

C++ Exceptions

CO650 Advanced Programming

Topics

- Exception Handling
- Assertions

Exceptions

- Means of handling runtime anomalies
- Examples being invalid input or running out of memory.
- Separates the code that detects the error from that that handles it.
- The mechanism provides the ability to
 1. Define exceptions
 2. Instantiate exceptions
 3. Create blocks to detect and handle exceptions

Exception Handling

- The code that may generate exceptions (under exception inspection) is placed within a **try** block.
- Exceptions thrown within this block cause the control to pass to the exception handler.
- If no exceptions are thrown the handler is ignored
- Handlers (catch clauses) are declared with the **catch** keyword.
- The handler must be defined immediately after the try block.
- Exceptions can be raised (created) explicitly within the code using the **throw** keyword.

Exception Handling

- The three dots in the brackets (...) after the catch keyword indicate the handler will catch any type of exception.

```
try{  
    // Code that may generate an exception  
}  
catch(...){  
    // Error handling code  
}
```

- If an exception is raised in the try block and no suitable catch cause has been defined execution will pass to the terminate function declared within std.

Function Try Block

- The entire body of a function can be placed inside a try block.
- Instead of embedding the try block within the function's statements, the try block encloses the function's statements.

```
type Identifier()  
try{  
  
}  
catch(){  
  
}
```

Throwing Exception

- An exception can be created using the throw keyword.
- Throw accepts one parameter
- The parameter is passed as an argument to the handler
- For literal values

```
throw(literal);
```

- For user defined types. Note the constructor is invoked.

```
throw ClassName();
```

- Alternatively create an instance of the user defined type

```
ClassName obj;  
throw obj;
```

Exception Handling

- Handlers (catch()) can only contain one type
- When an exception is thrown the handler with a matching type is executed.
- Exception handlers can be chained.
- Each one having a different argument type.
- Rather like overloading , the handler with the matching argument type will be invoked.

```
try{  
    throw parameter;  
}  
catch(type argument){  
    // Error handling code  
}  
catch(...){  
    // Catch any that got through  
}
```


Example Exception Handler

```
char * buffer;
int size;
try{
    cout << "Enter array size ";
    cin >> size;
    if (size < 1)
        throw 1;
    buffer = new char[size];
    if (buffer == 0)
        throw "Out of Memory";
}
catch(char * error){
    cout << "Exception : " << error << endl;
}
catch(int errorNo){
    cout << "Exception : " << errorNo << endl;
}
catch(...){
    cout << "Exception : " << endl;
}
```

Exception Specification

- The exceptions thrown within a function can be limited by specifying them within the function declaration.

```
void functionName() throw (type, type) ;
```

- If the function declaration includes throw () with no type it means the function can not throw an exception. However if it does the program will compile but may not execute correctly.

```
void functionName() throw () ;
```

- Omitting the throw keyword permits the function to throw any kind of exception.

```
void functionName() ;
```

- As does including the three dots.

```
void functionName() throw(...) ;
```

- The compiler will not enforce this checking at compile time

Exception Class

- While any user type can be used as an exception object, the C++ Standard Library includes the base Exception class.
- Located within the <exception> header in the std namespace
- Contains a single virtual member function **what** which returns a const char *

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

- Exceptions can be derived from this class and override the **what** member function, in order to return an appropriate message.
- const after the what() states the function can't change the state of the object.

Exceptions Example

Catching the exception as a reference (&) avoids the overhead of generating a copy.

Through the exception object (e) the **what** member function can be accessed.

In this example the handler displays the message “Controller Error”

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

class IOException: public exception {
public:
    virtual const char* what() const throw() {
        return "Controller Error";
    }
};

int main(){
    IOException eio;
    try {
        throw eio;
    }
    catch (IOException& e) {
        cout << e.what() << endl;
    }
}
```

Example Exception Handler

```
class MemoryException: public exception {
public:
    virtual const char* what() const throw() {
        return "Out of memory";
    }
};

int main(){
    char * buffer; int size;
    try{
        cout << "Enter array size "; cin >> size;
        buffer = new char[size];
        if (buffer == 0)
            throw MemoryException();
    }
    catch(MemoryException e){
        cout << "Exception : " << e.what() << endl;
    }
}
```

Example Exception Handler

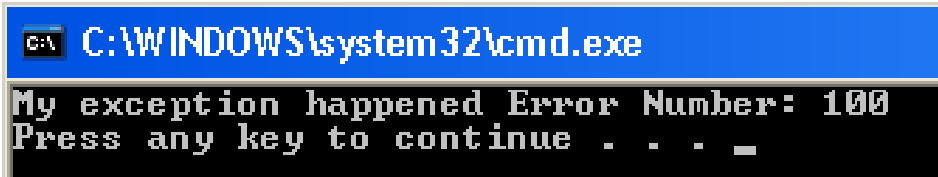
```
class MemoryException: public exception {
public:
    virtual const char* what() const throw() {
        return "My exception happened";
    }
};

int main(){
    char * buffer; int size;
    MemoryException me;
    try{
        cout << "Enter array size "; cin >> size;
        buffer = new char[size];
        if (buffer == 0)
            throw me;
    }
    catch(MemoryException e){
        cout << "Exception : " << e.what() << endl;
    }
}
```

Exception Information

Information relating to the exception can be passed to the derived exception object's constructor and later displayed within the error handler.

