

MODULE 21

EXCEPTION HANDLING

In the worst case, there must be an emergency exit!

My Training Period: hours

Note:

The compiler used to compile the program examples in this Module is Visual Studio 6.0®, Win32 Empty Console Mode application.

There is no C++ standard `<exception>` class found in my Borland® C++ 5.02 compiler. Check your compiler documentation :o) and don't forget to install any patches and Service Packs if any. For Borland you may try Borland C++ Builder 5.5, 6.0 or multiplatform C++ BuilderX. Examples also tested using VC++ .Net

Abilities

- Able to understand and use C++ exception handlings.

21.1 Introduction

- When we develop a program, we expect the program does what it is supposed to do without any error. Many operations, including object instantiation and file processing, are subject to failures that may go beyond errors. Out-of-memory conditions, for instance, can occur even when your program is running correctly.
- As an example, for typical application program the highest layer may consist of graphic user interface (GUI) part that provide interface for users. These high-level components interact with objects, which in turn encapsulate the application programming interface (API) routines.
- At a lower level, the API routines may interact with the operating system. The operating system itself invokes system services that deal with low-level hardware resources such as physical memory, file systems, and security modules. In general, runtime errors are detected in these lower code layers should not be handled by themselves.
- To handle an error appropriately, higher-level components have to be informed that an error has occurred. Generally, error handling consists of detecting an error and notifying the components that are in charge. These components in turn attempt to recover from the error or terminate the program properly.
- From the simplest one, we may use a proper prompting, for example:

Enter two integer separated by space:

- But what about if the user enter other than integer and not separate it by space? At least there must be a prompt message or an alert dialog box, if the invalid data entered such as classic messages **Abort**, **Retry**, **Ignore** where:

Message	Description
Retry	Debug the assertion or get help on asserts.
Ignore	Ignore the assertion and continue running the program.
Abort	Halt execution of the program and end the debugging session.

Table 21.1: Abort, Retry and Ignore

- Throughout this tutorial also, you should have encountered several mechanism used, such as conditional statements using the `if` statements combined with `exit()`, `abort()` and `terminate()` functions, when there are errors, the program just terminate with some error message, passing the control back to operating system. Some of the messages may be very useful for our debugging process.
- We also have had used the `assert()` function to test the validity of the program expressions as discussed in [Module 10](#).
- But program logic can't be proved correct under all situation, we must ready for this situation by providing the emergency exit for them.
- C++ provides two methods to handle this anomalous situation called **exceptions**, that are using **assertions** and **exceptions**.

21.2 Assertions

- Assertion has been discussed in [Module 10](#) and it should be a revision here.
- Same as C, C++ also supports assertion programming. Programmer specifies checks for correct conditions to continue program execution. We use `assert.h` library for standard C and `<cassert>` for C++, something like this:

```
//C++ and standard C
#include <assert.h>
//#include <cassert>
```

- Preprocessor macro `assert ()` used to provide assertion processing. Macro expects an **expression** with an **integral value**, for example:

```
//assertion macro
assert(expression);
```

- Program segment example:

```
cout<<"Enter an integer: "<<endl;
cin>>p;
if(p!=0)
    cout<<"p x p x p = "<<p*p*p<<endl;
else
    //0 - normal exit, non-zero-some error
    exit(1);
```

If (p!=0)	The program will continue normally.
If (p==0)	The assertion fails, error message displayed in the following form and program terminates. <i>Assertion failed: expression, file filename, line number</i>

- Assertion processing can be disabled by enabling the `NDEBUG` (no debug):

```
//turn assertion checking off
#define NDEBUG

//undefined the NDEBUG, turn on the assertion if
//#defined DEBUG has been defined...
#undef NDEBUG
```

- Assertion processing typically used only during program development and debugging. The `assert` expression is not evaluated in the Release Version of your program. Typically, assertions can be used for:
 - Catching the program logic errors. Use assertion statements to catch logic errors. You can set an assertion on a condition that must be true according to your program logic. The assertion only has an effect if a logic error occurs.
 - Checking the results of an operation. Use assertion statements to check the result of an operation. Assertions are most valuable for testing operations which results are not so obvious from a quick visual inspection.
 - Testing the error conditions that supposed to be handled. Use assertions to test for error conditions at a point in your code where errors supposed to be handled.

21.3 C Exception - structured exception handling (Microsoft® implementation)

- This part presented here just as a comparison and discussion to the standard C++.
- A structured exception handler has no concept of objects or typed exceptions, it cannot handle exceptions thrown by C++ code; but, C++ **catch** handlers can handle C exceptions.
- So, the C++ exception handling syntax using **try, throw...catch** is not accepted by the C compiler, but structured exception handling syntax (Microsoft® implementation) using **__try, __except, __finally** is supported by the C++ compiler.
- The major difference between structured exception handling and C++ exception handling is that the C++ exception handling deals with **types**, while the C structured exception handling deals with exceptions of one type specifically, **unsigned int**.

- C exceptions are identified by an unsigned integer value, whereas C++ exceptions are identified by data type.
- When an exception is raised in C, each possible handler executes a filter that examines the C exception context and determines whether to accept the exception, pass it to some other handler, or ignore it whereas when an exception is thrown in C++, it may be of any type.
- C structured exception handling model is referred to as what is called asynchronous, which the exceptions occur secondary to the normal flow of control whereas the C++ exception handling mechanism is fully synchronous, which means that exceptions occur only when they are invoked or thrown.
- If a C exception is raised in a C++ program, it can be handled by a structured exception handler with its associated filter or by a C++ `catch` handler, whichever is dynamically closer to the exception context.
- The following is a program example of the C++ program raises a C exception inside a C++ `try` block:

```
//C structured exception handling
//and C++ exception handling
#include <iostream.h>

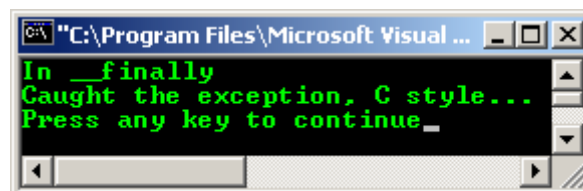
//function prototype...
void TestCFunct(void);

int main()
{
    //C++ try block...
    try
    {
        //function calls...
        TestCFunct();
    }

    //catch block...
    catch(...)
    {
        cout<<"Caught the exception, C style..."<< endl;
    }
    return 0;
}

//function definition...
void TestCFunct()
{
    //structured handling exception...
    __try
    {
        int p, r = 2, q = 0;
        //exception should be raised here
        //divide by 0...
        p = r*(10/q);
    }
    __finally
    {
        cout<<"In __finally" << endl;
        //finding the appropriate catch...
    }
}
```

Output:



- Besides that, C's exception that uses the `set jmp ()` and `long jmp ()` functions, do not support C++ object semantics.
- Using these functions in C++ programs may lesser the performance by preventing optimization on local variables.
- It is better to use the C++ exception handling `try`, `throw...catch` constructs instead.

