OOP

Gavrilut Dragos

Course 11

rev 1

Summary

- Basic Exceptions
- Standard exceptions
- noexcept keyword
- Modeling Exception Behavior
- SEH (Structured Exception Handling)
- RAII

Basic Exceptions

Exceptions

- Exceptions are a way to identify errors that can occur in a program and react to them.

```
try
{
    // cod to be executed
}
catch (<exception-1>) { … }
catch (<exception-2>) { … }
catch (<exception-3>) { … }
…
catch (…) { /* the rest of the possible exceptions */ }
```

Where **exception-1,2,…3** could be anything: a type, a reference, a pointer

Exceptions

- Additionally, the keyword **throw** can be used to throw an exception that will further be received by the **try … catch** block

```
try
{
    // cod to be executed
    throw <obj>
}
catch (<exception> /* exception has the same type as obj */ ) { … }
// optional
catch (…) { /* the rest of the possible exceptions */ }
```

When using this method, the object <obj> will be send to the catch block (as such the <exception> parameter in the catch block should have the same type or a type where there is a possible conversion from the thrown type exists.

Exceptions

- Let’s see an example:

App.cpp

```
#include <stdio.h>
#include <iostream>


int main() {
    try {
        for (int tr = 0; tr < 10; tr++) {
            if (tr == 5) throw "Reached 5";
        }
    }
    catch (const char* error) {
        std::cout << "Error: " << error;
    }


    return 0;
}
```

Output:

```
Error: Reached 5
```

Exceptions

- A more complex example:

App.cpp

```
#include <stdio.h>
#include <iostream>


int main() {
    for (int tr = 0; tr < 10; tr++) {
        try {
            if (tr == 5) throw "Reached 5";
            if (tr == 3) throw 3;
            if (tr == 1) throw true;
            std::cout << "Number : " << tr << std::endl;
        }
        catch (const char* e) { std::cout << "String Err: " << e << std::endl; }
        catch (int e) { std::cout << "Int Err: " << e << std::endl; }
        catch (bool e) { std::cout << "Bool Err: " << e << std::endl; }
    }
    return 0;
}
```

```
Output:
Number : 0
Bool Err: 1
Number : 2
Int Err: 3
Number : 4
String Err: Reached 5
Number : 6
Number : 7
Number : 8
Number : 9
```

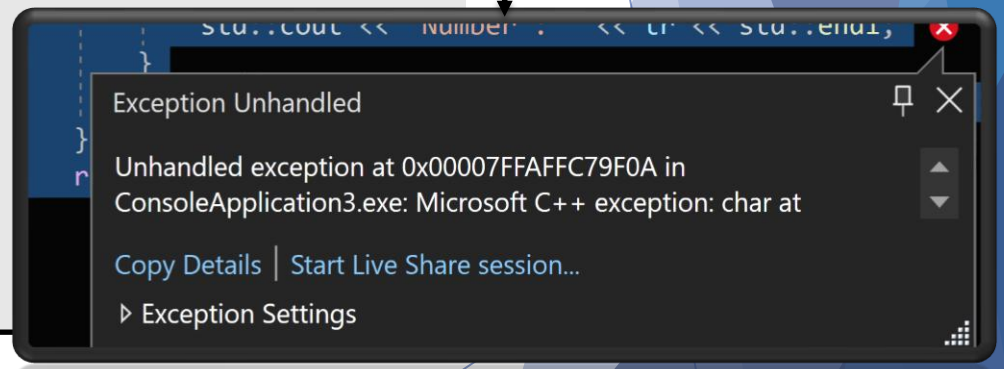
Exceptions

- However, the rules of promotion don't apply:

App.cpp

```
#include <stdio.h>
#include <iostream>


int main() {
    for (int tr = 0; tr < 10; tr++) {
        try {
            if (tr == 3) throw 'A';
            std::cout << "Number : " << tr << std::endl;
        }
        catch (int e) { std::cout << "Int Err: " << e << std::endl; }
    }
    return 0;
}
```

```
Output:
Number : 0
Number : 1
Number : 2
```



- The **throw** instruction sends a ***char***, but the **catch** requires an int. As such the exception is not caught and a ***runtime crash will happen.***

Exceptions

- However, you can always use catch(...) to capture any kind of exception. Details related to the exception will not be present.

