Exception Handling

Jiun-Long Huang
National Chiao Tung University

Introduction

- Upon detecting a problem that cannot be handled locally, the program could:
  - terminate the program,
  - return a value representing "error,"
    - NAN, NULL…
  - return a legal value and leave the program in an illegal state, or
    - errno
  - call a function supplied to be called in case of "error."

- Description of double sqrt(double x);

  RETURN VALUES
  The sqrt() functions returns the requested square root unless an error occurs. An attempt to take the sqrt() of negative x raises an invalid exception and causes an NaN to be returned.

```
#include <iostream>
#include <cmath>
using namespace std;
int main()
{
  double result;
  result=sqrt(-1.0);
  if (isnan(result))
    cout << true;
  else
    cout << false;
  if (isnan(result))
    cout << true;
  else
    cout << false;
};
```

Exception handling →
Aspect-oriented programming

- An exception is an occurrence that causes abnormal program termination.
- These abnormal occurrences may be caused by
  - hardware failures (e.g. insufficient resources, or errors in peripheral devices)
  - user (garbage data input).
    - a division by 0,
    - an even root of a negative number,
    - insufficient memory in the heap…

- When designing programs used in critical applications such as nuclear power plants, medical devices, or aircraft guidance systems, it is important that these programs are robust and not prone to errors.
- Programs must therefore efficiently process (handle) exceptions.

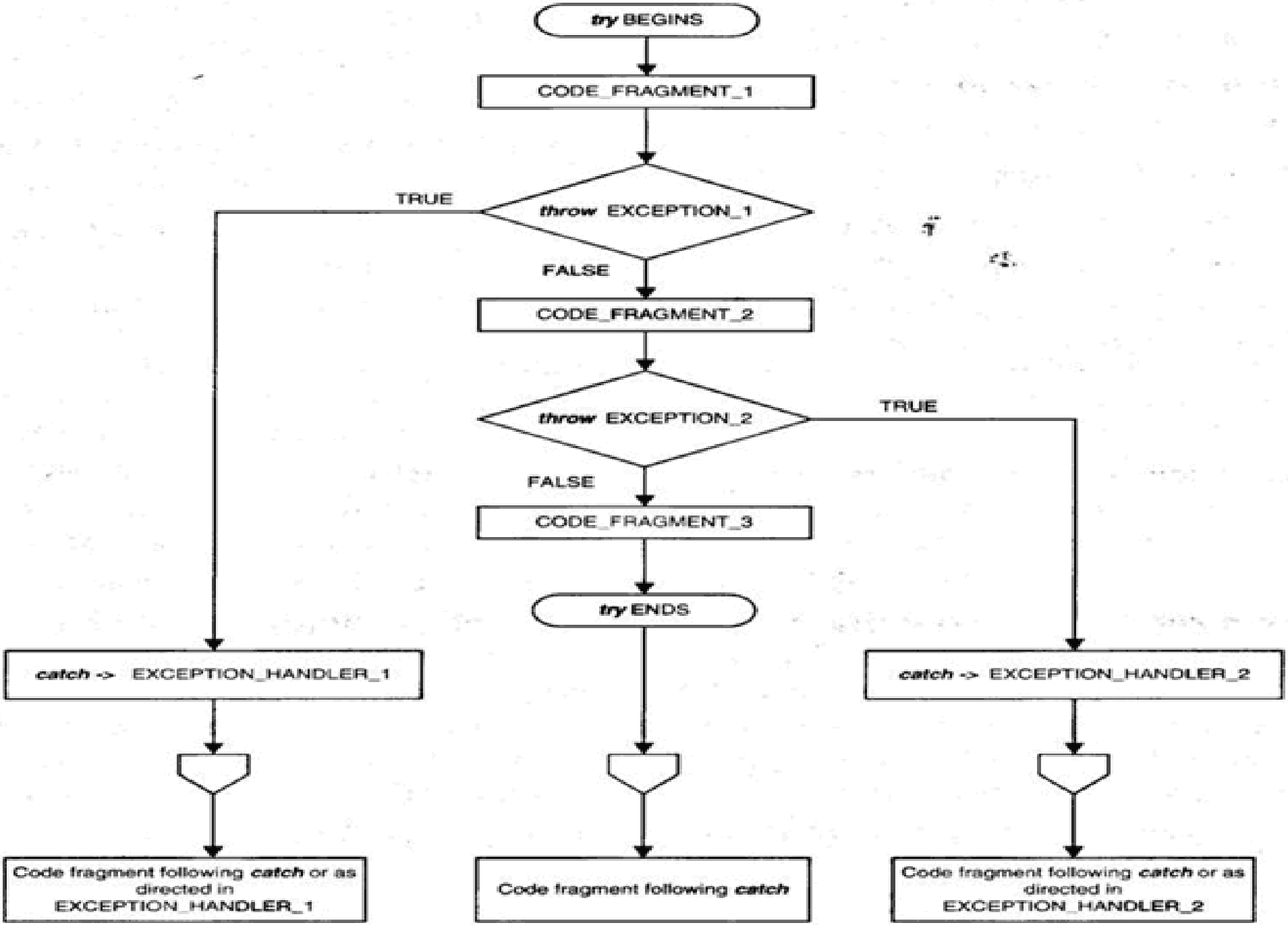
- A function that finds a problem it cannot cope with throws an exception, hoping that its (direct or indirect) caller can handle the problem.
  - An exception can be also defined as an object that is thrown at the location in a program where a run time error occurs.
- This object must be caught and handled by code called the exception handler, which is designed to process that particular type of run time error.

