MyException<T>("The array size is invalid") ;
   for ( int i = 0 ;  i < size ; i ++ ) if ( data[i] == v ) return i ;
   throw MyException<T>("Not found", v) ;
}
int main() {
   int iValues[] = {10, 20, 30, 40, 50} ;
   try {
      int index = find(iValues, 5, 60) ;
      index = find(iValues, 0, 50) ; index = find((int*) 0, 5, 50) ;
   }
   catch (const MyException<int>& e) { e.print() ; }
   catch (const MyException<int>* const e) { e->print() ; delete e ; }
}
```

Cleaning Exception Objects

- Exception objects should be cleaned.

```
int main() {
   Date* d1=0, *d2=0 ; // what if d1 and d2 not initialized with 0
   try {
      d1 = new Date(11, 11, 2004) ;
      d2 = new Date(15, 11, 2004) ;
      d1->print() ;
      d2->print() ;
      delete d1 ;
      delete d2 ;
   }
   catch ( const RangeException& e) {
      e.print() ;
      delete d1 ;
      delete d2 ;
   }
}
```

Inheritance and Exception

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

class MyException {
   const string msg ;
public:
   MyException(const string& _msg) : msg(_msg) {
      cout << "MyException created! " << msg << endl ;
   }
   void print() const { cout << msg << endl ; }
} ;
```

```
class Base {
protected: int value ;
public:
   Base(int n) : value(n) {
      cout << "Base constructor for " << value << endl ;
      if ( value < 0 ) throw MyException("Negative") ;
   }
   ~Base() { cout << "Base destructor for " << value << endl ; }
} ;
class Derived: public Base {
public:
  Derived(int n) : Base(n) {
    cout << "Derived constructor for " << value << endl ;
    if ( value > 100 ) throw MyException("Too Big") ;
  }
  ~Derived() { cout << "Derived destructor for " << value << endl ; }
} ;

int main() {
   try {
      Derived one(999) ;
   } catch (const MyException& e) { e.print() ; }
   try {
      Derived two(-1) ;
   } catch (const MyException& e) { e.print() ; }
}
```

```
Base constructor for 999
Derived constructor for 999
MyException created! Too Big
Base destructor for 999
Too Big
Base constructor for -1
MyException created! Negative
Negative
```

Exception Hierarchy

- Exception classes can be organized into a hierarchy

```
class RangeException {
        const int value ;
        const string msg ;
    public:
        RangeException(const string& _msg, int _value) : msg(_msg), value(_value) {}
        void print() const { cout << value << " " << msg << endl ; }
} ;
class MonthRangeException : public RangeException {
public:
    MonthRangeException(const string& m, int v) : RangeException(m, v) {}
} ;
class DayRangeException : public RangeException {
public:
    DayRangeException(const string& m, int v) : RangeException(m, v) {}
} ;
class YearRangeException : public RangeException {
public:
    YearRangeException(const string& m, int v) : RangeException(m, v) {}
} ;
```

```
class Date {
    int month, day, year ;
  public:
    Date(int m, int d, int y) { setDate(m, d, y) ; }
    void setDate(int m, int d, int y) {
       if ( m <= 0 || m > 12 ) throw MonthRangeException("Invalid Month !", m) ;
       if ( d <= 0 || d > 31 ) throw DayRangeException("Invalid Day!", d) ;
       if ( y <= 0 ) throw YearRangeException("Negative year not allowed!", y) ;
       month = m ; day = d ; year = y ;
    }
    void print() const { cout << month << '.' << day << '.' << year << endl ; }
} ;

int main() {
  try {
    Date d(1, 20, -10) ;
  }
  catch (const MonthRangeException& e) { cout << "Error in the month: "; e.print(); }
  catch (const DayRangeException& e) { cout << "Error in the day: " ; e.print() ; }
  catch (const YearRangeException& e) { cout << "Error in the year: " ; e.print() ; }
}
```

Exception Hierarchy

- By an exception class, its any derived exception can be caught!

```
int main() {
  try {
    Date d(1, 20, -10) ;
  }
  catch (const MonthRangeException& e) { cout << "Error in the month:"; e.print();}
  catch (const DayRangeException& e) { cout << "Error in the day: " ; e.print() ; }
  catch (const YearRangeException& e) { cout << "Error in the year: " ; e.print() ; }
}
```

```
int main() {
  try {
    Date d(1, 20, -10) ;
  }
  catch (const RangeException& e) { cout << "Error in the date: "; e.print(); }
}
```