App.cpp

```
#include <stdio.h>
#include <iostream>


int main() {
    for (int tr = 0; tr < 10; tr++) {
        try {
            if (tr == 1) throw true;
            if (tr == 3) throw 'A';
            if (tr == 5) throw "some error";
            if (tr == 7) throw 10;
            std::cout << "Number : " << tr << std::endl;
        }
        catch (int e) { std::cout << "Int Err: " << e << std::endl; }
        catch (...) { std::cout << "Generic exceptions" << std::endl; }
    }
    return 0;
}
```

Output:

```
Number : 0
Generic exceptions
Number : 2
Generic exceptions
Number : 4
Generic exceptions
Number : 6
Int Err: 10
Number : 8
Number : 9
```

Exceptions

- However, you can always use catch(...) to capture any kind of exception. Details related to the exception will not be present.

App.cpp

```
#include <stdio.h>
#include <iostream>


int main() {
    for (int tr = 0; tr < 10; tr++) {
        try {
            if (tr == 1) throw true;
            if (tr == 3) throw 'A';
            if (tr == 5) throw "some error";
            if (tr == 7) throw 10;
            std::cout << "Number : " << tr << std::endl;
        }
        catch (int e) { std::cout << "Int Err: " << e << std::endl; }
        catch (...) { std::cout << "Generic exceptions" << std::endl; }
    }
    return 0;
}
```

```
Output:
Number : 0
Generic exceptions
Number : 2
Generic exceptions
Number : 4
Generic exceptions
Number : 6
Int Err: 10
Number : 8
Number : 9
```

Exceptions

- However, you can always use catch(...) to capture any kind of exception. Details related to the exception will not be present.

App.cpp

```
#include <stdio.h>
#include <iostream>


int main() {
    for (int tr = 0; tr < 10; tr++) {
        try {
            if (tr == 1) throw true;
            if (tr == 3) throw 'A';
            if (tr == 5) throw "some error";
            if (tr == 7) throw 10;
            std::cout << "Number : " << tr << std::endl;
        }
        catch (int e) { std::cout << "Int Err: " << e << std::endl; }
        catch (...) { std::cout << "Generic exceptions" << std::endl; }
    }
    return 0;
}
```

Output:

```
Number : 0
Generic exceptions
Number : 2
Generic exceptions
Number : 4
Generic exceptions
Number : 6
Int Err: 10
Number : 8
Number : 9
```

Exceptions

- You have to use catch(…) as the last option. Adding it before other catch branches will result in a compile error.

App.cpp

```
#include <stdio.h>
#include <iostream>


int main() {
    for (int tr = 0; tr < 10; tr++) {
        try {
            if (tr == 1) throw true;
            if (tr == 3) throw 'A';
            if (tr == 5) throw "some error";
            if (tr == 7) throw 10;
            std::cout << "Number : " << tr << std::endl;
        }
        catch (...) { std::cout << "Generic exceptions" << std::endl; }
        catch (int e) { std::cout << "Int Err: " << e << std::endl; }
    }
    return 0;
}
```

error **C2311**: 'int': is caught by '...' on line …

Exceptions

- You can also use a try…catch block within another try…catch block

App.cpp

```
#include <stdio.h>
#include <iostream>


int main() {
    try {
        for (int tr = 0; tr < 10; tr++) {
            try {
                if (tr == 1) throw true;
                if (tr == 7) throw 7;
                std::cout << "Number : " << tr << std::endl;
            }
            catch (bool e) { std::cout << "Bool Err: " << e << std::endl; }
        }
    }
    catch (int e) { std::cout << "Int Err: " << e << std::endl; }
    return 0;
}
```

```
Output:
Number : 0
Bool Err: 1
Number : 2
Number : 3
Number : 4
Number : 5
Number : 6
Int Err: 7
```

Notice that the if second outer try…catch loop receives the exception the for loop is break (*we will not reach the value tr = 8*)

Exceptions

- You can also use throw without any parameter to send current exception to the outer try…catch (if exists).