21.4 C++ Exception

- An exception occurs when an unexpected error or unpredictable behaviors happened on your program not caused by the operating system itself. These exceptions are handled by code which is outside the normal flow of control and it needs an emergency exit.
- Compared to the structured exception handling, returning an integer as an error flag is problematic when dealing with objects. The C++ exception-handling can be a full-fledged object, with data members and member functions.
- Such an object can provide the exception handler with more options for recovery. A clever exception object, for example, can have a member function that returns a detailed verbal description of the error, instead of letting the handler look it up in a table or a file.
- C++ has incorporated three operators to help us handle these situations: `try`, `throw` and `catch`.
- The following is the `try`, `throw...catch` program segment example:

```
try
{
    buff = new char[1024];
    if(buff == 0)
        throw "Memory allocation failure!";
}

//catch what is thrown...
catch(char* strg)
{
    cout<<"Exception raised: "<<strg<<endl;
}
```

- In grammar form:

The try-block:

```
try
{compound-statement handler-list
    handler-list here
```

The throw-expression:

```
throw expression
}
```

The handler:

```
catch (exception-declaration) compound-statement
exception-declaration:
type-specifier-list here
}
```

- Let discuss in detail one by one.

21.4.1 try

- A `try` block is a group of C++ statements, enclosed in curly braces { }, that might cause an exception. This grouping restricts the exception handlers to the exceptions generated within the `try` block. Each `try` block may have one or more associated `catch` blocks.
- If no exception is thrown during execution of the guarded section, the `catch` clauses that follow the `try` block are not executed or bypassed. Execution continues at the statement after the last `catch` clause following the `try` block in which the exception was thrown.
- If an exception is thrown during execution of the guarded section or in any routine the guarded section calls either directly or indirectly such as functions, an exception object will be created from the object created by the `throw` operand.
- At this point, the compiler looks for a `catch` clause in a higher execution context that can handle an exception of the type thrown or a `catch` handler that can handle any type of exception. The **compound-statement** after the `try` keyword is the guarded section of code.

21.4.2 throw

- The `throw` statement is used to throw an exception and its value to a matching `catch` exception handler. A regular `throw` consists of the keyword `throw` and an expression. The **result type** of the expression determines which `catch` block receives control.
- Within a `catch` block, the current exception and value may be re-thrown simply by specifying the `throw` keyword alone that is without the expression.
- The `throw` is syntactically similar to the operand of a `return` statement but here, it returns to the `catch` handler.

21.4.3 catch

- A `catch` block is a group of C++ statements that are used to handle a specific thrown exception. One or more `catch` blocks, or **handlers**, should be placed after each `try` block. A `catch` block is specified by:
 1. The keyword `catch`
 2. A `catch` parameter, enclosed in parentheses (), which corresponds to a **specific type** of exception that may be thrown by the `try` block
 3. A group of statements, enclosed in curly braces { }, whose purpose is to handle the exception
- The **compound-statement** after the `catch` keyword is the exception handler, and catches or handles the exception thrown by the *throw-expression*.
- The **exception-declaration** statement part indicates the type of exception the clause handles. The type can be any valid data type, including a C++ class.
- If the **exception-declaration** statement part is just an ellipsis (...) such as,

```
catch(...)
```

- Then, the `catch` clause will handle any type of exception, including C exceptions and system or application generated exceptions such as divide by zero, memory protection and floating-point violations. Such a handler must be the last handler for its `try` block acting as default catch.
- The `catch` handlers are examined in order of their appearance following the `try` block. If no appropriate handler is found, the next dynamically enclosing `try` block is examined. This process continues until the outermost enclosing `try` block is examined if there are more than one `try` block.
- If a matching handler is still not found, or if an exception occurs while **unwinding**, but before the handler gets control, the predefined run-time function `terminate()` is called. If an exception occurs after throwing the exception, but before the unwinding begins, `terminate()` is also called.
- The `catch` block must go right after the `try` block without any line of codes between them.
- The order in which `catch` handlers appear is important, because handlers for a given `try` block are examined in order of their appearance. For example, it is an error to place the handler for a base class before the handler for a derived class.
- After a matching `catch` handler is found, subsequent handlers are not examined. That is why an ellipsis `catch(...)` handler must be the last handler for its `try` block.
- Besides that, `catch` may be overloaded so that it can accept different types as parameters. In that case the `catch` block executed is the one that matches the type of the exception sent through the parameter of `throw`
- Program example:

```
//simple try, throw...catch
#include <iostream.h>

int main()
{
    //declare char pointer
    char* buff;

    //try block...
    try
    {
        //allocate storage for char object...
        buff = new char[1024];

        //do a test, if allocation fails...
        if(buff == 0)
            throw "Memory allocation failure!";
    }
}
```

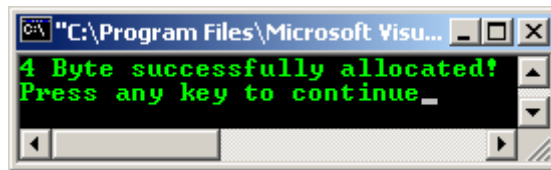
```

        //if allocation successful, display
        //the following message, bypass
        //the catch block...
        else
            cout<<sizeof(buff)<<" Byte successfully allocated!"<<endl;
    }

    //if allocation fails, catch the type...
    //display message...
    catch(char* strg)
    {
        cout<<"Exception raised: "<<strg<<endl;
    }
    return 0;
}

