BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

Programming and Computer Applications-2

Exception Handling

Instructor : PhD, Associate Professor Leyla Muradkhanli

Exception Handling

- Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.

- In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from std::exception class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

Exception Handling Keywords

Exception handling in C++ consist of three keywords: **try**, **throw** and **catch**:

The **try** statement allows you to define a block of code to be tested for errors while it is being executed.

The **throw** keyword throws an exception when a problem is detected, which lets us create a custom error.

The **catch** statement allows you to define a block of code to be executed, if an error occurs in the try block.

The **try** and **catch** keywords come in pairs:

Exception Handling Keywords

```
try {
  // Block of code to try
  throw exception; // Throw an exception when a problem arise
}
catch () {
  // Block of code to handle errors
}
```

C++ try/catch

- In C++ programming, exception handling is performed using try/catch statement. The C++ try block is used to place the code that may occur exception. The catch block is used to handle the exception.

C++ example without try/catch

```
#include <iostream>
using namespace std;
float division(int x, int y) {
   return (x/y);
}
int main () {
   int a = 50;
   int b = 0;
   float c;
      c = division(a, b);
      cout << c << endl;
   return 0;
}
```

C++ try/catch example

```
#include <iostream>
using namespace std;
float division(int x, int y) {
   if( y == 0 ) {
      throw "Attempted to divide by zero!";
   }
   return (x/y);
}
int main () {
   int a = 25;
   int b = 0;
   float c;
   try {
      c = division(a, b);
      cout << c << endl;
   }catch (const char* ex) {
      cout << ex << endl;
   }
   return 0;
}
```

Output :

Attempted to divide by zero!

C++ try/catch example

```
#include <iostream>
using namespace std;
int main()
{
    int a=10, b=0, c;
    // try block activates exception handling
    try
    {
        if(b == 0)
        {
            // throw custom exception
            throw "Division by zero not possible";
            c = a/b;
        }
    }
    catch(const char* ex) // catches exception
    {
        cout<<ex<<endl;
    }
    return 0;
}
```

Output :

Division by zero not possible

Exception Handling

```
#include <iostream>
using namespace std;
int main()
{
    int x = -1;

    // Some code
    cout << "Before try \n";
    try {
        cout << "Inside try \n";
        if (x < 0)
        {
            throw x;
            cout << "After throw (Never executed) \n";
        }
    }
    catch (int x ) {
        cout << "Exception Caught \n";
    }

    cout << "After catch (Will be executed) \n";
    return 0;
}
```

Output :

Before try

Inside try

Exception Caught

After catch (Will be executed)

Using Multiple catch blocks

```
try {
  // code here
}
catch (int param) { cout << "int exception"; }
catch (char param) { cout << "char exception"; }
catch (...) { cout << "default exception"; }
```

Using Multiple catch blocks

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

int main()
{
    try  {
       throw 10;
    }
    catch (char *excp)  {
        cout << "Caught " << excp;
    }
    catch (...)  {
        cout << "Default Exception\n";
    }
    return 0;
}
```

Output :

Default Exception

Using Multiple catch blocks

```
#include <iostream>
using namespace std;
int main()
{
    try  {
       throw 'a';
    }
    catch (int x)  {
        cout << "Caught " << x;
    }
    catch (...)  {
        cout << "Default Exception\n";
    }
    return 0;
}
```

Output :

Default Exception

Using Multiple catch blocks

```
#include <iostream>
using namespace std;
int main()
{
    int x[2] = {-1,2};
    for(int i=0; i<2; i++)
    {
        int ex = x[i];
        try
        {
            if (ex > 0)
                // throwing numeric value as exception
                throw ex;
            else
                // throwing a character as exception
                throw 'x';
        }
```

Using Multiple catch blocks

```
catch (int ex)  // to catch numeric exceptions
        {
            cout << "Integer exception\n";
        }
catch (char x) // to catch character/string exceptions
        {
            cout << "Character exception\n";
        }
    }
}
```

Output :

Character exception

Integer exception

Generalized catch block in C++

Below program contains a generalized catch block to catch any uncaught errors/exceptions.
catch(...) block takes care of all type of exceptions.

```
#include <iostream>
using namespace std;
int main()
{
    int x[2] = {-1,2};
    for(int i=0; i<2; i++)
    {
        int ex=x[i];
try
        {
            if (ex > 0)
                throw ex;
            else
                throw 'ex';
        }
        // generalised catch block
        catch (...)
        {
            cout << "Special exception\n";
        }
    }
return 0;
}
```

Output :

Special exception

Special exception

Nest try/catch blocks

```
try {
  try {
      // code here
  }
  catch (int n) {
      throw;
  }
}
catch (...) {
  cout << "Exception occurred";
}
```

Example

