module05

Jump to bottom

qingqingqingli edited this page on Jan 18, 2021 · 2 revisions

Table of contents

Nested classes

> Example

```
class Cat
{
public:
    class Leg
    {
          //[...]
    };
};

int main()
{
        Cat somecat; // instantiate the Cat class
        Cat::Leg somecatsleg; // instantiate the Leg class
}
```

Exceptions

- Exceptions are an additional approach to signal errors or unexpected conditions in a program.

- **Handling errors**. The quality of the error-handling code determines how robust a program is and it is usually a major factor in making a program user-friendly.
    - Not all errors are equal though the nature of the error determines how best to deal with it.
    - Exceptions are to deal with error conditions that you don't expect to occur in the normal course of events.
    - A primary advantage of using exceptions to signal these errors is that the *error-handling code is separated completely from the code that caused the error*
- **Exceptions need to be used exceptionally**. If you expect the function to fail often, you don't want to use exception, as calling exception is more resource consuming than returning an error value.
- With `catch`, you can catch a specific type of error messages. To catch any type of error message, you can use `catch(...)` to catch.
- If a statement that is not within a try block throws an exception, or a statement within a `try block` throws an exception that is not caught, the program terminates.
- A `try block` can be followed by several `catch` blocks, each of which handles an exception of a different type. The code in a `catch` block only executes when an exception of a matching type is thrown.
- Throwing an exception leaves the `try block` immediately, so at that point all the automatic objects that have been defined within the `try block` prior to the exception being thrown are destroyed.
- **An exception object must be of a type that can be copied. An object of a class type that has a private copy constructor can't be used as an exception.**
- None of the automatic objects created in the try block exists by the time the handler code is executed is very important. It implies that you must not throw an exception object that's a pointer to an object that is local to the `try block`. It's also why the exception object is copied in the throw process. You can throw objects that are local to the try block, but not pointers to local objects.
- **Catch exception by reference**:

> Example

```
#include <stdexcept>

void test1()
{
        try
    {
        // Do some stuff here
        if (/*there is an error*/)
        {
            throw std::exception();
        }
        else
        {
                // Do some more stuff
        }
    }
        catch (std::exception e)
        {
            // handle the error here
    }
}

void test2()
{
    // Do some stuff here
    if (/*there is an error*/)
    {
        throw std::exception();
    }
    else
    {
        // Do some other stuff
    }
}

void test3()
{
    try
    {
        test2();
    }
    catch (std::exception& e) // catch exception by reference
    {
        // handle error
    }
}

void test4()
{
    class PEBKACException: public std::exception
    {
        public:
            virtual const char* what() const throw() // understand better
            {
                return ("Problem exists between keyboard and chair");
            }
    };
    try
    {
        test3();
    }
    catch (PEBKACException& e) // specific catch
    {
        // handle the fact that the user did something stupid
    }
    catch (std::exception& e) // generic catch
    {
        // handle other exceptions that are like std::exception
    }
}
```

Unhandled exceptions

- If an exception is thrown in a try block and is not handled by any of its catch blocks, then the Standard Library function `std:: terminate() is called. This function is declared in the exception header and calls a predefined default terminate handler function, which in turn calls the Standard Library function std::abort()``` that is declared in the cstdlib header.
- **Custom terminate handler**. The action provided by the default terminate handler can be disastrous in some situations. For example, it may leave files in an unsatisfactory state, or connection to a communications line may be left open. In such cases, you'd want to make sure that things are tidied up properly before the program ends. You can do this by replacing the default terminate handler function with your own version by calling the Standard Library function `std::set_terminate()` .
- Example of your custom terminate function

```
void myHandler()
{
// Do necessary clean-up to leave things in an orderly state...
std::exit(1);
}
```

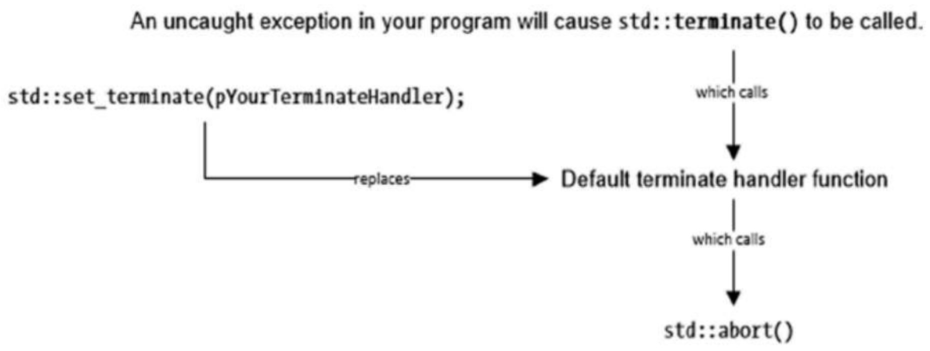
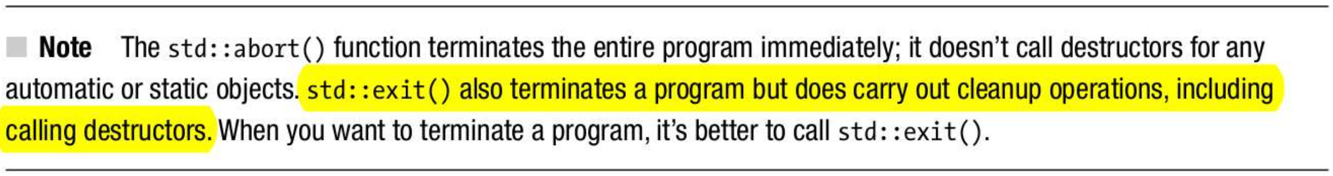