```

Output:



- Program example for multiple catch:

```

//exception: multiple catch blocks
#include <iostream.h>
#include <stdlib.h>

int main ()
{
    try
    {
        char * teststr;
        teststr = new char [10];

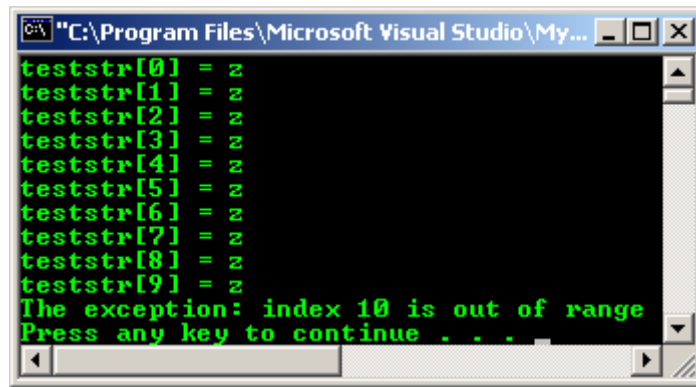
        //test, if memory allocation fails then,
        //throws this error to the matching catch...
        if (teststr == NULL) throw "Allocation failure";

        for (int i=0; i<=15; i++)
        {
            //another test, if n>9, throw this error
            //to the respective catch..
            if (i>9) throw i;
            teststr[i]='z';
            cout<<"teststr["<<i<<" = "<<teststr[i]<<endl;
        }
    }

    //catch the error if, i > 9, by displaying some
    //error message...
    catch (int j)
    {
        cout<<"The exception: ";
        cout<<"index "<<j<<" is out of range"<<endl;
    }
    //catch the error if, allocation fail for *teststr
    //by displaying some error...
    catch (char * strg)
    {
        cout<<"The exception: "<<strg<<endl;
    }
    system("pause");
    return 0;
}

```

Output:



```
C:\Program Files\Microsoft Visual Studio\My...
teststr[0] = z
teststr[1] = z
teststr[2] = z
teststr[3] = z
teststr[4] = z
teststr[5] = z
teststr[6] = z
teststr[7] = z
teststr[8] = z
teststr[9] = z
The exception: index 10 is out of range
Press any key to continue . . .
```

21.5 Catching Exceptions

- Since exceptions are a run-time and not a compile-time feature, standard C++ specifies the rules for matching exceptions to catch-parameters is slightly different from those for finding an overloaded function to match a function call.
- We can define a handler for an object of type named `Type` several different ways. In the following examples, the variable `test` is optional, just as the ordinary functions in C++:

```
catch(Type test)
catch(const Type test)
catch(Type & test)
catch(const Type& test)
```

- Such handlers can catch exception objects of type `Type1` if:
 1. `Type` and `Type1` are the same type, or
 2. `Type` is an accessible base class of `Type1` at the throw point, or
 3. `Type` and `Type1` are pointer types and there exists a standard pointer conversion from `Type1` to `Type` at the throw point. `Type` is an accessible base class of `Type1` if there is an inheritance path from `Type1` to `Type` with all public derivations.
- For the third rule, let `Type1` be a type pointing to type `Type2`, and `Type` be a type that points to type `Type3`. Then there exists a standard pointer conversion from `Type1` to `Type` if:
 1. `Type` is the same type as `Type1`, except it may have added any or both of the qualifiers `const` and `volatile`, or
 2. `Type` is `void*`, or
 3. `Type3` is an unambiguous, accessible base class of `Type2`. `Type3` is an unambiguous base class of `Type2` if `Type2`'s members can refer to members of `Type3` without ambiguity (this is usually only a concern with multiple inheritance).
- The C++ type conversion is discussed in next Module, Module 22.
- As conclusion, for these rules, the exceptions and catch parameters must either match exactly, or the exception caught by pointer or reference must be derived from the type of the catch parameter.
- For example, the following exception is not caught:

```
//mismatch type, throw integer type
//catch the double type...
#include <iostream.h>

void Funct();

int main()
{
    try
    { Funct(); }
    catch(double)
    { cerr<<"caught a double type..."<<endl; }
    return 0;
}

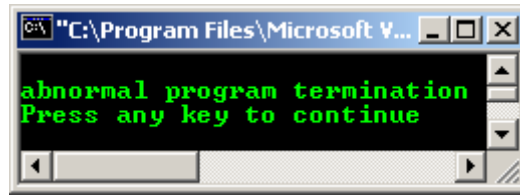
void Funct()
{
```

```

        //3 is not a double but int
        throw 3;
    }

```

Output:



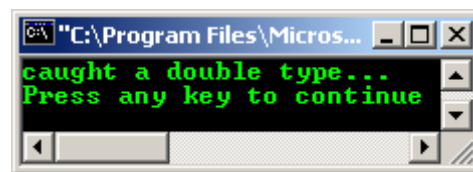
- Change the following statement

```

        throw 3;      to      throw 4.123;

```

- Recompile and rerun, the program output should be as follows:



- As a summary, when an exception is thrown, it may be caught by the following types of `catch` handlers:
 - A handler that can accept any type (using the ellipsis syntax).
 - A handler that accepts the same type as the exception object; because it is a copy, **const** and **volatile** modifiers are ignored.
 - A handler that accepts a reference to the same type as the exception object.
 - A handler that accepts a reference to a **const** or **volatile** form of the same type as the exception object.
 - A handler that accepts a base class of the same type as the exception object; since it is a copy, **const** and **volatile** modifiers are ignored. The **catch** handler for a base class must not precede the **catch** handler for the derived class.
 - A handler that accepts a reference to a base class of the same type as the exception object.
 - A handler that accepts a reference to a **const** or **volatile** form of a base class of the same type as the exception object.
 - A handler that accepts a pointer to which a thrown pointer object can be converted via standard pointer conversion rules.
- C++ exception is automatically call destructor functions during the **stack unwinding** process, for all local objects constructed before the exception was thrown.
- Program example.

```

//exception, class and destructor
#include <iostream.h>

void TestFunc(void);

//class Test1 declaration...
class Test1
{
public:
    Test1(){};
    ~Test1(){};
    const char *TestShow() const
    {
        cout<<"In class member function *TestShow():\n";
        return " Exception in Test1 class.";
    }
};

//another class declaration, DestrTest...
class DestrTest