Exception Hierarchy

- Thus, the order of handlers are important!
- What is different with the two programs ?

```
int main() {
  try { Date d(1, 20, -10) ; }
  catch (const MonthRangeException& e) { cout << "Error in the month:"; e.print();}
  catch (const DayRangeException& e) { cout << "Error in the day: " ; e.print() ; }
  // YearRangeException caught by RangeException
  catch (const RangeException& e) { cout << "Error in the date: "; e.print(); }
}
```

```
int main() {
  try { Date d(1, 20, -10) ; }
  catch (const RangeException& e) { cout << "Error in the date: "; e.print(); }
  // the following codes never executed !
  catch (const MonthRangeException& e) { cout << "Error in the month:"; e.print();}
  catch (const DayRangeException& e) { cout << "Error in the day: " ; e.print() ; }
  catch (const YearRangeException& e) { cout << "Error in the year: " ; e.print() ; }
}
```

Default Handler

❖ The default handler can catch any type of exceptions

```
catch(const string& message)
{
   cerr << message << endl;
   exit(1);
}
catch( ... )          // default action to be taken
{
   cerr << "THAT'S ALL FOLKS." << endl;
   abort();
}
```

Exception Specification

- An Exception Specification Syntax

  ***FunctionHeader* throw (*TypeList*)**

  - The *TypeList* is the list of types that a throw expression within the function can have.
  - Examples
    - void Date::setDate(int, int, int)

      throw(DayRangeException,
      MonthRangeException,YearRangeException) ;
    - void noexception(int i) throw() ; // no exception thrown
    - void throwAnyException(int i) ; // can throw any exception
- System-Provided Handlers
  - **`unexpected():`** called when an exception specification is violated; that is, undeclared exceptions are thrown
  - **`terminate():`** called when no handler has been provided to deal with an exception.
- unexpected() → terminate() → abort()

Standard Exceptions

- Some standard exceptions are thrown **by language**:

name	thrown by
bad_alloc	new
bad_cast	dynamic_cast
bad_typeid	typeid
bad_exception	exception specification

- And, some are thrown by **standard library functions**:

name	thrown by
out_of_range	at() bitset<>::operator[]()
invalid_argument	bitset<> constructor
overflow_error	bitset<>::to_ulong()
io_base::failure	io_base::clear

Standard Exceptions: bad_alloc

```
#include <iostream>
#include <exception>     // standard exceptions here
using namespace std;
int main() {
   int  *p, n = 1000000, m = 0;
   try {
      while ( true ) {
         p = new int[n];
         m++;
      }
   }
   catch (bad_alloc) {
      cerr << "bad_alloc after allocating “ << m << "M ints" << endl;
   }
   catch (...) { cerr << "default catch" << endl; }
}
```

Assertions

- Assertions
  - **ensures certain conditions hold at some program points**
  - precondition: assertions on input
  - postcondition: assertions on output
- The assert macro
  - provided by the standard <cassert> library
  - syntax: assert(*expression*)
  - If the *expression* evaluates as false, execution is aborted with diagnostic output.
  - **The assertions are discarded if the macro NDEBUG is defined.**

Assertions: An Example

```
# include <cassert>

void swap(int a, int b) { int temp = a; a = b; b = temp; }

void bubble(int a[], int size) {
   for ( int i = 0; i != size - 1; ++i)
      for ( int j = i ; j != size - 1; ++j) {
         if (a[j] < a [j + 1]) swap(a[j], a[j + 1]);
         assert(a[j] >= a[j+1]);
      }
}
int main() {
   int  t[10] = { 9, 4, 6, 4, 5, 9, -3, 1, 0, 12};
   bubble(t, 10);
   for (int i = 0; i < 10; ++i) {
      assert(i == 9 || t[i] >= t[i+1]);
   }
}
```

Usage of Assertions

- Assertions are often used for specifying pre/post conditions of a function

```
void withdraw(int& balance, int amount) {
   // precondition
   assert (amount > 0 && amount >= balance ) ;

   int _balance = balance ;

   …
   …

   // post condition
   assert ( balance = _balance – amount ) ;
}
```

- Exception handling vs Assertions ?

Exercise

- Try to improve CharStack by providing class StackException.

```
class StackException {
   const string msg ;
   public:
      StackException(const string& msg) ;
      void print() const { cout << msg << endl ; }
} ;
```

```
class CharStack {
   int size ;
   int top ;
   char* s ;
  public:
      CharStack(int sz) { top = 0 ; s = new char[size=sz]; }
      // destructor, copy constructor, and assignment operator
      void push(char c) { s[top++] = c ; }
      char pop() { char r = s[--top]; s[top] = '₩0' ; return r ; }
      void print() const {
         for ( int i = 0 ; i < top ; i ++ ) cout << s[i]  ;
         cout << endl ;
      }
} ;
```