Exceptions and Stack Unwinding in C++

Visual Studio 2015

For the latest documentation on Visual Studio 2017, see Visual Studio 2017 Documentation.

In the C++ exception mechanism, control moves from the throw statement to the first catch statement that can handle the thrown type. When the catch statement is reached, all of the automatic variables that are in scope between the throw and catch statements are destroyed in a process that is known as *stack unwinding*. In stack unwinding, execution proceeds as follows:

1. Control reaches the `try` statement by normal sequential execution. The guarded section in the `try` block is executed.

2. If no exception is thrown during execution of the guarded section, the `catch` clauses that follow the `try` block are not executed. Execution continues at the statement after the last `catch` clause that follows the associated `try` block.

3. If an exception is thrown during execution of the guarded section or in any routine that the guarded section calls either directly or indirectly, an exception object is created from the object that is created by the `throw` operand. (This implies that a copy constructor may be involved.) At this point, the compiler looks for a `catch` clause in a higher execution context that can handle an exception of the type that is thrown, or for a `catch` handler that can handle any type of exception. The `catch` handlers are examined in order of their appearance after 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.

4. If a matching handler is still not found, or if an exception occurs during the unwinding process but before the handler gets control, the predefined run-time function `terminate` is called. If an exception occurs after the exception is thrown but before the unwind begins, `terminate` is called.

5. If a matching `catch` handler is found, and it catches by value, its formal parameter is initialized by copying the exception object. If it catches by reference, the parameter is initialized to refer to the exception object. After the formal parameter is initialized, the process of unwinding the stack begins. This involves the destruction of all automatic objects that were fully constructed—but not yet destructed —between the beginning of the `try` block that is associated with the `catch` handler and the throw site of the exception. Destruction occurs in reverse order of construction. The `catch` handler is executed and the program resumes execution after the last handler—that is, at the first statement or construct that is not a `catch` handler. Control can only enter a `catch` handler through a thrown exception, never through a `goto` statement or a `case` label in a `switch`statement.

Stack Unwinding Example

The following example demonstrates how the stack is unwound when an exception is thrown. Execution on the

thread jumps from the throw statement in `C` to the catch statement in `main`, and unwinds each function along the way. Notice the order in which the `Dummy` objects are created and then destroyed as they go out of scope. Also notice that no function completes except `main`, which contains the catch statement. Function `A` never returns from its call to `B()`, and `B` never returns from its call to `C()`. If you uncomment the definition of the `Dummy` pointer and the corresponding delete statement, and then run the program, notice that the pointer is never deleted. This shows what can happen when functions do not provide an exception guarantee. For more information, see How to: Design for Exceptions. If you comment out the catch statement, you can observe what happens when a program terminates because of an unhandled exception.

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

class MyException{};
class Dummy
{
    public:
    Dummy(string s) : MyName(s) { PrintMsg("Created Dummy:"); }
    Dummy(const Dummy& other) : MyName(other.MyName){ PrintMsg("Copy created
Dummy:"); }
    ~Dummy(){ PrintMsg("Destroyed Dummy:"); }
    void PrintMsg(string s) { cout << s  << MyName <<  endl; }
    string MyName;
    int level;
};

void C(Dummy d, int i)
{
    cout << "Entering FunctionC" << endl;
    d.MyName = " C";
    throw MyException();

    cout << "Exiting FunctionC" << endl;
}

void B(Dummy d, int i)
{
    cout << "Entering FunctionB" << endl;
    d.MyName = "B";
    C(d, i + 1);
    cout << "Exiting FunctionB" << endl;
}

void A(Dummy d, int i)
{
    cout << "Entering FunctionA" << endl;
    d.MyName = " A" ;
  //  Dummy* pd = new Dummy("new Dummy"); //Not exception safe!!!
    B(d, i + 1);
 //   delete pd;
    cout << "Exiting FunctionA" << endl;
```

```
}

int main()
{
    cout << "Entering main" << endl;
    try
    {
        Dummy d(" M");
        A(d,1);
    }
    catch (MyException& e)
    {
        cout << "Caught an exception of type: " << typeid(e).name() << endl;
    }

    cout << "Exiting main." << endl;
    char c;
    cin >> c;
}

/* Output:
    Entering main
    Created Dummy: M
    Copy created Dummy: M
    Entering FunctionA
    Copy created Dummy: A
    Entering FunctionB
    Copy created Dummy: B
    Entering FunctionC
    Destroyed Dummy: C
    Destroyed Dummy: B
    Destroyed Dummy: A
    Destroyed Dummy: M
    Caught an exception of type: class MyException
    Exiting main.

*/
```