```



```

{
    public:
        DestrTest();
        ~DestrTest();
};

//constructor class implementation
DestrTest::DestrTest()
{
    cout<<"Next, in constructor DestrTest():\n";
    cout<<"  Constructing the DestrTest...\n";
}

//destructor class implementation
DestrTest::~DestrTest()
{
    cout<<"Next, in destructor ~DestrTest():\n";
    cout<<"  Destructing the DestrTest...\n";
}

void TestFunc()
{
    //instantiate an object, constructor invoked...
    DestrTest p;
    cout<<"Next in TestFunc(): \n  Throwing Test1 type exception...\n";
    //first throw...
    throw Test1();
}

int main()
{
    cout<<"Starting in main()...\n";
    try
    {
        cout<<"Now, in the try block: \n  Calling TestFunc()...\n";
        TestFunc();
    }
    //instantiate another object, constructor invoked...
    catch(Test1 q)
    {
        cout<<"Next, in catch handler:\n";
        cout<<"  Caught Test1 type exception...\n";
        cout<<q.TestShow()<<"\n";
    }
    catch(char *strg)
    {
        cout<<"Caught char pointer type exception: "<<strg<<"\n";
    }

    cout<<"Back in main...\n";
    return 0;
}

```

Output:

```

C:\Program Files\Microsoft Visual Studio\...
Starting in main()...
Now, in the try block:
  Calling TestFunc()...
Next, in constructor DestrTest():
  Constructing the DestrTest...
Next in TestFunc():
  Throwing Test1 type exception...
Next, in destructor ~DestrTest():
  Destructing the DestrTest...
Next, in catch handler:
  Caught Test1 type exception...
In class member function *TestShow():
  Exception in Test1 class.
Back in main...
Press any key to continue

```

21.6 Exception Processing-Stack Unwinding

- When an exception is thrown, the runtime mechanism first searches for an appropriate matching handler (`catch`) in the current scope. If no such handler exists, control is transferred from the current scope to a higher block in the calling chain or in outward manner.
- Iteratively, it continues until an appropriate handler has been found. At this point, the stack has been unwound and all the local objects that were constructed on the path from a `try` block to a `throw` expression have been destroyed.
- The run-time environment invokes destructors for all automatic objects constructed after execution entered the `try` block. This process of destroying automatic variables on the way to an exception handler is called **stack unwinding**.
- During the unwinding the stack, objects on stack are destroyed, local variables, local class objects destructors are called and program goes back to a normal state.
- The stack unwinding process is very similar to a sequence of `return` statements, each returning the same object to its caller.
- In the absence of an appropriate handler, the program terminates. However, C++ ensures proper destruction of local objects only when the thrown exception is handled. Whether an uncaught exception causes the destruction of local objects during stack unwinding is implementation-dependent.
- To ensure that destructors of local objects are invoked in the case of an uncaught exception, you can add a `catch(...)` statement in `main()`. For example:

```
int main()
{
    try
    {
        //throw exceptions...
        throw Something;
    }
    //handle expected exceptions
    catch(TheSomething)
    {
        //handle all the exceptions...
    }
    //ensure proper cleanup in the case
    //of an uncaught exception
    catch(...)
    {
        //catch other things...
    }
}
```

- A `throw` expression with **no operand** re-throws the exception currently being handled. Such an expression should appear only in a `catch` handler or in a function called from within a `catch` handler.
- The re-thrown exception object is the original exception object (not a copy). For example:

```
try
{
    throw SomeException();
}

//Handle all exceptions
catch(...)
{
    //Respond (perhaps only partially) to exception
    //...
    //re throw without operand...
    //Pass exception to some other handler
    throw;
}
```

- An empty `throw` statement tells the compiler that the function does not throw any exceptions. For example:

```
//empty throw statement
#include <iostream.h>

//this empty throw will be ignored...
void Nothing() throw()
{ cout<<"In Nothing(), empty throw..."<<endl; }

void SomeType() throw(double)
{
    cout<<"In SomeType, will throw a double type..."<<endl;
    throw(1.234);
}
```

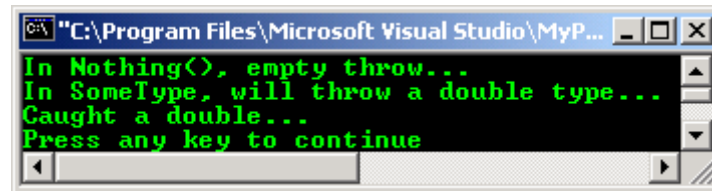
```

}

void main()
{
    try
    {
        Nothing();
        SomeType();
    }
    catch (double)
    { cout<<"Caught a double..."<<endl; }
}

```

Output:



21.7 Uncaught/Unhandled Exception

- When the system can't find a handler for an exception, it calls the standard library function `terminate()`, which by default aborts the program. You can substitute your own termination function by passing a pointer to it as a parameter to the `set_terminate()` library function.
- An exception caught by a pointer can also be caught by a `void*` handler. In the following program example, exception is caught, since there is a handler for an accessible base class:

```

#include <iostream.h>
//base class
class Test1 {};
//derived class
class Test2 : public Test1 {};

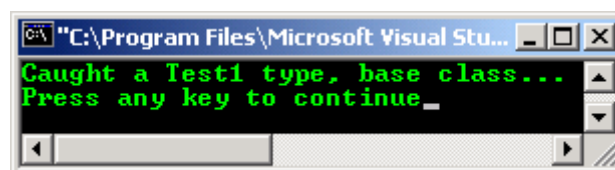
void Funct();

int main()
{
    try
    {
        //function call, go to Funct()
        Funct();
    }
    catch(const Test1&)
    {
        cerr<<"Caught a Test1 type, base class..."<<endl;
    }
    return 0;
}

//throw function definition
//a throw of Test2 type, derived class...
void Funct()
{
    throw Test2();
    //next, find the catch handler
}