App.cpp

```
#include <stdio.h>
#include <iostream>


int main() {
    try {
        for (int tr = 0; tr < 10; tr++) {
            try {
                if (tr == 5) throw 5;
                std::cout << "Number : " << tr << std::endl;
            }
            catch (int e) {
                std::cout << "Int Err: " << e << std::endl;
                throw;
            }
        }
    }
    catch (...) { std::cout << "Generic Err: " << std::endl; }
    return 0;
}
```

```
Output:
Number : 0
Number : 1
Number : 2
Number : 3
Number : 4
Int Err: 5
Generic Err:
```

Exceptions

- In this case, .at(…) method from a vector throws an exception if the index is invalid:

App.cpp

```
#include <stdio.h>
#include <iostream>
#include <vector>

int main() {
    std::vector<int> a = { 1,2,3 };
    try {
        int x = a.at(100);
    }
    catch (...)
    {
        std::cout << "Invalid index" << std::endl;
    }
    return 0;
}
```

Output:

```
Invalid index
```

Exceptions

- An exception (if thrown) will exit any function from where it is triggered. Noticed that “end foo()” message is not printed as the execution never reaches that point.

App.cpp

```
#include <stdio.h>
#include <iostream>

void foo() {
    std::cout << "start foo()" << std::endl;
    throw 100;
    std::cout << "end foo()" << std::endl;
}
int main() {
    try {
        foo();
    }
    catch (...)
    {
        std::cout << "Exception" << std::endl;
    }
    return 0;
}
```

Output:

```
start foo()
Exception
```

Standard Exceptions

Standard Exceptions

- The standard library also provides an exception class: std::exception
- The purpose of this class is to provide a basic class from where a hierarchy of exceptions can be built.
- The class has a virtual method **what()** that can be overwritten to provide a description of the exception:

```
class exception
{
   virtual const char* what() const throw();
}
```

Standard Exceptions

- The hierarchy of exceptions derived from **std::exceptions** are:

exception	scope
bad_alloc	When we fail to allocate memory
bad_cast	dynamic_cast exception
logic_error	Generic exception for a logic error
runtime_error	An exception describing a runtime error
…	

- **logic_error** is also the base class for the following errors: **length_error, invalid_argument, out_of_range, etc**
- **runtime_error** is also the base class for the following errors: **overflow_error, range_error, system_error, underflow_error, etc.**

Standard Exceptions

- Let’s see an example:

App.cpp

```
#include <stdio.h>
#include <iostream>


int main() {
    try {
        for (int tr = 0; tr < 10; tr++) {
            if (tr == 5) throw std::exception("value is 5");
            std::cout << tr << std::endl;
        }
    }
    catch (std::exception e) {
        std::cout << "Exception: " << e.what();
    }
    return 0;
}
```

```
Output:
0
1
2
3
4
Exception: value is 5
```

Standard Exceptions

- Let’s see an example:

App.cpp

```
#include <stdio.h>
#include <iostream>


int main() {
    try {
        for (int tr = 0; tr < 10; tr++) {
            if (tr == 5) throw std::exception("value is 5");
            std::cout << tr << std::endl;
        }
    }
    catch (std::exception e) {
        std::cout << "Exception: " << e.what();
    }
    return 0;
}
```

```
Output:
0
1
2
3
4
Exception: value is 5
```

Standard Exceptions

- Usually, when using std::exceptions it is best to use a constant reference (so that we don't need to copy the exception object if it gets passed to the outer try … catch block).

