

Stack unwinding (C++ only)

Last Updated: 2021-03-22

When an exception is thrown and control passes from a try block to a handler, the C++ run time calls destructors for all automatic objects constructed since the beginning of the try block. This process is called *stack unwinding*. The automatic objects are destroyed in reverse order of their construction. (Automatic objects are local objects that have been declared `auto` or `register`, or not declared `static` or `extern`. An automatic object `x` is deleted whenever the program exits the block in which `x` is declared.)

If an exception is thrown during construction of an object consisting of subobjects or array elements, destructors are only called for those subobjects or array elements successfully constructed before the exception was thrown. A destructor for a local static object will only be called if the object was successfully constructed.

If during stack unwinding a destructor throws an exception and that exception is not handled, the `terminate()` function is called. The following example demonstrates this:

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

struct E {
    const char* message;
    E(const char* arg) : message(arg) { }
};

void my_terminate() {
    cout << "Call to my_terminate" << endl;
};

struct A {
    A() { cout << "In constructor of A" << endl; }
    ~A() {
        cout << "In destructor of A" << endl;
        throw E("Exception thrown in ~A()");
    }
};

struct B {
    B() { cout << "In constructor of B" << endl; }
    ~B() { cout << "In destructor of B" << endl; }
};

int main() {
    set_terminate(my_terminate);

    try {
        cout << "In try block" << endl;
        A a;
        B b;
        throw("Exception thrown in try block of main()");
    }
```



```

}
catch (const char* e) {
    cout << "Exception: " << e << endl;
}
catch (...) {
    cout << "Some exception caught in main()" << endl;
}

cout << "Resume execution of main()" << endl;
}

```

The output of this example:

```

In try block
In constructor of A
In constructor of B
In destructor of B
In destructor of A
Call to my_terminate

```

In the try block, two automatic objects are created: a and b. The try block throws an exception of type `const char*`. The handler `catch (const char* e)` catches this exception. The C++ run time unwinds the stack, calling the destructors for a and b in reverse order of their construction. The destructor for a throws an exception. Since there is no handler in the program that can handle this exception, the C++ run time calls `terminate()`. (The function `terminate()` calls the function specified as the argument to `set_terminate()`. In this example, `terminate()` has been specified to call `my_terminate()`.)

► C++11

When the delegating constructors feature is enabled, if an exception is thrown in the body of a delegating constructor, the destructors of the objects constructed through target constructor will be invoked automatically. The destructors must be called in such a way that it calls the destructors of subobjects as appropriate. In particular, it should call the destructors for virtual base classes if the virtual base classes are created through the target constructor.

If an exception is thrown in the body of a delegating constructor, the destructor is invoked for the object created by the target constructor. If an exception escapes from a non-delegating constructor, the unwinding mechanism will call the destructors for the completely constructed subobjects. The following example demonstrates this:

```

class D{
    D():D('a') { printf("D:D().\n");}

    D:D(char) try: D(55){
        printf("D:D(char). Throws.\n");
        throw 0;
    }
}

```

```
catch(...) {  
    printf("D::D(char).Catch block.\n");  
}  
  
D::D(int i):i(i_) {printf("D::D(int).\n");}  
  
D::~D() {printf("D::~D().\n");}  
}  
  
int main(void){  
    D d;  
}
```

The output of the example is:

```
D::D(int).  
D::D(char).Throws.  
D::~D().  
D::D(char).Catch block.
```

In this example, an exception occurs in the delegating constructor `D::D(char)`, so destructor `D::~D()` is invoked for object `d`.

For more information, see [Delegating constructors \(C++11\)](#)

C++11

Parent topic:

→ [Exception handling \(C++ only\)](#)