```

Output:



- Another program example for terminating the `try` block:

```

#include <iostream.h>

```

```

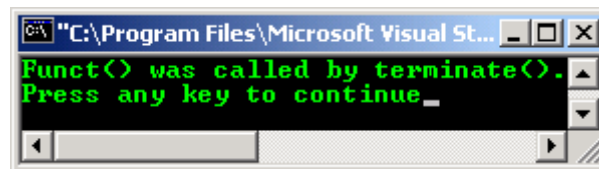
//exit()
#include <stdlib.h>
//set_terminate()
#include <exception>

void Funct()
{
    cout<<"Funct() was called by terminate()."<<endl;
    //0-normal exit, non zero-exit with some error
    exit(0);
}

int main()
{
    try
    {
        set_terminate(Funct);
        //No catch handler for this exception
        throw "Out of memory!";
    }
    catch(int)
    { cout<<"Integer exception raised."<<endl; }
    return 0;
}

```

Output:



21.8 Exception Specifications

- Exception specifications are used to provide summary information about what exceptions can be thrown out of a function. Exceptions not listed in an exception specification should not be thrown from that function.
- An exception specification consists of the keyword `throw` after the function's parameter list, followed by a list of potential exceptions, for example:

```
void test(int somecode) throw (bad_code, no_auth);
```

- An exception specification isn't considered a part of a function's type. Therefore, it doesn't affect overload resolution. That means pointers to functions and pointers to member functions may contain an exception specification, for example:

```
void (*PtrFunct)(double) throw(string, double);
```

- `PtrFunct` is a pointer to a function that may throw `string` or `double`. You can assign to a function whose exception specification is as restrictive as, or more restrictive than `PtrFunct`'s exception specification.
- An exception specification P is said to be more restrictive than an exception specification Q if the set of exceptions P contains is a subset of Q 's exceptions. In other words, P contains every exception in Q but not vice versa. For example:

```

void (*PtrFunct)(double) throw(string, double);
...
//more restrictive than PtrFunct:
void One(double) throw (string);
//as restrictive as PtrFunct:
void Two(double) throw (string, double);
//less restrictive than PtrFunct:
void Three(double) throw (string, double, bool);
PtrFunct = One; //OK
PtrFunct = Two; //OK
PtrFunct = Three; //error, Three is not subset of the PtrFunct

```

- A function with **no exception-specification** allows all exceptions. A function with an **empty exception specification** doesn't allow any exceptions, for example:

```

class Test
{
public:
    //may throw any exception
    int One(char *VarPtr);
    //doesn't throw any exception
    int Two(double *VarPtr1) throw();
};

```

- Exception specifications are enforced at the runtime. When a function violates its exception specification, `unexpected()` function is called.
- The `unexpected()` function invokes a user-defined function that was previously registered by calling `set_unexpected()`.
- If no function was registered with `set_unexpected()`, `unexpected()` calls `terminate()` which aborts the program unconditionally.
- The following table summarizes C++'s implementation of exception specifications:

Exception specification	Meaning
<code>throw()</code>	The function does not throw any exception.
<code>throw(...)</code>	The function can throw an exception.
<code>throw(type)</code>	The function can throw an exception of type <i>type</i> .

Table 21.2: Exception specification

- The following are examples of the exception specification implementation.

Example	Description
<code>void Funct() throw(int)</code>	The function may throw an <code>int</code> exception.
<code>void Funct() throw()</code>	The function will throw no exceptions.
<code>void Funct() throw(char*, T)</code>	The function may throw a <code>char*</code> and/or a <code>T</code> , user defined type exception.
<code>void Funct() or void Funct(...)</code>	The function may throw anything.

Table 21.3: Exception specification example

- If exception handling is used in an application, there must be one or more functions that handle thrown exceptions.
- Any functions called between the one that throws an exception and the one that handles the exception must be capable of throwing the exception. However, explicit exception specifications are not allowed on C functions.

```

//exception specification
#include <iostream.h>

//handler function
void handler()
{cout<<"In the handler()\n";}

//int throw...
void Funct1(void) throw(int)
{
    static int x = 1;
    cout<<"Funct1() call #"<<x++<<endl;
    cout<<"About to throw 1\n";
    if (1)
        throw 1;
}

//empty throw...
void Funct5(void) throw()
{
    try
    {Funct1();}
    catch(...)
    {handler();}
}

// invalid, doesn't handle the int exception thrown from Funct1()
// void Funct3(void) throw()
// {

```

```

//  Funct1();
//}

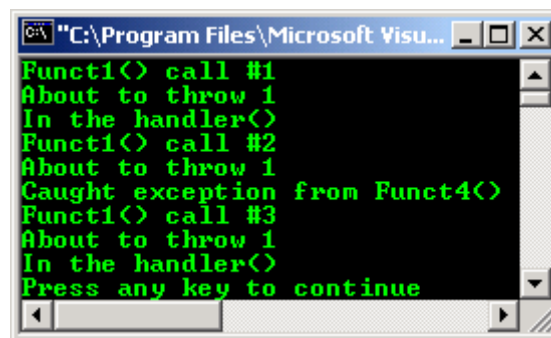
void Funct2(void)
{
    try
    {Funct1();}
    catch(int)
    {handler();}
}

//assume extern "C" functions don't throw exceptions
extern "C" void Funct4(void);
void Funct4(void)
{Funct1();}

int main()
{
    Funct2();
    try
    {Funct4();}
    catch(...)
    {cout<<"Caught exception from Funct4()\n";}
    Funct5();
    return 0;
}

```

Output :



21.9 Exception Handling Overhead

- The extra overhead associated with the C++ exception handling mechanism may increase the size of executable files and slow your program execution.
- So, exceptions should be used only in truly exceptional situations. Exception handlers should not be used to redirect the program's normal flow of control.
- For example, an exception should not be thrown in cases of potential logic or user input errors, such as the overflow of an array boundary. In these cases, simply returning an error code by using for example, the conditional if statement may be simpler and more concise.

