

EXCEPTION HANDLING (Software exceptions)

- ❖ A software exception is raised (initiated) by a piece of code when it encounters an abnormal or exceptional condition.
- ❖ Exception handling provides a disciplined way of dealing with erroneous runtime events.
- ❖ It is a mechanism that allows a program to detect and possibly recover from errors during execution. The alternative is to terminate the program.
- ❖ Exception handling makes functions easier to understand because the normal control flow is separated from error handling logic.
- ❖ Once an exception condition is raised, it must be handled. It will not go away by ignoring it. It will go away only when it is recognized and "handled".

Common Exceptions

- Out-of-Bound array subscript
- Arithmetic overflow
- Divide by Zero
- Out of memory
etc

C++ Exception Handling model

C++ introduces three keywords - **try**, **throw** & **catch**, to handle exceptions

1. You **try** an operation anticipating an error.
2. When the function encounters an error, it **throws** an exception.
3. You anticipate the error and **catch** the exception for cleanup/recovery.

Syntax

```
try {  ←// try block
    ....

    throw exception() ; // What is thrown is an object
    ....
}
catch (exception) {    // ← catch block
    ....
}
```

How does C++ exception handling work?

```
void func()
{
    // some code

    throw 1;

    // more code
}
```

```
int main()
{
    // some code

    try {
        // some more code

        func(); //call func().Be prepared for any errors

        // more code
    }
    catch (...) // catches all exceptions
    {
        // catch exception thrown by func()

        // cleanup & do other things
    }

    // more code in main continued
}
```

- What happens if there is no "catch" block in main()?

The exception is not handled & a pre-defined function `terminate()` is called after control returns to main from `func()`.

The call sequence is

1. No exception case

calls
main() → func() → skip catch block → return to main()

2. Exception raised in func()

calls
main() → func() → jump to catch block → catch → rest of code in main

- ❖ When you leave a function due to an exception, cleanup happens.
- ❖ destructor is called for each object created in func()
- ❖ temporary objects and args passed are destroyed
- ❖ the stack is removed & control return to the caller

This happens for all functions in the call chain.

Example 1

```
//Prototypes
class DivideByZero {};
float divide(int, int);

// Main function
int main(void)
{
    int a, b;

    cin >> a >> b;

    try {

        float c = divide(a, b);

        cout << "a/b = " << c << endl;
    }

    catch (DivideByZero)
    { //Catches only the DivideByZero exception

        cout << "Attempt to divide by zero.\n";

    }
}

float divide(int x, int y) {

    if ( y == 0) throw DivideByZero();

    return (float)x/y;
}
```

Example 2

```
class DivideByZero { // Exception class

    const char* _errmsg;

public:
    DivideByZero(): errmsg("Error: dividing by zero.")
    {}
    void print() { cout << _errmsg; }
};

int main(void)
{
    int a, b;

    cin >> a >> b;

    try {

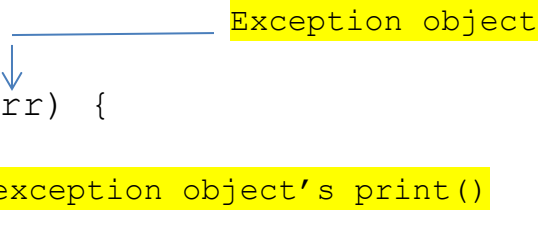
        float c = divide(a, b);

        cout << "a/b = " << c << endl;
    }
    catch (DivideByZero err) {
        err.print(); //Call exception object's print()
    }
}

float divide(int x, int y) {

    if ( y == 0) throw DivideByZero();

    return (float)x/y;
}
```



The try block

- hint to the compiler to look for catch blocks in case of an exception
- try blocks can be nested
- when an exception is thrown, the search for a catch block starts with the try block that was most recently entered.

The throw expression

- throws an object
- The expression can throw any type of object

```
throw ("Here's an exception") <-> catch (const char*)  
throw (3.15146) <-> catch(double)
```

- A throw without an operand **re-throws** the exception being handled

The catch block

- catch (operand) catches the exception matching the operand
- catch (...) catches all exceptions

Another example - handling multiple exceptions

```
#include <iostream>
#include <vector.h>

class intStack {
public:
    intStack(int size) : _stack(size), _top(0) { }
    void pop(int);
    void push(int);
    bool empty() {
        return _top ? false : true;
    }
    bool full() {
        return _top < size()-1 ? false : true;
    }
    int size() { return _top;}

private:
    int _top;
    vector<int> _stack;
};

// Exception classes

class PopOnEmpty { };
class PushOnFull { };

// Implementation of intStack methods

void intStack::pop(int& top_value) {
    if (empty())
        throw PopOnEmpty();
    top_value = _stack[--_top];
}

void intStack::push(int value) {
    if (full())
        throw PushOnFull();
    _stack[_top++] = value;
}
```



```

//    Main program

int main(void)
{
    intStack stack(32);

    try
    {
        for (int i = 1; i < 51; i++)
        {
            if (i % 3 == 0)
                stack.push(i);

            if (i % 10 == 0) {
                int dummy;
                stack.pop(dummy);
                cout << dummy;
            }
        }
    }
    catch (PushOnFull)
    {
        cerr << "Trying to push a value on a full stack";
        return err99;
    }
    catch (PopOnEmpty)
    {
        cerr << "Trying to pop a value from a empty stack";
        return err100;
    }

    return 0;
}

```