App.cpp

```
#include <stdio.h>
#include <iostream>
#include <vector>


int main() {
    std::vector<int> v = { 1,2,3 };
    try {
        int c = v.at(100);
    }
    catch (const std::out_of_range &e) {
        std::cout << "Exception: " << e.what();
    }
    return 0;
}
```

Output:

```
Exception: invalid vector subscript
```

Standard Exceptions

- You can also use **throw** to forward current exception to an outer try…catch block

App.cpp

```
#include <stdio.h>
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v = { 1,2,3 };
    try {
        try {
            int c = v.at(100);
        }
        catch (const std::out_of_range& e) {
            std::cout << "Exception: " << e.what() << std::endl;
            throw;
        }
    }
    catch (const std::exception& e) {
        std::cout << "Outer: " << e.what() << std::endl;
    }
    return 0;
}
```

Output:

```
Exception: invalid vector subscript
Outer: invalid vector subscript
```

Standard Exceptions

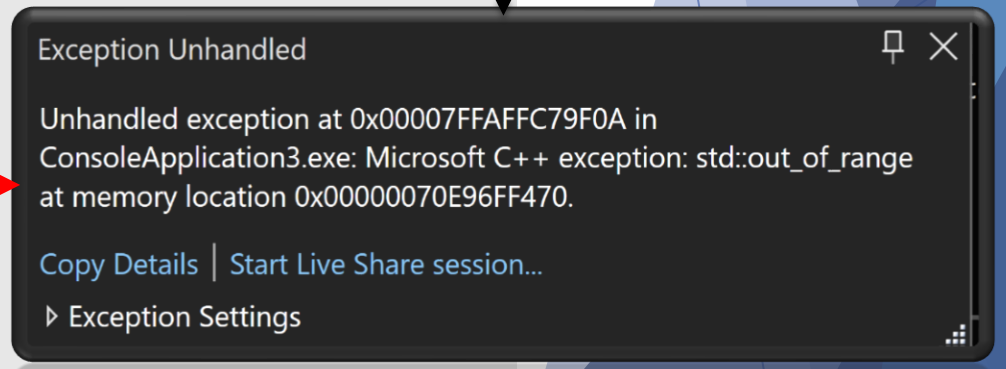
- A std based exception **can not be converted** into a regular exception:

App.cpp

```
#include <stdio.h>
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v = { 1,2,3 };
    try {
        try {
            int c = v.at(100);
        }
        catch (const std::out_of_range& e) {
            std::cout << "Exception: " << e.what() << std::endl;
            throw;
        }
    }
    catch (const char* text) {
        std::cout << "Outer: " << text << std::endl;
    }
    return 0;
}
```

Output:

```
Exception: invalid vector subscript
```



Standard Exceptions

- However, they work with a catch(…) block. The following code will not crash during runtime.

App.cpp

```
#include <stdio.h>
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v = { 1,2,3 };
    try {
        try {
            int c = v.at(100);
        }
        catch (const std::out_of_range& e) {
            std::cout << "Exception: " << e.what() << std::endl;
            throw;
        }
    }
    catch (...) {
        std::cout << "Outer Exception" <<  std::endl;
    }
    return 0;
}
```

Output:

```
Exception: invalid vector subscript
Outer Exception
```

Standard Exceptions

- We can create a custom exception by overwriting the **what()** method, like in the following example:

App.cpp

```
#include <stdio.h>
#include <iostream>
#include <vector>

struct DivisionBy0 : public std::exception
{
    int value;
    virtual const char* what() const override
    {
        return "Division by zero ";
    }
    DivisionBy0(int v): value(v) {}
};
```

- That can be used in the following way:

Standard Exceptions

- We can create a custom exception by overwriting the **what()** method, like in the following example:

App.cpp

```
#include <stdio.h>
#include <iostream>
#include <vector>

struct DivisionBy0 : public std::exception { … };

int main() {
    int x = 10;
    for (int i = -2; i < 2; i++) {
        try {
            if (i == 0) throw DivisionBy0(x);
            std::cout << x << "/" << i << "=" << x / i << std::endl;
        }
         catch (const DivisionBy0& e) {
             std::cout << "Exception: " << e.what() << " for number: " << e.value << std::endl;
         }
    }
    return 0;
}
```

Output:

```
10/-2=-5
10/-1=-10
Exception: Division by zero  for number: 10
10/1=10
```

Standard Exceptions

- You can however use **throw** and capture a custom exception using its base class (in our case std::exception).