21.10 Standard Exceptions

- The C++ exception class serves as the base class for all exceptions thrown by certain expressions and by the Standard C++ Library.

Class hierarchy	Description
exception	
bad_alloc	Thrown by new, an allocation request fails.
bad_cast	Thrown by dynamic_cast when failed cast to a reference type.
bad_exception	Thrown when an exception doesn't match any catch clause.
bad_typeid	Thrown by typeid operator when the operand for typeid is a NULL pointer.
The logical errors are normally caused by programmer mistakes.	
logic_error	As the base class for all exceptions thrown to report errors presumably detectable before the program executes, such as violations of logical preconditions.
domain_error	As the base class for all exceptions thrown to report a domain error.

invalid_argument	As the base class for all exceptions thrown to report an invalid argument.
length_error	As the base class for all exceptions thrown to report an attempt to generate an object too long to be specified.
out_of_range	As the base class for all exceptions thrown to report an argument that is out of its valid range.
The run-time errors normally occur because of mistakes in either the library functions or in the run-time system	
runtime_error	As the base class for all exceptions thrown to report errors presumably detectable only when the program executes.
overflow_error	As the base class for all exceptions thrown to report an arithmetic overflow.
range_error	As the base class for all exceptions thrown to report a range error.
underflow_error	As the base class for all exceptions thrown to report an arithmetic underflow.
ios_base::failure	The member class serves as the base class for all exceptions thrown by the member function <code>clear()</code> in template class <code>basic_ios</code> .

Table 21.4: exception class

- Logical and run time errors are defined in Standard C++ `<stdexcept>` header file and this `<stdexcept>` is a derived class from the `exception` class where the Standard C++ header file is `<exception>`.
- Do not confuse with the `exception` class and `<exception>` header file, they refer to different things here. Header is denoted by the angled bracket `<>`. Exception class definition is shown below:

```
class exception
{
public:
    exception( ) throw( );
    exception(const exception& right) throw( );
    exception& operator=(const exception& right) throw( );
    virtual ~exception( ) throw( );
    virtual const char *what( ) const throw( );
};
```

- Some functions of the standard C++ library send exceptions that can be caught by including them within a `try` block. These exceptions are sent with a class derived from `std::exception` as their type. It is better to use these exceptions instead of creating your own, because these exceptions have been tested.
- Because this is a class hierarchy, if you include a `catch` block to capture any of the exceptions of this hierarchy using the argument by reference that is by adding an ampersand, `&` after the type, you will also capture all the derived ones.
- Referring to the hierarchy of the exception, exceptions are caught in a bottom-down hierarchy: Specific derived classes exceptions are handled first, followed by less specific groups of exceptions that is, up to the base classes and, finally, a `catch(. . .)` handler:
- Handlers of the specific derived objects must appear before the handlers of base classes. This is because handlers are tried in order of appearance. It's therefore possible to write handlers that are never executed; for example, by placing a handler for a derived class after a handler for a corresponding base class.
- You can use the classes of standard hierarchy of exceptions to throw your exceptions or derive new classes from them.
- The following example catches an exception of type `bad_typeid` (derived from `exception`) that is generated when requesting information about the type pointed to by a `NULL` pointer:

```
//standard exceptions
//program example
#include <iostream.h>
#include <exception>
#include <typeinfo>

class Test1
{
    virtual Funct() {};
};

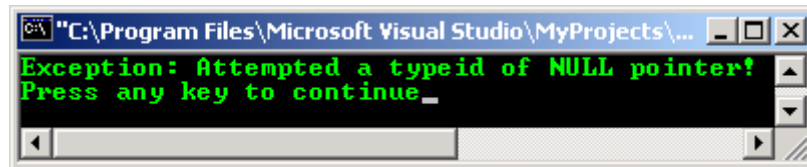
int main ()
{
    try {
```

```

        Test1 * var = NULL;
        typeid (*var);
    }
    catch (std::exception& typevar)
    {
        cout<<"Exception: "<<typevar.what()<<endl;
    }
    return 0;
}

```

Output:



- Another program code segment example:

```

class out_of_range : public logic_error
{
    public:
        out_of_range(const string& message);
};

```

- The value returned by **what** is a copy of **message**.
- Program example:

```

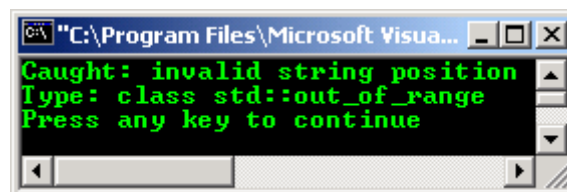
//out_of_range example
#include <string>
#include <iostream>

using namespace std;

int main( )
{
    try
    {
        string strg1("Test");
        string strg2("ing");
        strg1.append(strg2, 4, 2);
        cout<<strg1<<endl;
    }
    catch (exception &e)
    {
        cerr<<"Caught: "<<e.what()<<endl;
        cerr<<"Type: "<<typeid(e).name()<<endl;
    };
    return 0;
}

```

Output



- Other `<exception>` header members are listed in the following tables.

typedef	Description
terminate_handler	A type that describes a pointer to a function suitable for use as a terminate_handler.
unexpected_handler	A type that describes a pointer to a function suitable for use as an unexpected_handler.

Table 21.5: `<exception>` typedef

Member function	Description
set_terminate()	Establishes a new terminate_handler to be called at the termination of the program.
set_unexpected()	Establishes a new unexpected_handler to be when an unexpected exception is encountered.
terminate()	Calls a terminate handler.
uncaught_exception()	Returns true only if a thrown exception is being currently processed.
unexpected()	Calls an unexpected handler.

Table 21.6: <exception> member function

Class	Description
bad_exception	The class describes an exception that can be thrown from an unexpected_handler .
exception	The class serves as the base class for all exceptions thrown by certain expressions and by the Standard C++ Library.