An uncaught exception in your program will cause std::terminate() to be called.
std::set_terminate(pYourTerminateHandler);
which calls
replaces
Default terminate handler function
which calls
std::abort()

Figure 15-3. *Uncaught exceptions*



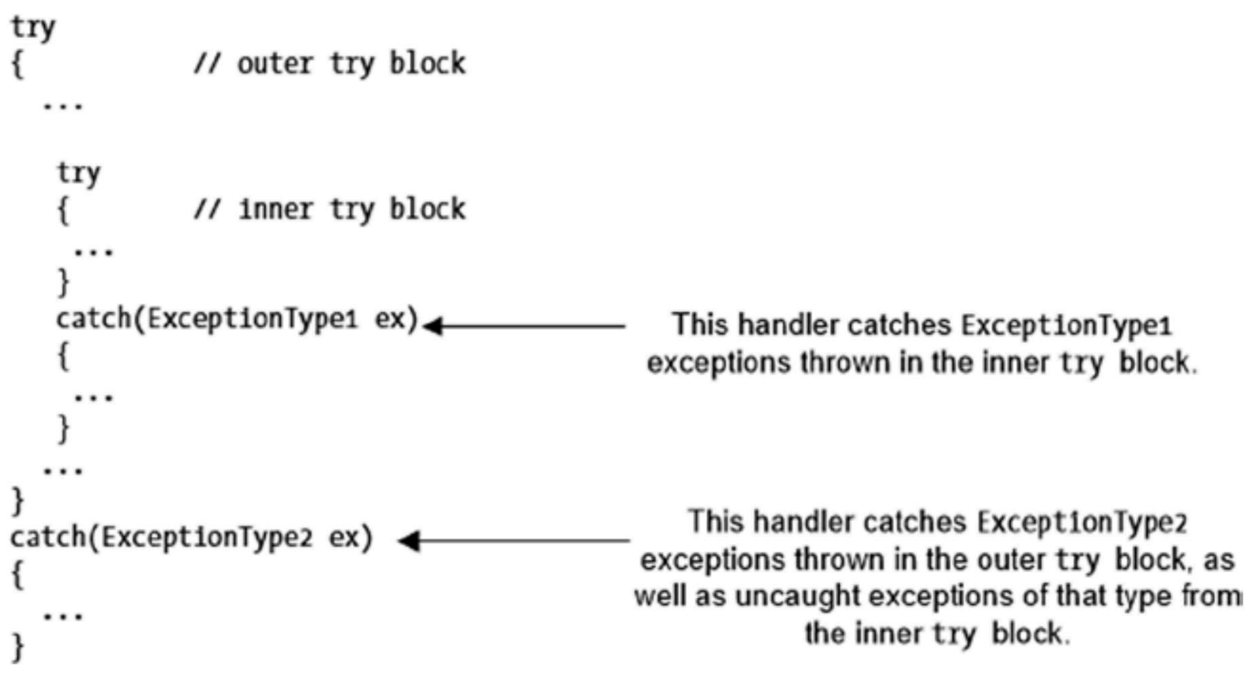
■ **Note** The `std::abort()` function terminates the entire program immediately; it doesn't call destructors for any automatic or static objects. `std::exit()` also terminates a program but does carry out cleanup operations, including calling destructors. When you want to terminate a program, it's better to call `std::exit()`.

Code that causes an exeception to be thrown

- The try blocks enclose code that may throw an exception. However, this doesn't mean that the code that throws an exception must be physically between the braces bounding the try block.
- It only needs to be logically within the try block. If a function is called within a try block, any exception that is thrown and not caught within that function can be caught by one of the catch blocks for the try block.

Nested try blocks

- You can nest a try block inside another try block. Each try block has its own set of catch blocks to handle exceptions that may be thrown within it, and the catch blocks for a try block are only invoked for exceptions thrown within that try block.



```
try
{           // outer try block
  ...

   try
   {        // inner try block
     ...
   }
   catch(ExceptionType1 ex)
   {
     ...
   }
  ...
}
catch(ExceptionType2 ex)
{
  ...
}
```

This handler catches ExceptionType1 exceptions thrown in the inner try block.

This handler catches ExceptionType2 exceptions thrown in the outer try block, as well as uncaught exceptions of that type from the inner try block.

Add custom exception class

> Example

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

- `virtual` : It adds nothing, as the method being overridden is already virtual. It can be omitted.
- `const char* what()` : A member function named what() that takes no arguments and returns a pointer to const char
- `const` : The member function can be called via a const pointer or reference to an instance of this class or a derived class
- `throw()` : throws no exceptions

resources

- What is the meaning of this header (virtual const char* what() const throw())?
- CPP reference: exception
- How do I create and use an array of pointer-to-member-function?

▾ Pages 11

Find a page...

Clone this wiki locally

```
https://github.com/qingqingqingli/CPP.wiki.git
```