

OOP

Lecture 25

20/01/2021

Error Handling :

As such, C programming does not provide direct support for error handling but being a system programming language, it provides you access at a lower level in the form of return values. Most of the C function calls return -1 or NULL in case of any error. So a C programmer can check the returned values and can take appropriate action depending on the return value.

Error testing is usually a straightforward process involving if statements or other control mechanisms like if-else if structures. For example, the following code segment will trap a division-by-zero error before it occurs:

```
if (denominator == 0)
    cout << "ERROR: Cannot divide by zero.\n";
else
    quotient = numerator / denominator;
```

But what if similar code is part of a function that returns the quotient, as in the following example?

```
float divide(int numerator, int denominator)
{
    if (denominator == 0)
    {
        cout << "ERROR: Cannot divide by zero.\n";
        return 0;
    }
}
```

```
        else
            return (double)numerator / denominator;
    }
```

Functions commonly signal/show error conditions by returning a predetermined value (The value which is returned when an error occurred like -1 or NULL). Apparently, the function in this example returns 0 when division by zero has been attempted. This is unreliable, because 0 is a valid result of a division operation. Even though the function displays an error message, the part of the program that calls the function will not know when an error has occurred. Problems like these require sophisticated error handling techniques. Like when we call this function from the main function we will never be able to check whether the division operation is successful or not, as we cannot set any unique value which will be returned in case of division by 0 error. It is because every value could be a valid result of the division.

Exceptions :

Exceptions are used to point out errors or unexpected events that occur while a program is running. One of the advantages of C++ over C is Exception Handling. Exceptions are run-time errors or abnormal conditions that a program encounters during its execution.

C++ provides the following specialized **keywords** for this purpose.

try : represents a block of code that contains success scenario code or it can throw an exception in case of error.

catch : represents a block of code that is executed when a particular exception is thrown. It contains alternate scenario code.

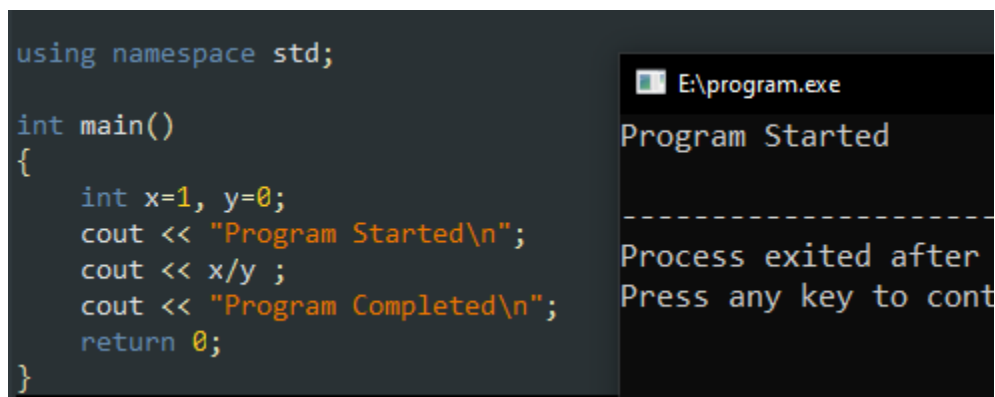
throw : Used to throw an exception.

The try and catch keywords come in pairs.

The syntax of try-catch is:

```
try{  
    -----  
    //Success Scenario code  
}  
  
catch( parameter ){  
    -----  
    //Alternate Scenario code  
}
```

Now let's go to some coding part : Check out this program,



The image shows a code editor on the left and a console window on the right. The code editor contains the following C++ code:

```
using namespace std;  
  
int main()  
{  
    int x=1, y=0;  
    cout << "Program Started\n";  
    cout << x/y ;  
    cout << "Program Completed\n";  
    return 0;  
}
```

The console window on the right shows the output of the program:

```
E:\program.exe  
Program Started  
-----  
Process exited after  
Press any key to cont
```

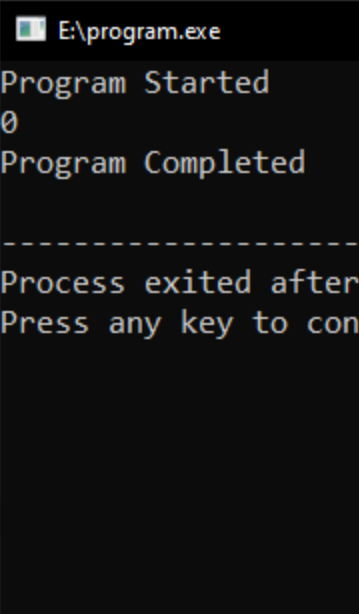
As we can see here the program crashed when we attempted to divide by 0.