Table 21.7: <exception> class member

- Some simple program examples.

```
//bad_cast
//Need to enable the Run-Time Type Info,
//rtti of your compiler. You will learn
//Typecasting in another Module...
#include <typeinfo.h>
#include <iostream>
using namespace std;

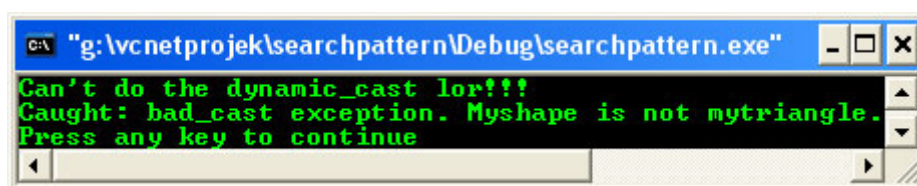
class Myshape
{
public:
    virtual void myvirtualfunc() const {}
};

class mytriangle: public Myshape
{
public:
    virtual void myvirtualfunc() const
    { };
};

int main()
{
    Myshape Myshape_instance;
    Myshape &ref_Myshape = Myshape_instance;

    try {
        //try the run time typecasting, dynamic_cast
        mytriangle &ref_mytriangle = dynamic_cast<mytriangle*>(ref_Myshape);
    }
    catch (bad_cast) {
        cout<<"Can't do the dynamic_cast lor!!!"<<endl;
        cout<<"Caught: bad_cast exception. Myshape is not mytriangle.\n";
    }
}
```

Output:



```
//bad_alloc, first version
//the allocation is OK
```

```

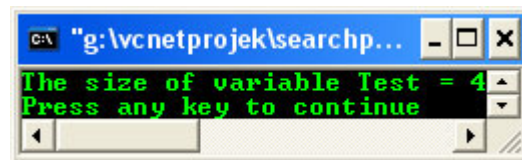
#include <new>
#include <iostream>
using namespace std;

int main()
{
    char* ptr;

    unsigned long int Test = sizeof(size_t(0)/3);
    cout<<"The size of variable Test = "<<Test<<endl;
    try
    {
        //try some allocation...
        //size of an array must not exceed certain bytes
        ptr = new char[size_t(0)/3]
        delete[] ptr;
    }
    catch(bad_alloc &thebadallocation)
    {
        cout<<thebadallocation.what()<<endl;
    };
}

```

Output:



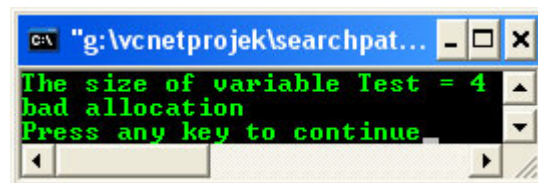
- Let negate/inverse the array size, change the following

```
sizeof(size_t(0)/3)
```

- To the following code.

```
sizeof(~size_t(0)/3)
```

- Recompile and re run the program, the following output should be expected.



```

//set_unexpected
#include <exception>
#include <iostream>
using namespace std;

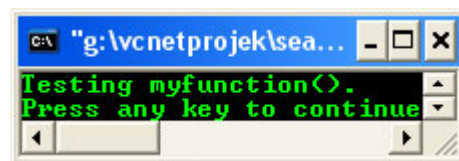
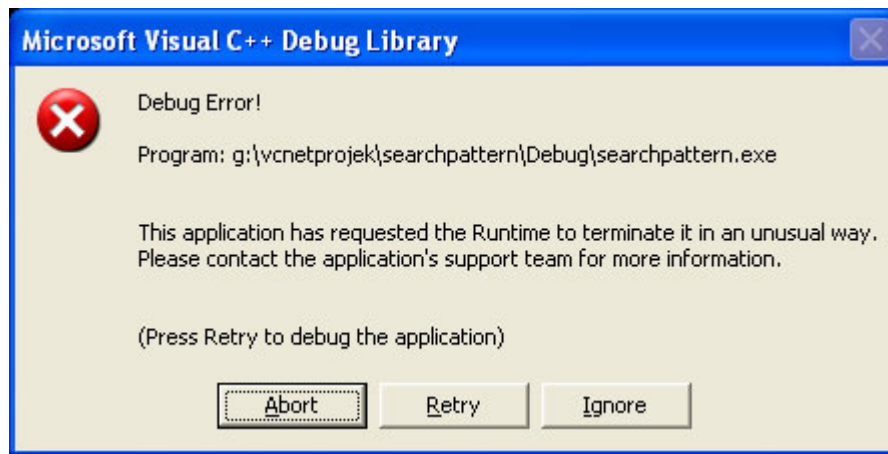
void myfunction()
{
    cout<<"Testing myfunction()."<<endl;
    //terminate() handler
    terminate();
}

int main( )
{
    unexpected_handler oldHandler = set_unexpected(myfunction);
    //unexpected() function call
    unexpected();
}

```

Output:

- Click the Abort button.



- The following example shows the typeid operator throwing a bad_typeid exception.

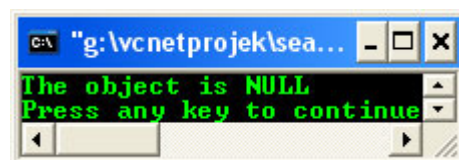
```
//bad_typeid
#include <typeinfo.h>
#include <iostream>
using namespace std;

class Test
{
public:
    //object for a class needs vtable
    //for the rtti...
    Test();
    virtual ~Test();
};

int main()
{
    Test *ptrvar = NULL;

    try {
        //the error condition
        cout<<typeid(*ptrvar).name()<<endl;
    }
    catch (bad_typeid){
        cout<<"The object is NULL"<<endl;
    }
}
```

Output:



```
//domain_error and typeid()
#include <iostream>
using namespace std;

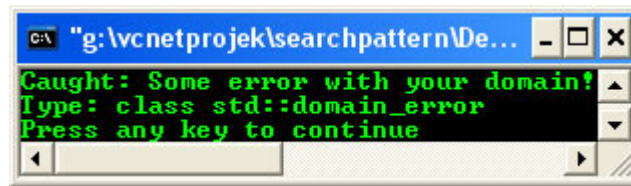
int main()
{
    try
    {
        throw domain_error("Some error with your domain!");
    }
    catch (exception &err)
```

```

    {
        cerr<<"Caught: "<<err.what()<<endl;
        cerr<<"Type: "<<typeid(err).name()<<endl;
    };
}