```
class MyException: public exception {  
public:  
    int data;  
    MyException(int d):data(d){ }  
    virtual const char* what() const throw() {  
        return "My exception happened";  
    }  
};  
  
int main(){  
    MyException ex(100);  
    try {  
        throw ex;  
    }  
    catch (MyException& e) {  
        cout << e.what() << " Error Number: " << e.data << endl;  
    }  
}
```



A screenshot of a Windows command prompt window. The title bar is blue and contains the text "C:\WINDOWS\system32\cmd.exe". The command prompt itself has a black background with white text. It displays the output of the program: "My exception happened Error Number: 100" followed by a prompt "Press any key to continue . . . _".

Re-throwing an Exception

- Once an exception is caught in the catch block there may be occasions when it should be thrown again.
- This can be achieved through the throw keyword

```
try {  
    throw exceptionIdentifier;  
}  
catch(...){  
    throw;  
}
```

- There is no need to include the exception identifier when re-throwing the exception.
- Ensure your code catches the re-thrown exception.

Rethrow Example

```
class AnotherExceptionExample{
private:
    GeneralException generalError;
public:
    void DoSomething()throw(GeneralException){
        try{
            throw generalError;
        }
        catch(GeneralException& e){
            cout << e.what() << endl;
            throw;
        }
    }
    void Controller(){
        try{
            DoSomething();
        }
        catch(GeneralException& e){
            cout << e.what() << endl;
        }
    }
};
```

Assertions

- A pre-processor macro that evaluates a condition.
- An assert specifies a condition that should hold true. If not the execution of the program is terminated.
- The condition must evaluate to an integer value where 0 is false any other number true

```
assert(condition);
```

- The assert command will print an error message on the console if the condition evaluates to false.
- The error message includes the expression whose assertion failed and the line number and source file.
- When using assertions be sure to include the header file

```
#include <assert.h>
```

Assertions

- Assertions can be turned off by including the directive below before the assert.h include.

```
#define NDEBUG
```

- Assertions are typically used to check the validity of parameter values passed to functions and array bounds checking.

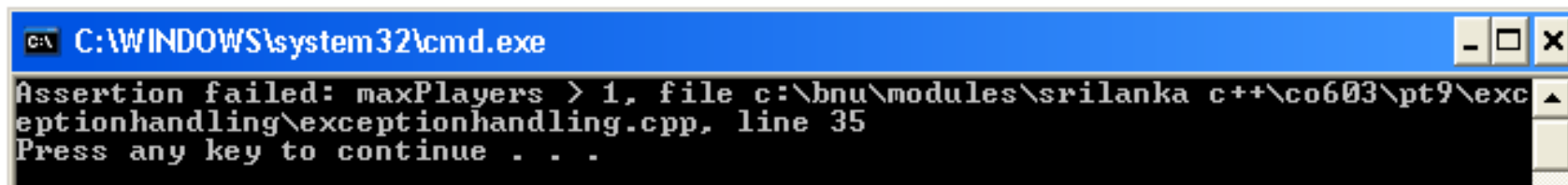
Assertions Example

```
class Game{
private:
    int maxPlayers;
    Player **players;
public:
    Game(int maxPlayers):maxPlayers(maxPlayers){
        assert(maxPlayers > 1);
        players = new Player*[maxPlayers];
    }
    Player* GetPlayer(int index){
        assert(index >=0 && index < maxPlayers);
        return players[index];
    }
};

int main(){
    Game *myGame = new Game(1);
    Player *opponent = myGame->GetPlayer(2);
}
```

Assertions Example

- Here we use assert to ensure the game has at least two players.
- And that we check the array bounds is correct, before returning one of the players within the member function GetPlayer.
- Both assertions would fail as the game was created with one player and their index would be 0.
- The compiler will only identify the first failed assertion.
- Execution of the program will be terminated.



```
C:\WINDOWS\system32\cmd.exe
Assertion failed: maxPlayers > 1, file c:\bnu\modules\srilanka c++\co603\pt9\exceptionhandling\exceptionhandling.cpp, line 35
Press any key to continue . . .
```

The screenshot shows a Windows command prompt window with a blue title bar. The title bar text is "C:\WINDOWS\system32\cmd.exe". The command prompt area is black with white text. It displays an assertion failure message: "Assertion failed: maxPlayers > 1, file c:\bnu\modules\srilanka c++\co603\pt9\exceptionhandling\exceptionhandling.cpp, line 35". Below this message, it says "Press any key to continue . . .". The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

Summary

- As with a number of OO languages C++ includes an exception handling mechanism.
- Exceptions allow the separation of error detection and handling.
- Exceptions provide a vital tool in developing robust solutions.