



BIT

Exception Handling

Hu Sikang
skhu@163.com

BIT

Contents

- **Exception and Exception Handling**
- **Structure of exception handling in C++**

BIT

Choices upon an error

- Ignore the error
- Terminate immediately
- Set an error flag, check by the caller
- Exception handling

BIT

Common Exceptions

- **out-of-bound array subscript**

```
int a[100], index;  
cin >> index;  
for (int i = 0; i <= index; i++)  
    cin >> a[i];
```

- **dividing by zero**

```
double div(double x, double y)  
{    return x / y; }
```

- **arithmetic overflow**

```
int i = 1;  
while (i > 0)    i++;
```

BIT

Without Exception Handling

```
#include <iostream>
using namespace std;
double Div(double a, double b)
{
    if (b == 0)
    {
        cout << "Untenable
                arguments to Div()";
        exit(0);
    }
    return a / b;
}
```

```
int main()
{
    double x, y, z;
    cout << "Enter two numbers: ";
    cin >> x >> y;
    z = Div(x, y);
    cout << x << " / " << y <<
        " = " << z << endl;

    return 0;
}
```



BIT

What is Exception Handling?

- It is a *mechanism* that allows a calling program to detect and possibly *recover from errors* during execution.

BIT

With Exception Handling

```
#include <iostream>

using namespace std;

double Div(double a, double b)
{
    if (b == 0)
        throw "Divided by zero";
    return a / b;
}
```

```
int main( ) {
    double x, y, z;
    cout << "Enter two numbers: ";
    cin >> x >> y;
    try {
        z = Div(x, y);
        // The statement should NOT be
        // written after catch.
        cout << x << " / " << y << " = " << z;
    }
    catch (const char* info)
    { cout << info << endl; }
    return 0;
}
```

BIT

Structure of Exception Handling

```
try {  
    statement-list  
}  
catch (exception1) {  
    statement-list1  
}  
catch (exception2) {  
    statement-list2  
}  
catch (...) {  
    statement-list...  
}
```



From top to bottom

BIT

Passing Data with Exceptions

```
class CArray {  
private: int* v, lower, upper;  
public:  
    CArray(int l, int u) : lower(l), upper(u)  
    {  
        if (lower < 0) throw CError(0);  
        if (upper < 0) throw CError(1);  
        v = new int[upper - lower + 1];  
    }  
    int& operator[ ](int);  
    ~CArray() { if (v) delete[] v; }  
};
```

```
int& CArray::operator[ ](int i) {  
    if (i >= lower && i < upper)  
        return *(v + i - lower);  
  
    throw CError(2);  
};
```

BIT

Passing Data with Exceptions

```
class CError {  
private: int index;  
public:  
    CError(int i) { index = i; }  
    string Info()  
    {  
        switch (index)  
        {  
        case 0:  
            return "Lower Error";    break;  
        case 1:  
            return "Upper Error";    break;  
        case 2:  
            return "Subscriptor Error"; break;  
        }  
    }  
};
```

BIT

Passing Data with Exceptions

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

int main()
{
    try
    {
        CArray arr(0, 10); // CArray arr(-1, 10)
        arr[9] = 90;       // arr[10] = 100
    }
    catch (CError error)
    {
        cout << error.Info() << endl;
    }

    return 0;
}
```

BIT

Multiple Handlers

- Most programs performing exception handling have to handle more than one type of exception. A single *try* block can be followed by multiple handlers(*catch*), each configured to match a different exception type.

BIT

Multiple Handlers

```
int main()
{
    try {
        CArray arr(0, 10); // Carray arr(-1, 10)
        arr[9] = 90;       // arr[10] = 100
        double result = Div(arr[9], 0);
    }
    catch (CError error) { cout << error.Info() << endl; }
    catch (string info)  { cout << info << endl; }
    return 0;
}
```

```
double Div(double a, double b) {
    if (b == 0)
    { throw "Untenable arguments to Div() "; }
    return a / b;
}
```

Exception with no Catch

- If no catch matches the exception generated by the try block, the search continues with the next enclosing try block.
- If no catch found, error!

```
void Callfun(const array& a)
{
    try

        fun (a);
    }
    // error if no catch!
}
```



BIT

Using Inheritance

```
#include <exception>
```

```
class exception;
```

```
class CMyException : public exception;
```

```
try {
```

```
    ...
```

```
}
```

```
catch (CMyException& my) {
```

```
    ...
```

```
}
```

```
catch (Exception) {
```

```
    ...
```

```
}
```



BIT

Using Inheritance

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

class CMyException : public exception
{
public:
    virtual const char* what() const throw( )
    {
        return "CMyException";
    }
};
```


BIT

Using Inheritance

```
int main( )  
{  
    try  
    {  
        throw CMyException();  
    }  
    catch (const CMyException& my) { cout << my.what() << endl; }  
    catch (const exception& e)      { cout << e.what() << endl; }  
    return 0;  
}
```