Now, we need to save our program from crashing. For this we can apply an if-else check to verify that the denominator is not 0. Consider we are using a divide function and the function returns the quotient of the division operation. Check out this code,

```
float divide(int numerator, int denominator)
{
    if (denominator == 0)
        return 0;

    else
        return (double)numerator / denominator;
}

int main()
{
    int x=1, y=0;
    float result;
    cout << "Program Started\n";
    result = divide (x, y);
    cout << result;
    cout << "\nProgram Completed\n";
    return 0;
}
```



Here, the program is working completely fine but there is a logical error in it, that is whenever the denominator is 0 it will return 0. But, 0 could also be a valid result of the division. So, we can never know if the result variable is holding a valid value of our division or it just returned 0 just because the division is not valid. These kinds of problems are best handled by throwing an exception.

Whenever we encounter a problem where we know that some cases of input in this function can cause the program to crash, we throw an exception from that function and catch that exception in our main() function.

If an exception is thrown from a function and there is no catch block from where the function was called, then our program will immediately be crashed saying an exception is thrown and not handled.

For further understanding check out this program,

```
float divide(int x, int y)
{
    if (y==0)
        throw ("Denominator is zero\n");

    return x/y;
}
int main(){
    int x=1, y=0;
    cout << "Program Started\n";
    cout << divide (x, y) << '\n';
    cout << "Program Completed\n";
    return 0;
}
```

```
E:\program.exe
Program Started
terminate called after throwing an instance of 'char const*'
-----
Process exited after 0.661 seconds with return value 3
Press any key to continue . . .
```

Here, we can see that we throw an exception from the function but there is no try catch block in the main function, so our program crashes. We need proper try catch blocks before throwing an exception.

```
int divide(int x, int y){
    if (y==0) throw ("Denominator is zero.");
    return x/y;
}
int main(){
    int x=1, y=0;
    cout << "Program Started\n";

    try{
        cout << divide (x, y) << '\n';
    }catch(const char* s){
        cout << s << " Division is not possible\n";
    }
    cout << "Program Completed\n";

    return 0;
}
```

```
E:\program.exe
Program Started
Denominator is zero. Division is not possible
Program Completed
-----
Process exited after 0.105 seconds with return value 0
Press any key to continue . . .
```

Now, this program is working completely fine. An exception is thrown and successfully handled in the catch block. Here some important points to understand are :

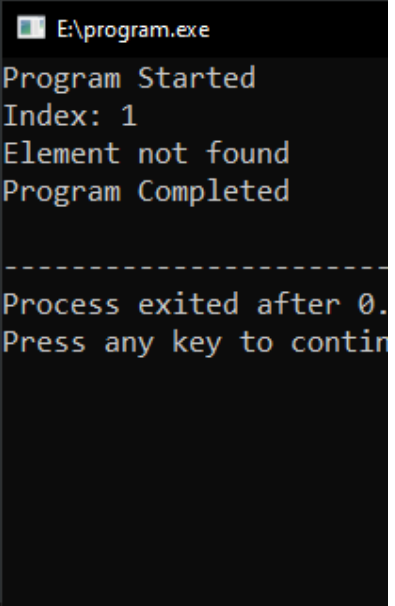
- We can throw any type of exception i.e int, float, char*, or even an object of any class.
- We can see that the catch block is parameterized. We need to provide the relevant parameter here of which the exception is thrown.

- There could be multiple catch blocks with different types of parameters and the relevant catch block will execute of which type the exception is thrown.

Consider one more example of a function which finds the index of an element in an array and throws an integer exception in case of the index not found.

```
int indexOf(int x[], int ELEMENT, int SIZE){
    for (int i=0;i<SIZE;i++)
        if (x[i]==ELEMENT)
            return i;
    throw (0);
}

int main(){
    int x[]={13,48,26,27,95};
    cout << "Program Started\n";
    try{
        cout << "Index: " << indexOf(x, 48,5) << '\n';
        cout << "Index: " << indexOf(x, 84,5) << '\n';
    }catch(int){
        cout << "Element not found\n";
    }
    cout << "Program Completed\n";
    return 0;
}
```



This program is just to show that we can throw and catch any type of exception.

