Chapter 9: Exception Handling

BIBHA STHAPIT
ASST. PROFESSOR
IOE, PULCHOWK CAMPUS

Introduction

- Two common types of error in a program are:
  - 1) **Syntax error** (arises due to missing semicolon, comma, and wrong program constructs etc)
  - 2) **Logical error** (wrong understanding of the problem or wrong procedure to get the solution)

- **Exceptions** are the errors occurred during a program execution. Exceptions are of two types:
  - Synchronous (generated by software i.e. division by 0, array bound etc).
  - Asynchronous (generated by hardware i.e. out of memory, keyboard etc).

Introduction

- The purpose of exception handling mechanism is to detect and report an exceptional circumstances so that appropriate action can be taken. The mechanism for exception handling is:
    1. Find the problem(hit the exception).
    2. Inform that an error has occurred(throw the exception).
    3. Receive the error information(Catch the exception).
    4. Take corrective actions(Handle the exception).

Exception handling mechanism

- C++ exception handling mechanism is basically built upon three keywords namely, try, throw and catch.

- Try block hold a block of statements which may generate an exception.
  - The try block is the one that can throw an exception. The code that is to be monitored for exceptions is placed within this block. Thus, the try block is the scope of exception generation. Whenever a specific code segment is expected to throw an exception, such segment is placed within the try block. Thus, this block contains either a throw statement or a function containing either a throw statement or a similar function inside the body.

Exception handling mechanism

- When an exception is detected, it is thrown using a throw statement in the try block.
  - This is a mechanism to generate the exception. It is usually a single statement starting with the keyword throw or a function call that contains throw inside its body. After the execution of this statement, the control is transferred to the corresponding catch block written immediately after the try block, if the exception is thrown.
  - The identifier following the throw is the name of the variable being thrown. The control now is permanently transferred to the catch block and the statements after the throw statement are not executed.

Exception handling mechanism

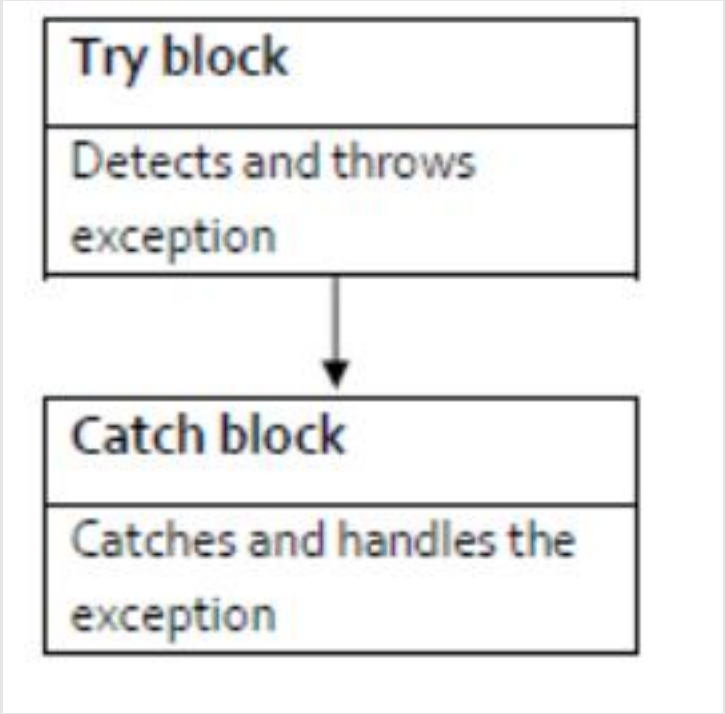
- A try block can be followed by any number of catch blocks.
  - This is the section where the exception is handled. There are two ways to throw an exception: first, by using an explicit throw statement or second, by calling a function which in turn contains a throw statement or a similar function call in its body. The exception is handled by the catch block, which should immediately follow the try block.

Exception handling mechanism

- The general form of **try** and **catch** block is as follows:

```
try
{
        /* try block;
        throw exception*/
}

catch (type1 arg)
{
        /* catch block*/
}
```



1. W/O function and exception class

```
main()
{       int a,b;
        cout<<“Enter two numbers (num/den):";
        cin>>a>>b;
        try
                {       if (b==0)
                                throw b;
                        else
                                cout<<“Result=“<<a/b;
                }
        catch(int x)
        {       cout<<"Denominator cannot be zero";   }
}
```

Output:

1st Run:

Enter two numbers (num/den):8 2

Result=4

2nd Run:

Enter two numbers (num/den):4 0

Denominator cannot be zero

2. Exception generated by function

```
void divide (int a, int b)
{       if (b==0)
                        throw b;
                else
        cout<<"Result="<<a/b<<endl;
}
main()
{       try
        {       divide(6,3);
                divide(5,0);
        }
        catch(int x)
        {       cout<<"Denominator cannot be zero";   }
}
```

Output:

Result=2

Denominator cannot be zero

3. Exception class

```
class numbers
{    int a, b;
public:
    class div_zero{ };// Exception class
    void divide()
    {
        cout<<"enter two numbers (num/den):";
        cin>>a>>b;
            if (b==0)
                throw div_zero();
            else
                cout<<"Result="<<a/b<<endl;
    }
};
```

```
main()
{
    numbers n;
    try{
    n.divide();
    n.divide();
    }
    catch(numbers::div_zero)
       {     cout<<"Denominator cannot be zero";   }
}
```

```
Output:
enter two numbers (num/den):9 3
Result=3
enter two numbers (num/den):7 0
Denominator cannot be zero
```

Multiple exception handling

- In some situations the program segment has more than one condition to throw an exception. In such case more than one catch blocks can be associated with a try block as shown:

```
try   {     //try block
            }
catch(type1 arg)
{                   //catch block1
}
catch(type 2 arg)
{                   //catch block 2
}
……..
catch (type N arg)
{                   //catch block N
}
```

```
main()
{
   int n;
   cout<<"Enter any integer value:";
   cin>>n;
   try
   {
      if(n==0)
         throw 1;
      if (n>0)
         throw '1';
      if (n<0)
         throw 1.0;
   }
```

```
catch(int i)
   {
      cout<<"Integer exception caught";
   }
   catch(char ch)
   {
      cout<<"char exception caught";
   }
catch(double d)
   {
      cout<<"Double exception caught";
   }
}
```

Output:

1st run:

Enter any integer value: -9

Double exception caught

2nd run:

Enter any integer value: 0

Integer exception caught

3rd run:

Enter any integer value: 12

Character exception caught

Catch all exception

- In some cases when all possible type of exceptions cannot be anticipated and may not be able to design independent catch handlers to catch them
- In such situations a single catch statement is forced to catch all exceptions instead of certain type alone.
- This can be achieved by defining the catch statement using ellipses as follows

```
catch(. . .)
{
//statement for processing all exceptions
}
```

```
main()
{    int n;
    cout<<"Enter any integer value:";
    cin>>n;
    try
    {
        if(n==0)
            throw 1;
        if (n>0)
            throw '1';
        if (n<0)
            throw 1.0;
    }
catch(. . .)
    {       cout<<“An exception caught";
    }
}
```

Output:

1st run:

Enter any integer value: -9

An exception caught

2nd run:

Enter any integer value: 0

An exception caught

3rd run:

Enter any integer value: 12

An exception caught

Re-throwing exception

- A handler may decide to re-throw an exception caught without processing them. In such situations we can simply invoke throw without any argument like

```
throw;
```

- This cause the current exception to be thrown to the next enclosing try/catch sequence and is caught by a catch statement listed after that enclosing try block.

```
void divide(int x, int y)
{
cout<<" Inside Function \n";
try
{ if (y== 0)
    throw y; //throwing int
else
    cout<<"Result =" << x/y<<endl;
}
catch(int) //Catch a int
{
cout<<"Caught int inside a function \n";
throw; //re-throwing int
}
cout<<"End of function\n\n";
}
```

```
int main()
{
cout <<"Inside main \n";
try
{ divide(10,2);
divide(20,0);
}
catch (int)
{
cout <<"Caught int inside main \n";
}
cout <<"End of main\n ";
return 0;
}
```

Output:

```
Inside main
 Inside Function
Result =5
End of function

 Inside Function
Caught int inside a function
Caught int inside main
End of main
```

Exception specification for function

- In some cases it may be possible to restrict a function to throw only certain specified exceptions. This is achieved by adding a throw list clause to function definition. The general form of using an exception specification is:

```
Ret_Type function (arg-list) throw (type-list)
{
// function body
}
```

- The type list specifies the type of exceptions that may be thrown. Throwing any other type of exception will cause abnormal program termination. To prevent a function from throwing any exception, it can be done by making the type list empty like throw(); in the function header line.
- Similarly, include throw(…); in function header line to throw all types of exceptions.

```
void test(int n)throw(double,int)
{
      cout<<"Inside function\n";
       if(n==0)
          throw 1;
      if (n>0)
          throw '1';
      if (n<0)
          throw 1.0;
           cout<<"End of function\n";
}
```

```
main()
{   cout<<"Inside main\n";
    try
    {  cout<<"testing throw restrictions\n";
    test(23);
    test(0);
    }
 catch(int i)
 {       cout<<"Integer exception caught\n";    }
 catch(char ch)
 {       cout<<"character exception caught\n";    }
catch(double f)
 {       cout<<"Double exception caught\n";    }
  cout<<"End of main\n";
}
```

Output:

```
Inside main
testing throw restrictions
Inside function
terminate called after
throwing an instance of
'char'
```