- When processing exceptions, the following two tasks must be successfully completed:
  - Detecting and throwing (raising) an exception
    - try, throw
  - Handling an exception
    - catch

- C++ provides a structure that uses the try, throw, and catch keywords to process run time errors.

```
try {
  //code fragment
  throw exception_1;
  //code fragment
  throw exception_2;
...
}
catch(exception_1) {
  //exception handler
}
catch(exception_2) {
  //exception handler
}
...
```

try-throw-catch FLOW CHART



1. `exception_1` occurs.

   The code sequence that executes:

   - `code_fragment_1`
   - `exception_handler_1`

2. `exception_2` occurs (but not `exception_1`).

   The code sequence that executes:

   - `code_fragment_1`
   - `code_fragment_2`
   - `exception_handler_2`

3. None of the exceptions occurs.

   The code sequence that executes:

   - `code_fragment_1`
   - `code_fragment_2`
   - `code_fragment_3`

- Throwing an exception transfers control to an exception handler within a catch block.
  - This process commonly involves passing an object from the point where the exception occurs to the exception handler code, which can use that object to process the exception.
- A selection of a catch block contains appropriate exception handler code based on the type of the object passed (thrown).

- The type of the object thrown by throw must match the type of the argument in one of the catch statements following the try block, or the exception will not be handled.
- Throwing an unhandled automatically invokes the terminate() function, which terminates the program by calling the abort() function.

```
void f()
{
  try {
    throw E();
  }
  catch(H) {
    // when do we get here?
  }
}
```

- The handler is invoked:
  - If H is the same type as E .
  - If H is an unambiguous public base of E .
  - If H and E are pointer types and [1] or [2] holds for the types to which they refer.
  - If H is a reference and [1] or [2] holds for the type to which H refers.

```
class Exception {
public:
  Exception(int, float, char);      //class constructor
};

float fun(int);
int a;
             //Examples of throw statements:
throw a;    //throws an integer
throw 1;    //throws an integer constant
throw a*0.1;      //throws a floating point value
throw "string";   //throws a string constant
//throws an object of Exception
throw Exception(1,3.3,'E');
throw fun(a);     //throws a value returned by fun( )
throw;      //re-throws exception
```

```
#include <iostream>
#include <cmath>
using namespace std;
double roots(int n, double a)  //returns the nth root of a
number
{
   double e=(double)1/n;
   return pow(a,e);
}
int main()
{
  int n;
  double a, b;
  char c;
  do {
    cout<<"Enter root => ";
    cin>>n;
    cout<<"Enter number => ";
    cin>>a;
```

```
try { //try block begins
  if(n==0)              //division by 0
    throw 1;            //throws an integer constant
  //even root from a negative number
  if((n%2==0)&&(a<0))
    //throws a string constant
   //ERROR: throw "an even root of a negative number!";
    throw string("an even root of a negative number!");
  //odd root from a negative number
  if((n%2!=0)&&(a<0)) {
    b=roots(n,-a);
    cout<<n<<". root of "<<a<<" is "<<-b<<endl;
  }
  else { //nth root from a positive number
    b=roots(n,a);
    cout<<n<<" root of "<<a <<" is "<<b<<endl;
  }
} //try block ends
```

```
    catch (int x) { //handles an integer exception
      cout<<"Exception # "<<x<<" occurred!"<<endl;
      cout<<"Cannot be 0 root!"<<endl;
    }
    // ERROR: catch (char *exc2) {
    catch (string exc2) { //handles a string exception
      cout<<"Exception # 2"<<" occurred!"<<endl;
      cout<<"Cannot be "<<exc2<<endl;
    }
    cout<<"\tTo exit enter X or any key to continue => ";
    cin>>c;
  } while (c!='X');
  return 0;
}
```

```
Enter root => 2
Enter number => 4
2. root of 4 is 2
        To exit enter X or any key to continue => a
Enter root => 0
Enter number => 8
Exception # 1 occurred!
Cannot be 0 root!
        To exit enter X or any key to continue => X
```

```
#include <iostream>
#include <cmath>
using namespace std;
//function that throws exceptions
double roots(int n, double a)
{
  if(n==0)              //division by 0
    throw 1;            //throws an integer constant
  if((n%2==0)&&(a<0)) //even root from a negative number
    throw string("an even root and a negative number!");
                        //throws a string constant
  double e=(double)1/n;
  return pow(a,e);
}
int main()
{
  int n;
  double a,b;
  char c;
  do {
    cout<<"Enter root => ";
    cin>>n;
    cout<<"Enter number => ";
    cin>>a;
```

```
    try { //try block begins
      //an odd root of a negative number
      if((n%2!=0)&&(a<0)) {
        b=roots(n,-a);
        cout<<n<<". root of "<<a<<" is "<<-b<<endl;
      }
      else { //a root of a positive number
        b=roots(n,a);
        cout<<n<<". root of "<<a<<" is "<<b<<endl;
      }
    } //try block ends
    catch (int x) { //catches an integer exception
      cout<<"Exception # "<<x<<" occurred!"<<endl;
      cout<<"Cannot be the 0th root!"<<endl;
    }
    catch (string exc2) { //catches a string exception
      cout<<"Exception # 2"<<" occurred!"<<endl;
      cout<<"Cannot be "<<exc2<<endl;
    }
    cout<<"\tTo exit enter X or any key to continue => ";
    cin>>c;
  } while (c!='X');
  return 0;
}
```