App.cpp

```
#include <stdio.h>
#include <iostream>
#include <vector>

struct DivisionBy0 : public std::exception { … };

int main() {
    int x = 10;
    try {
        for (int i = -2; i < 2; i++) {
            try {
                if (i == 0) throw DivisionBy0(x);
                std::cout << x << "/" << i << "=" << x / i << std::endl;
            }
            catch (const DivisionBy0& e) {
                std::cout << "Exception: " << e.what() << " for number: " << e.value << std::endl;
                throw;
            }
        }
    }
    catch (const std::exception& e) {
        std::cout << "Outer Exception: " << e.what() << std::endl;
    }
    return 0;
}
```

```
Output:
10/-2=-5
10/-1=-10
Exception: Division by zero  for number: 10
Outer Exception: Division by zero
```

Standard Exceptions

- You can also **throw** another exception directly in the **catch** block like in the following example:

App.cpp

```
#include <stdio.h>
#include <iostream>
#include <vector>

struct DivisionBy0 : public std::exception { … };

int main() {
    int x = 10;
    try {
        for (int i = -2; i < 2; i++) {
            try {
                if (i == 0) throw DivisionBy0(x);
                std::cout << x << "/" << i << "=" << x / i << std::endl;
            }
            catch (const DivisionBy0& e) {
                std::cout << "Exception: " << e.what() << " for number: " << e.value << std::endl;
                throw std::exception("generic exception");
            }
        }
    }
    catch (const std::exception& e) {
        std::cout << "Outer Exception: " << e.what() << std::endl;
    }
    return 0;
}
```

Output:

```
10/-2=-5
10/-1=-10
Exception: Division by zero  for number: 10
Outer Exception: generic exception
```

- noexcept

noexcept

- noexcept is a specifier used to indicate whether a function **is guaranteed not to throw exceptions**.
- It is mainly used to improve code optimizations
- The format is:

```
type function_name (parameter…) noexcept;
```

- noexcept is used in std container where templates like vector<> may decide to use a copy or a move constructor

- You can also use **noexcept(false)** to specify that a function might raise an exception:

```
type function_name (parameter…) noexcept(false);
```

Standard Exceptions

- Let’s analyze the following case:

App.cpp

```
#include <iostream>
#include <vector>

struct Test {
    Test() { printf("CTOR(this=%p)\n", this); }
    Test(const Test& obj) { printf("COPY-CTOR(this=%p, obj=%p)\n", this, &obj); }
    Test(Test&& obj) { printf("MOVE-CTOR(this=%p, obj=%p)\n", this, &obj); }
};


int main() {
    std::vector<Test> v;
    Test t;
    v.reserve(1);
    std::cout << "First element " << std::endl;
    v.push_back(t);
    std::cout << "Second element " << std::endl;
    v.push_back(t);
    return 0;
}
```

Standard Exceptions

- Let’s analyze the following case:

App.cpp

```
#include <iostream>
#include <vector>

struct Test {
    Test() { printf("CTOR(this=%p)\n", this); }
    Test(const Test& obj) { printf("COPY-CTOR(this=%p,
    Test(Test&& obj) { printf("MOVE-CTOR(this=%p, obj=%
};


int main() {
    std::vector<Test> v;
    Test t;
    v.reserve(1);
    std::cout << "First element " << std::endl;
    v.push_back(t);
    std::cout << "Second element " << std::endl;
    v.push_back(t);
    return 0;
}
```

```
Output:
CTOR(this=0000002CB51EF974)
First element
COPY-CTOR(this=000001A8F5C18300, obj=0000002CB51EF974)
Second element
COPY-CTOR(this=000001A8F5C17DC1, obj=0000002CB51EF974)
COPY-CTOR(this=000001A8F5C17DC0, obj=000001A8F5C18300)
```

Standard Exceptions

- Let’s analyze the following case:

App.cpp

```
#include <iostream>
#include <vector>

struct Test {
    Test() { printf("CTOR(this=%p)\n", this); }
    Test(const Test& obj) { printf("COPY-CTOR(this=%p,
    Test(Test&& obj) { printf("MOVE-CTOR(this=%p, obj=%
};


int main() {
    std::vector<Test> v;
    Test t;
    v.reserve(1);
    std::cout << "First element " << std::endl;
    v.push_back(t);
    std::cout << "Second element " << std::endl;
    v.push_back(t);
    return 0;
}
```