Exception with argument

```
class numbers
{
    int a, b;
public:
    class div_zero// Exception class
    {
    public:
        int err_no;
        char err_name[50];
        div_zero(int num, char *name)
        {    err_no=num;
            strcpy(err_name,name);
        }
    };
    void divide()
        {
            cout<<"enter two numbers (num/den):";
            cin>>a>>b;
                if (b==0)
                    throw div_zero(b,"Divide by zero");
                else
                    cout<<"Result="<<a/b<<endl;

        }
};
```

Exception with argument(contd.)

```
main()
{
   numbers n;
   try{
   n.divide();
   n.divide();
   }
   catch(numbers::div_zero z)
     {    cout<<"Exception Caught:"<<z.err_no<<z.err_name;

      }
}
```

Output:

```
enter two numbers (num/den):65 5
Result=13
enter two numbers (num/den):65 0
Exception Caught:0Divide by zero
```

Handling uncaught exception

- If no handler at any level catches the exception, the special library function **terminate( )** (declared in the **<exception>** header) is automatically called. By default, **terminate( )** calls the library function **abort( )** , which abruptly exits the program.

- This **terminate()** function is modifiable as per user requirements. To change the terminate handler, we use **set_terminate()** which is defined under **<exception>** class as:

  set_terminate(defined_terminate_handler)

- The terminate handler set by **set_terminate()** should be a function that take no argument and do not have return type. It must stop program execution and it must not return to program or resume it in any way.

```
void test(int n)
{
        cout<<"Inside function\n";
         if(n==0)      throw 1;
        if (n>0)       throw '1';
        if (n<0)        throw 1.0;
            cout<<"End of function\n";
}

void my_uncaught_handler()
{
    cout<<"Uncaught exception found\n";
      abort();
}
```

```
main()
  {
     set_terminate(my_uncaught_handler);
      cout<<"Inside main\n";
     try
     {
      test(-3);
     }
   catch(int i)
   {       cout<<"Integer exception caught\n";
   }
      cout<<"End of main\n";
   }
```

Output:

Inside main

Inside function

Uncaught exception found

Handling unexpected exception

- The special function **unexpected( )** is called when we throw something other than what appears in the exception specification. The default **unexpected( )** calls the **terminate( )** function.

- But this behavior can be redefined by calling **set_unexpected**. This function is automatically called when a function throws an exception that is not listed in its *dynamic-exception-specifier* (i.e., in its throw specifier).

- This function is provided so that the *unexpected handler* can be explicitly called by a program, and works even if **set_unexpected** has not been used to set a custom *unexpected handler* (calling **terminate** in this case)

```
void test(int n)throw(int)
{
   cout<<"Inside function\n";
      if(n<0) throw 1;
      if (n>=0)  throw '1';
      cout<<"End of function\n";
}

void my_unexpected_handler()
{
   cout<<"Unexpected exception raised";
}
```

```
main()
  {    set_unexpected(my_unexpected_handler);
       cout<<"Inside main\n";
      try
      {
       test(23);
       }
   catch(int i)
   {      cout<<"Integer exception caught\n";    }
catch(char c)
   {      cout<<“Character exception caught\n";    }
      cout<<"End of main\n";
   }
```

Output:

Inside main

Inside function

Unexpected exception raised

STACK USING EXCEPTION:

```
class my_stack
{
    int *a;
    int SIZE;
    int top;
public:
    class EMPTY{ };
    class FULL{ };
    my_stack(int n)
    {
        top=-1;
        SIZE=n;
        a=new int[n];
    }
    void push(int x)
    {    if(top==SIZE-1)
            throw FULL();
        a[++top]=x;
    }
    int pop()
    {
        if(top==-1)
            throw EMPTY();
        return a[top--];
    }
    void display()
    {
        for(int i=0;i<=top;i++)
            cout<<a[i]<<ends;
    }    };
```

```
main()
{
    int item,sz;
    cout<<"\nEnter the size of stack:";
    cin>>sz;
    my_stack s(sz);

        int ch=1;
        cout<<"\nStack with Exception Handling";
        cout<<"\n\n\tMENU\n1.PUSH\n2.POP\n
                    3.SHOW STACK\n4.EXIT";
        cout<<"\nEnter your choice:";
        cin>>ch;
```

```
do
        {
        switch(ch)
        {
        case 1:
        cout<<"\nEnter the item to push:";
        cin>>item;
        try
        {
        s.push(item);
        }
        catch(my_stack::FULL)  //FULL object is caught
        {
        cout<<"\n***Stack Overflow***\n";
        }
        break;
```

```
case 2:
        try
        {
        cout<<"\nPoped Item is:"<<s.pop();
        }
        catch(my_stack::EMPTY) //EMPTY object caught
        {
        cout<<"\n***Stack Empty***\n";
        }
        break;
```

```
case 3:
        cout<<"\nThe Stack is:\n";
        try
        {       s.display();    }
        catch(my_stack::EMPTY)
        {
        cout<<"\n***Stack Empty***\n";
        }
        break;
        case 4:
              exit(0);
        }
        cout<<"\nEnter your choice:";
        cin>>ch;
        }while(ch<5);
}
```