- After an exception is thrown from a function, the function terminates.
- Use exception handling to process exceptions generated in library functions in order to prevent certain types of run time errors when using these functions.
- Throwing an exception transfers control from a try block to an exception handler. This action results in the destruction of each variable declared in the try block.

- C++ provides a special kind of a catch statement that can be used to catch an exception of any type.

```
//default exception handler
catch (...) {
...
}
```

- A substitution of an argument declaration of a specific type by ellipses within a catch statement enables using a single catch statement for exceptions of any type.

```
#include <iostream>
using namespace std;
void temperature(int t) {
  try {
    if(t==100)    throw string("BOILING");
    else if(t==0) throw string("FREEZING");
    else if((t<-100)||(t>200)) throw 0.1;
    else throw t;
  } //end of try block
  catch(string str) { cout<<str<<endl;   }
  catch(int x) {
    cout<<"Temperature = "<<x<<" deg C"<<endl; }
  catch(...) { cout<<"Incorrect value!"; }
}
int main() {
  temperature(0);
  temperature(47);
  temperature(100);
  temperature(-155);
  return 0;
}
```

```
FREEZING
Temperature = 47 deg C
BOILING
Incorrect value!
```

Exception Specifications

- C++ uses an exception specification list to restrict types of exceptions that can be thrown directly or indirectly from a function.

```
void fun(int) throw(double, char *); //function prototype
//function pointer declaration with an exception
//specification
void (*ptr)() throw (int);
```

- An exception specification can appear only on a function declaration/definition and a function pointer (or reference) declaration,

- If a function throws an exception that does not belong to a specified type, function unexcepted is called.
- Placing throw() (an empty exception specification) after a function's parameter list to indicate that the function does not throw exceptions.

```
void myUnexpected()
{
  //Code that handles unexpected exceptions
}
int main()
{
  // Sets up my Unexpected( ) as a handler of
  // unexpected exceptions
  set_unexpected (myUnexpected);
}
```

```
// set_unexpected example
#include <iostream>
#include <exception>
using namespace std;
void myunexpected () {
  cerr << "unexpected called\n";
  exit(-1);
}
void myfunction () throw (int) {
  // throws char (not in exception-specification)
  throw 'a';
}
int main (void) {
  set_unexpected (myunexpected);
  myfunction();
  return 0;
}
```

```
unexpected called
```

Handling Memory Allocation Errors

- The new operator, however, may fail to successfully allocate memory (insufficient resources), thus causing a run time error.
- A simple method to handle a memory allocation error is to check for a null pointer returned by new when memory allocation fails.

```
double *dptr = new double[100];
//Allocates 100 double variables
if (dptr==0) //checks for a null pointer
{
  cout<<"Memory allocation error";
  exit(1);
}
```

User Defined Recovery Routine

- C++ enables programmers to design their own recovery routine (a user defined function) to handle memory allocation errors.
- To register a user defined function as a default handler of new failures, the set_new_handler() function (defined in the new header file) is called.

```
set_new_handler(my_handler);
```

- A call to set_new_handler() with the 0 argument removes the current new handler, resulting in a program without a default new handler.
- The set_new_handler() function takes a function pointer as an argument. A function pointed to by the pointer will automatically be invoked each time new fails.

```
#include <iostream>
#include <new>
using namespace std;
const int n=200;
const int size=20;
void memError()  //user-defined new handler
{
   cout<<"Memory allocation error!"<<endl;
   cout<<"Program will terminate!";
   exit(1);
}
//defines a type of 1000000 bytes
typedef char Mbyte[1000000];
int main()
{
  set_new_handler(memError);  //Sets new handler
  Mbyte *ptr[n];              //an array of pointers
  for(int i=0; i<n;i++)
  {
    ptr[i]=new Mbyte[size];
    cout<<((i+1)*20)<<" Mb allocated!"<<endl;
   }
   return 0;
}
```

```
1900 Mb allocated!
1920 Mb allocated!
1940 Mb allocated!
1960 Mb allocated!
1980 Mb allocated!
2000 Mb allocated!
2020 Mb allocated!
2040 Mb allocated!
2060 Mb allocated!
2080 Mb allocated!
Memory allocation error!
Program will terminate!
```