```
#include<iostream>
using namespace std;
int main()
{
    try {
        throw 6;
    }
    catch (int a) {
        cout << "An exception occurred!" << endl;
        cout << "Exception number is: " << a << endl;
    }
}
```

Output :

An exception occurred!

Exception number is: 6

When to Use Exception Handling

- Exception handling is designed to process synchronous errors, which occur when a statement executes, such as out-of-range array subscripts, arithmetic overflow (i.e., a value outside the representable range of values), division by zero, invalid function parameters and unsuccessful memory allocation (due to lack of memory).
- Exception handling is not designed to process errors associated with asynchronous events (e.g., disk I/O completions, network message arrivals, mouse clicks and keystrokes), which occur in parallel with, and independent of, the program's flow of control.

Rethrowing an Exception

- It's possible that an exception handler, upon receiving an exception, might decide either that it cannot process that exception or that it can process the exception only partially.
- In such cases, the exception handler can defer the exception handling (or perhaps a portion of it) to another exception handler.
- In either case, you achieve this by rethrowing the exception via the statement
    - `throw;`
- Regardless of whether a handler can process an exception, the handler can rethrow the exception for further processing outside the handler.
- The next enclosing `try` block detects the rethrown exception, which a `catch` handler listed after that enclosing `try` block attempts to handle.

Rethrowing an Exception (cont.)

The next program demonstrates rethrowing an exception.

```
// Demonstrating exception rethrowing.
#include <iostream>
#include <exception>
using namespace std;

void throwException()
{
try
   {
      cout << "Function throwException throws an exception\n";
      throw exception(); // generate exception
   }
catch (exception &) // handle exception
   {
      cout << "Exception handled in function throwException"
         << "Function throwException rethrows exception";
      throw; // rethrow exception for further processing
   }
   cout << "This also should not print\n";
}
```

```cpp
int main()
{
   // throw exception
   try
   {
      cout << "main invokes function throwException\n";
      throwException();
      cout << "This should not print\n";
   }
   catch (exception &) // handle exception
   {
      cout << "Exception handled in main\n";
   }
   cout << "Program continues after catch in main\n";

}
```

Output:

```
main invokes function throwException
   Function throwException throws an exception
   Exception handled in function throwException
   Function throwException rethrows exception

Exception handled in main
Program control continues after catch in main
Press any key to continue . . .
```

Stack Unwinding

The process of removing function entries from function call stack at run time is called **Stack Unwinding.**

In C++, when an exception occurs, the function call stack is linearly searched for the exception handler, and all the entries before the function with exception handler are removed from the function call stack. So exception handling involves Stack Unwinding if exception is not handled in same function (where it is thrown).

Stack Unwinding

- When an exception is thrown but not caught in a particular scope, the function call stack is “unwound,” and an attempt is made to `catch` the exception in the next outer `try`…`catch` block.
- Unwinding the function call stack means that the function in which the exception was not caught terminates, all local variables in that function are destroyed and control returns to the statement that originally invoked that function.
- If a `try` block encloses that statement, an attempt is made to `catch` the exception.
- If a `try` block does not enclose that statement, stack unwinding occurs again.
- If no `catch` handler ever catches this exception, function `terminate` is called to terminate the program.
- The program of Fig. 4 demonstrates stack unwinding.

```
#include <iostream>
#include <exception>
using namespace std;
void f1()
{
cout<<"f1() start \n";
throw 10;
cout<<"f1() end ";
}
void f2() throw
{
cout<<"f2() start \n ";
f1();
cout<<"f2() end ";
}
```

```
void f3() {
cout<<"f3() start \n ";
try {
f2();
}
catch (int i) {
cout<<"Caught exception "<<i<<endl;
}
cout<<"f3() end \n";
}
int main()
{
f3();
return 0;
}
```

Output:

```
f3() start
f2() start
f1() start
Caught exception 10
f3() end
Press any key to continue . . .
```

Stack Unwinding (cont.)

In the above program, when f1() throws exception, its entry is removed from the function call stack (because it f1() doesn't contain exception handler for the thrown exception), then next entry in call stack is looked for exception handler. The next entry is f2(). Since f2() also doesn't have handler, its entry is also removed from function call stack. The next entry in function call stack is f3(). Since f3() contains exception handler, the catch block inside f3() is executed, and finally the code after catch block is executed.

Note that the following lines inside f1() and f2() are not executed at all.