Output:

```
CTOR(this=t 0000002CB51EF974)
First element
COPY-CTOR(this=000001A8F5C18300, obj=t 0000002CB51EF974)
Second element
COPY-CTOR(this=000001A8F5C17DC1, obj=t 0000002CB51EF974)
COPY-CTOR(this=000001A8F5C17DC0, obj=000001A8F5C18300)
```

Variable “t” is located at the following address: **0000002CB51EF974**

Standard Exceptions

- Let's analyze the following case:

App.cpp

```
#include <iostream>
#include <vector>

struct Test {
    Test() { printf("CTOR(this=%p)\n", this); }
    Test(const Test& obj) { printf("COPY-CTOR(this=%p,
    Test(Test&& obj) { printf("MOVE-CTOR(this=%p, obj=%
};


int main() {
    std::vector<Test> v;
    Test t;
    v.reserve(1);
    std::cout << "First element " << std::endl;
    v.push_back(t);
    std::cout << "Second element " << std::endl;
    v.push_back(t);
    return 0;
}
```

Output:

```
CTOR(this=t 0000002CB51EF974)
First element
COPY-CTOR(this=000001A8F5C18300, obj=t 0000002CB51EF974)
Second element
COPY-CTOR(this=000001A8F5C17DC1, obj=t 0000002CB51EF974)
COPY-CTOR(this=000001A8F5C17DC0, obj=000001A8F5C18300)
```

We will reserve 1 element, but add 2 elements (so that we force a resize call from the vector)

Standard Exceptions

- Let's analyze the following case:

App.cpp

```
#include <iostream>
#include <vector>

struct Test {
    Test() { printf("CTOR(this=%p)\n", this); }
    Test(const Test& obj) { printf("COPY-CTOR(this=%p,
    Test(Test&& obj) { printf("MOVE-CTOR(this=%p, obj=%
};


int main() {
    std::vector<Test> v;
    Test t;
    v.reserve(1);
    std::cout << "First element " << std::endl;
    v.push_back(t);
    std::cout << "Second element " << std::endl;
    v.push_back(t);
    return 0;
}
```

```
Output:
CTOR(this=t 0000002CB51EF974)
First element
COPY-CTOR(this=000001A8F5C18300, obj=t 0000002CB51EF974)
Second element
COPY-CTOR(this=000001A8F5C17DC1, obj=t 0000002CB51EF974)
COPY-CTOR(this=000001A8F5C17DC0, obj=000001A8F5C18300)
```

Standard Exceptions

- Let's analyze the following case:

App.cpp

```
#include <iostream>
#include <vector>

struct Test {
    Test() { printf("CTOR(this=%p)\n", this); }
    Test(const Test& obj) { printf("COPY-CTOR(this=%p,
    Test(Test&& obj) { printf("MOVE-CTOR(this=%p, obj=%
};


int main() {
    std::vector<Test> v;
    Test t;
    v.reserve(1);
    std::cout << "First element " << std::endl;
    v.push_back(t);
    std::cout << "Second element " << std::endl;
    v.push_back(t);
    return 0;
}
```

Output:

```
CTOR(this=t 0000002CB51EF974)
First element
COPY-CTOR(this=v[0] 000001A8F5C18300, obj=t 0000002CB51EF974)
Second element
COPY-CTOR(this=000001A8F5C17DC1, obj=t 0000002CB51EF974)
COPY-CTOR(this=000001A8F5C17DC0, obj=v[0] 000001A8F5C18300)
```

We copy "t" into the vector (meaning that **000001A8F5C18300** is the address of the first element in the vector: v[0])

Standard Exceptions

- Let's analyze the following case:

App.cpp

```
#include <iostream>
#include <vector>

struct Test {
    Test() { printf("CTOR(this=%p)\n", this); }
    Test(const Test& obj) { printf("COPY-CTOR(this=%p,
    Test(Test&& obj) { printf("MOVE-CTOR(this=%p, obj=%
};


int main() {
    std::vector<Test> v;
    Test t;
    v.reserve(1);
    std::cout << "First element " << std::endl;
    v.push_back(t);
    std::cout << "Second element " << std::endl;
    v.push_back(t);
    return 0;
}
```

```
Output:
CTOR(this=t 0000002CB51EF974)
First element
COPY-CTOR(this=v[0] 000001A8F5C18300, obj=t 0000002CB51EF974)
Second element
COPY-CTOR(this=000001A8F5C17DC1, obj=t 0000002CB51EF974)
COPY-CTOR(this=000001A8F5C17DC0, obj=v[0] 000001A8F5C18300)
```

Standard Exceptions

- Let's analyze the following case:

App.cpp

```
#include <iostream>
#include <vector>

struct Test {
    Test() { printf("CTOR(this=%p)\n", this); }
    Test(const Test& obj) { printf("COPY-CTOR(this=%p,
    Test(Test&& obj) { printf("MOVE-CTOR(this=%p, obj=%
};


int main() {
    std::vector<Test> v;
    Test t;
    v.reserve(1);
    std::cout << "First element " << std::endl;
    v.push_back(t);
    std::cout << "Second element " << std::endl;
    v.push_back(t);
    return 0;
}
```

Output:

```
CTOR(this=t 0000002CB51EF974)
First element
COPY-CTOR(this=v[0] 000001A8F5C18300, obj=t 0000002CB51EF974)
Second element
COPY-CTOR(this=new_v[1] 000001A8F5C17DC1, obj=t 0000002CB51EF974)
COPY-CTOR(this=000001A8F5C17DC0, obj=v[0] 000001A8F5C18300)
```

First we will resize the vector (to fit two elements) and we will copy "t" into the second element of the new address of the vector (meaning that **000001A8F5C17DC1** is the address of the second element in the new vector : new_v[1])

Standard Exceptions

- Let's analyze the following case:

App.cpp

```
#include <iostream>
#include <vector>

struct Test {
    Test() { printf("CTOR(this=%p)\n", this); }
    Test(const Test& obj) { printf("COPY-CTOR(this=%p,
    Test(Test&& obj) { printf("MOVE-CTOR(this=%p, obj=%
};


int main() {
    std::vector<Test> v;
    Test t;
    v.reserve(1);
    std::cout << "First element " << std::endl;
    v.push_back(t);
    std::cout << "Second element " << std::endl;
    v.push_back(t);
    return 0;
}
```

```
Output:
CTOR(this=t 0000002CB51EF974)
First element
COPY-CTOR(this=v[0] 000001A8F5C18300, obj=t 0000002CB51EF974)
Second element
COPY-CTOR(this=new_v[1] 000001A8F5C17DC1, obj=t 0000002CB51EF974)
COPY-CTOR(this=new_v[0] 000001A8F5C17DC0, obj=v[0] 000001A8F5C18300)
```

The next step is to copy the original v[0] into the new_v[0] element follow by the deletion of the original allocated vector.

Standard Exceptions

- Now let's see what happens if we use noexcept for the move-ctor:

App.cpp

```
#include <iostream>
#include <vector>

struct Test {
    Test() { printf("CTOR(this=%p)\n", this); }
    Test(const Test& obj) { printf("COPY-CTOR(this=%p,
    Test(Test&& obj) noexcept { printf("MOVE-CTOR(th
};


int main() {
    std::vector<Test> v;
    Test t;
    v.reserve(1);
    std::cout << "First element " << std::endl;
    v.push_back(t);
    std::cout << "Second element " << std::endl;
    v.push_back(t);
    return 0;
}
```

Output:

```
CTOR(this=t 0000002CB51EF974)
First element
COPY-CTOR(this=v[0] 000001A8F5C18300, obj=t 0000002CB51EF974)
Second element
COPY-CTOR(this=new_v[1] 000001A8F5C17DC1, obj=t 0000002CB51EF974)
MOVE-CTOR(this=new_v[0] 000001A8F5C17DC0, obj=v[0] 000001A8F5C18300)
```

The code functions almost identical, except for the last step where the compiler decides to use MOVE-CTOR instead of COPY-CTOR to transfer the content of the old vector into the new one (much faster !!!).

noexcept

The main reason for this behavior is that the logic behind std containers is that **if an operation fails for some reason, the container remains unchanged.**

So … while move-ctor is preferred, if it might trigger an alert than we might end up with some inconsistencies (e.g. a pointer is being copied and not deleted in the original source leading to a potential double-free).