```

Output:



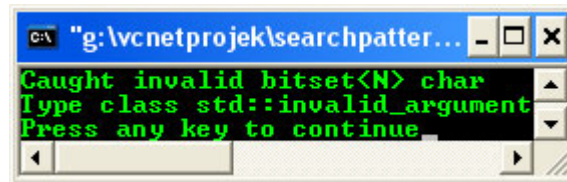
```

//invalid_argument
#include <bitset>
#include <iostream>
using namespace std;

int main()
{
    try
    {
        //binary wrongly represented by char X
        //template based...
        bitset<32> bitset(string("0101001X01010110000"));
    }
    catch (exception &err)
    {
        cerr<<"Caught "<<err.what()<<endl;
        cerr<<"Type "<<typeid(err).name()<<endl;
    };
}

```

Output:



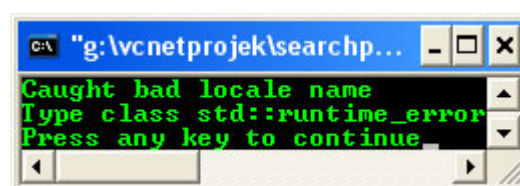
```

//runtime_error
#include <iostream>
using namespace std;

int main()
{
    //runtime_error
    try
    {
        locale testlocale("Something");
    }
    catch(exception &err)
    {
        cerr<<"Caught "<<err.what()<<endl;
        cerr<<"Type "<<typeid(err).name()<<endl;
    };
}

```

Output:



```

//overflow_error

```

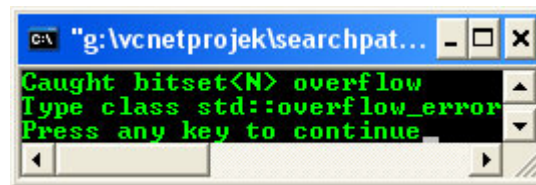
```

//storage reserved is not enough
#include <bitset>
#include <iostream>
using namespace std;

int main()
{
    try
    {
        //template based...
        bitset<100> bitset;
        bitset[99] = 1;
        bitset[0] = 1;
        //to_ulong(), converts a bitset object to the integer
        //that would generate the sequence of bits
        unsigned long Test = bitset.to_ulong();
    }
    catch(exception &err)
    {
        cerr<<"Caught " <<err.what()<<endl;
        cerr<<"Type " <<typeid(err).name()<<endl;
    };
}

```

Output:



```

//range_error
#include <iostream>
using namespace std;

int main()
{
    try
    {
        throw range_error("Some error in the range!");
    }
    catch(exception &Test)
    {
        cerr<<"Caught: " <<Test.what()<<endl;
        cerr<<"Type: " <<typeid(Test).name()<<endl;
    };
}

```

Output:



```

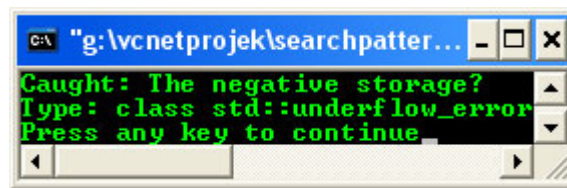
//underflow_error
//negative storage...
#include <iostream>
using namespace std;

int main()
{
    try
    {
        throw underflow_error("The negative storage?");
    }
    catch(exception &Test)
    {
        cerr<<"Caught: " <<Test.what()<<endl;
        cerr<<"Type: " <<typeid(Test).name()<<endl;
    };
}

```

```
}
```

Output:



- Program example compiled using **g++**.

```
/******-except.cpp-*****  
//exception, class and destructor  
#include <iostream>  
using namespace std;  
  
void TestFunc(void);  
  
//class Test1 declaration...  
class Test1  
{  
    public:  
    Test1(){};  
    ~Test1(){};  
    const char *TestShow() const  
    {  
        cout<<"In class member function *TestShow():\n";  
        return " Exception in Test1 class.";  
    }  
};  
  
//another class declaration, DestrTest...  
class DestrTest  
{  
    public:  
    DestrTest();  
    ~DestrTest();  
};  
  
//constructor class implementation  
DestrTest::DestrTest()  
{  
    cout<<"Next, in constructor DestrTest():\n";  
    cout<<" Constructing the DestrTest...\n";  
}  
  
//destructor class implementation  
DestrTest::~DestrTest()  
{  
    cout<<"Next, in destructor ~DestrTest():\n";  
    cout<<" Destructing the DestrTest...\n";  
}  
  
void TestFunc()  
{  
    //instantiate an object, constructor invoked...  
    DestrTest p;  
    cout<<"Next in TestFunc(): \n Throwing Test1 type exception...\n";  
    //first throw...  
    throw Test1();  
}  
  
int main()  
{  
    cout<<"Starting in main()...\n";  
    try  
    {  
        cout<<"Now, in the try block: \n Calling TestFunc()...\n";  
        TestFunc();  
    }  
    //instantiate another object, constructor invoked...  
    catch(Test1 q)  
    {  
        cout<<"Next, in catch handler:\n";  
        cout<<" Caught Test1 type exception...\n";  
    }  
}
```

```

        cout<<q.TestShow()<<"\n";
    }
    catch(char *strg)
    {
        cout<<"Caught char pointer type exception: "<<strg<<"\n";
    }
    cout<<"Back in main...\n";
    return 0;
}

```

[bodo@bakawali ~]\$ g++ except.cpp -o except

[bodo@bakawali ~]\$./except

```

Starting in main()...
Now, in the try block:
    Calling TestFunc()...
Next, in constructor DestrTest():
    Constructing the DestrTest...
Next in TestFunc():
    Throwing Test1 type exception...
Next, in destructor ~DestrTest():
    Destructing the DestrTest...
Next, in catch handler:
    Caught Test1 type exception...
In class member function *TestShow():
    Exception in Test1 class.
Back in main...

```

-----o0o-----

Further reading and digging:

1. [Check the best selling C/C++, Object Oriented and pattern analysis books at Amazon.com.](#)