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];
for (int i = 0; i <= 100; 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() " << endl;
        exit(0);
    }
    return a / b;
}
```

BIT

Without Exception Handling

```
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 "Untenable arguments to Div() ";
    }
    return a / b;
}
```

With Exception Handling

```
int main(void) {
        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 << endl;
        }
        catch (const char* info)
        {   cout << info << endl;   }
        return 0;
}
```

BIT

Structure of Exception Handling

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

Passing Data with Exceptions

```
class CArray {
private:  int* v, lower, upper;
public:
        CArray(int l, int u) : lower(l), upper(u)
        {
                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(i);
};
```

```
#include <iostream>
using namespace std;
class CError {
private: int index;
public:
        CError(int i)  { index = i; }
        int Get() { return  index; }
};
```

BIT

Passing Data with Exceptions

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

class CError {
private:  int index;
public:
        CError(int i) { index = i; }
        int Get() { return  index; }
        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

```
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

```
int main()
{
        try
        {
                CArray arr(0, 10);    // Carray arr(-1, 10)
                arr[9] = 100;         // 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.

Multiple Handlers

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

BIT

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 fun(array& a)
{
    try

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

BIT

Using Inheritance

```
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 MyException();
        }
        catch (CMyException& my)
        {
                cout << my.what() << endl;
        }
        catch (exception& e)
        {
                cout << e.what() << endl;
        }
        return 0;
}
```