As we have described earlier we could have multiple catch blocks with different types of parameters. There exists a default catch block as well which is executed if no matching catch block is found. We can write a default catch block like this **catch(...){ }**. It can catch any type of exception and do the relevant job which we want to do as default.

In the following program we wrote 2 catch blocks 1 is for string and other is the default block. We throw the integer exception from the function and it is caught by the default catch block.

```

int indexOf(int x[], int ELEMENT, int SIZE){
    for (int i=0;i<SIZE;i++)
        if (x[i]==ELEMENT)
            return i;
    throw (0);
}

int main(){
    int x[]={13,48,26,27,95};
    cout << "Program Started\n";
    try{
        cout << indexOf(x, 48,5) << '\n';
        cout << indexOf(x, 84,5) << '\n';
    }catch(const char*s){
        cout << s;
    }catch(...){
        cout << "Default\n";
    }
    cout << "Program Completed\n";
    return 0;
}

```

```

E:\program.exe
Program Started
1
Default
Program Completed

-----
Process exited after 0
Press any key to conti

```

In C++, many built in functions throw exceptions in case of any error occurred. A good programmer always reads the complete documentation of the built in function before using it. If it throws an exception in case of any error occurring, then it should be written inside a **try** block and the relevant **catch** block should also be written to catch the exception of that specific type which could be thrown by the function.

Object Oriented Approach for Exceptions :

Generally when we write **throw (0)** what does 0 mean? In object oriented programming, we define classes to reuse the code and to make our code more readable. To do this, C++ provides us exception classes which are declared to make our code more readable and presentable. By using exception class we can throw an object of the class, which will then be caught by any catch blocking containing that class type parameter.

```
#include <iostream>
#include "balance.cpp"

using namespace std;

void withdraw(int amount, int &balance)
{
    if( balance >= amount )
        balance = balance - amount;
    else
    {
        InsufficientBalance objB(balance);
        throw (objB);
    }
    cout << "Withdraw Succcessful\n";
}

int main(){
    int balance = 1000;
    try{
        withdraw( 500, balance);
        withdraw( 1000, balance);
    }
    catch(InsufficientBalance &b){
        b.show();
    }
    return 0;
}

balance.cpp - Notepad
File Edit Format View Help
#ifndef BALANCE
#define BALANCE

#include<iostream>

using namespace std;

//Exception class
class InsufficientBalance{
    int balance;

public:
    InsufficientBalance(int b)
    {
        balance = b;
    }
    void show() const
    {
        cout << "Balance is " << balance << '\n';
    }
};

E:\program.exe
Withdraw Succcessful
Balance is 500

-----
Process exited after 0.49
Press any key to continue
```

In this way we can make our code more readable, so that we can just see the main() function and tell which kind of error occurred here. Like, in this case we know that the Insufficient Balance error could occur here so we wrote the catch block for it beforehand knowing that it could throw an object of the InsufficientBalance class.

Consider one more example, when we allocate dynamic memory by using **new operator**, it allocates the memory if sufficient memory is available/free, but what would happen if we provide a very very big number to allocate memory, in such case, the **new** operator throws an object of **bad_alloc** class which should be caught in a catch block otherwise our program will crash immediately. Check out this program.


```

int main(){
    int *x;
    try{
        while (1){
            x = new int[100000000];
            cout << "Memory Allocated\n";
        }
    }catch (bad_alloc &b){
        cout << "Limit Reached\n";
    }
    cout << "Program Completed\n";
    return 0;
}

```

```

Memory Allocated
Memory Allocated
Memory Allocated
Memory Allocated
Memory Allocated
Memory Allocated
Memory Allocated
Memory Allocated
Memory Allocated
Memory Allocated
Memory Allocated
Memory Allocated
Memory Allocated
Memory Allocated
Memory Allocated
Memory Allocated
Memory Allocated
Memory Allocated
Memory Allocated
Memory Allocated
Memory Allocated
Limit Reached
Program Completed

```

-THE END-