```
//inside f1()
  cout<<"f1() end ";
//inside f2()
  cout<< "f2() end ";
```

Standard Library Exception Hierarchy

- The C++ Standard Library includes a hierarchy of exception classes, some of which are shown in Fig. 5.
- This hierarchy is headed by base-class `exception` (defined in header file `<exception>`), which contains `virtual` function `what`, which derived classes can override to issue appropriate error messages.

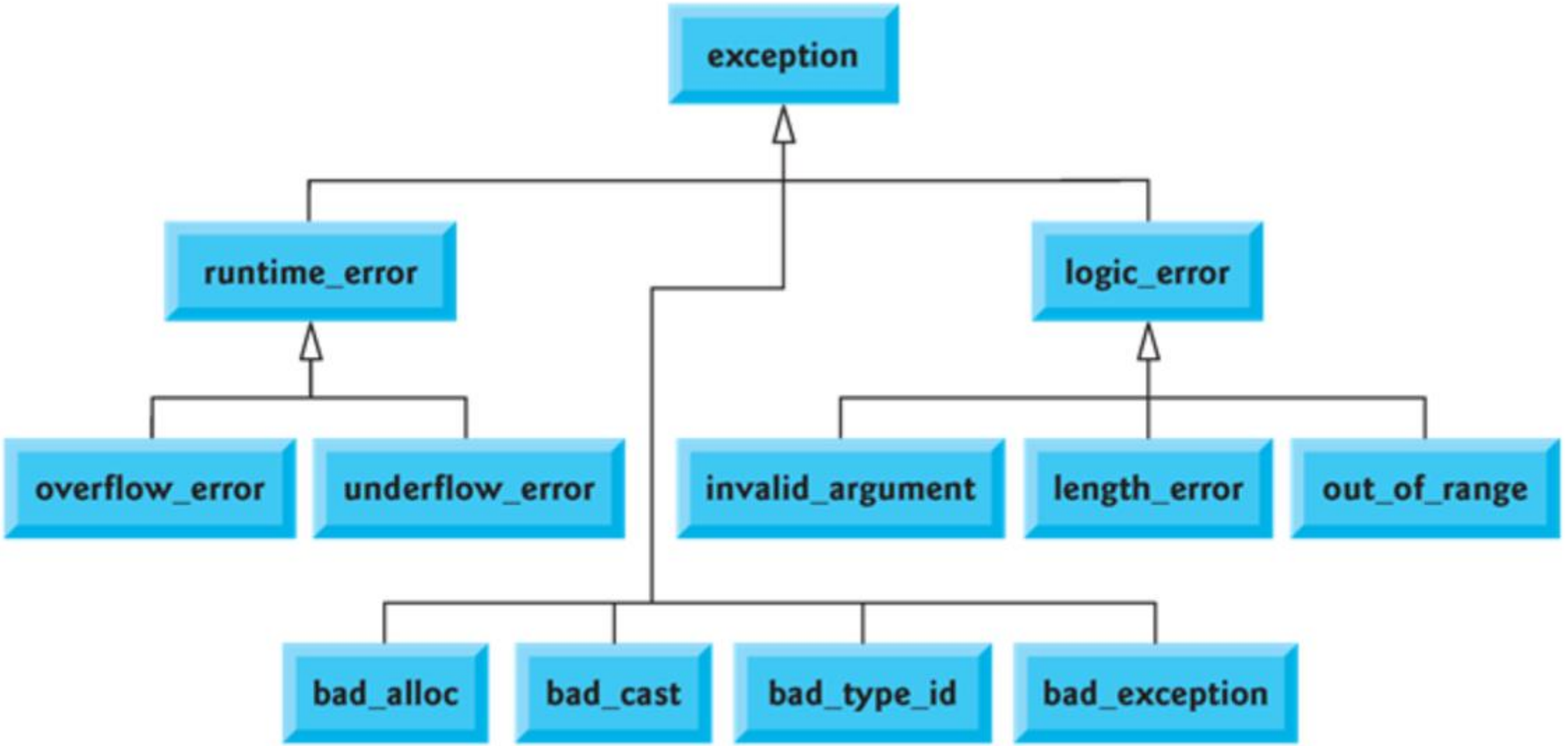

Fig. 5 | Some of the Standard Library exception classes.

Standard Library Exception Hierarchy (cont.)

- Immediate derived classes of base-class `exception` include **runtime_error** and **logic_error** (both defined in header `<stdexcept>`), each of which has several derived classes.
- Also derived from `exception` are the exceptions thrown by C++ operators—for example, **bad_alloc** is thrown by `new`, **bad_cast** is thrown by `dynamic_cast` and **bad_typeid** is thrown by `typeid`.
- Including **bad_exception** in the `throw` list of a function means that, if an unexpected exception occurs, function `unexpected` can throw `bad_exception` rather than terminating the program's execution (by default) or calling another function specified by `set_unexpected`.

Standard Library Exception Hierarchy (cont.)

- Class `logic_error` is the base class of several standard exception classes that indicate errors in program logic.
  - For example, class **invalid_argument** indicates that an invalid argument was passed to a function.
  - Proper coding can, of course, prevent invalid arguments from reaching a function.
- Class **length_error** indicates that a length larger than the maximum size allowed for the object being manipulated was used for that object.
- Class **out_of_range** indicates that a value, such as a subscript into an array, exceeded its allowed range of values.

Standard Library Exception Hierarchy (cont.)

- Class **`runtime_error`**, is the base class of several other standard exception classes that indicate execution-time errors.
  - For example, class **`overflow_error`** describes an arithmetic overflow error (i.e., the result of an arithmetic operation is larger than the largest number that can be stored in the computer) and class **`underflow_error`** describes an arithmetic underflow error (i.e., the result of an arithmetic operation is smaller than the smallest number that can be stored in the computer).