To avoid this kind of scenarios, move constructor are only being used (in std) if ***noexcept*** is guarantee (meaning that there is no possibility for that code to actually trigger an exception).

noexcept

Noexept can also be used to validate if a function / method is defined using noexcept or not:

```
noexcept(foo())
```

Noexcept also works with **static_assert** allowing various types / templates to add an additional logic / constraints.

Standard Exceptions

- Let’s analyze the following case:

App.cpp

```
#include <iostream>

void f1() noexcept {}
void f2() {}

struct Test {
    void f1() noexcept {}
    void f2() {}
};

int main()
{
    std::cout << std::boolalpha;
    std::cout << "f1 is noexcept: " << noexcept(f1()) << std::endl;
    std::cout << "f2 is noexcept: " << noexcept(f2()) << std::endl;

    Test t;
    std::cout << "Test::f1 is noexcept: " << noexcept(t.f1()) << std::endl;
    std::cout << "Test::f2 is noexcept: " << noexcept(t.f2()) << std::endl;
    return 0;
}
```

Output:

```
f1 is noexcept: true
f2 is noexcept: false
Test::f1 is noexcept: true
Test::f2 is noexcept: false
```

noexcept

Additionally, std also provides several template methods that can be used to check if a type has **noexcept** attribute or not on its special methods (e.g., constructors).

Methods	Purpose
std::is_nothrow_constructible<T, Args...>	True if T can be constructed from Args... without throwing an exception
std::is_nothrow_default_constructible<T>	True if T has a default constructor defined with noexcept
std::is_nothrow_copy_constructible<T>	True if T has a copy constructor defined with noexcept
std::is_nothrow_move_constructible<T>	True if T has a move constructor defined with noexcept
std::is_nothrow_assignable<T, U>	True if T has an assign operator that receives a value of type U that is defined with noexcept
std::is_nothrow_copy_assignable<T>	Similar, but for copy assignment
std::is_nothrow_move_assignable<T>	Similar, but for move assignament

Standard Exceptions

- Let’s analyze the following case:

App.cpp

```
#include <type_traits>
#include <iostream>

struct Test {
    Test() noexcept {}
    Test(const Test&) noexcept {}
    Test(Test&&) noexcept {}
    void operator=(const Test&) noexcept {}
    void operator=(Test&&) noexcept {}
};

int main() {
    std::cout << std::boolalpha;
    std::cout << "nothrow default-ctor: " << std::is_nothrow_default_constructible<Test>::value << std::endl;
    std::cout << "nothrow copy-ctor: " << std::is_nothrow_copy_constructible<Test>::value << std::endl;
    std::cout << "nothrow move-ctor: " << std::is_nothrow_move_constructible<Test>::value << std::endl;
    std::cout << "nothrow copy-asign: " << std::is_nothrow_copy_assignable<Test>::value << std::endl;
    std::cout << "nothrow move-asign: " << std::is_nothrow_move_assignable<Test>::value << std::endl;
}
```

Output:

```
nothrow default-ctor: true
nothrow copy-ctor: true
nothrow move-ctor: true
nothrow copy-asign: true
nothrow move-asign: true
```

Standard Exceptions

- If we remove **noexcept** from the definitions we will however get a different output.

App.cpp

```
#include <type_traits>
#include <iostream>

struct Test {
    Test() {}
    Test(const Test&) {}
    Test(Test&&) {}
    void operator=(const Test&) {}
    void operator=(Test&&) {}
};

int main() {
    std::cout << std::boolalpha;
    std::cout << "nothrow default-ctor: " << std::is_nothrow_default_constructible<Test>::value << std::endl;
    std::cout << "nothrow copy-ctor: " << std::is_nothrow_copy_constructible<Test>::value << std::endl;
    std::cout << "nothrow move-ctor: " << std::is_nothrow_move_constructible<Test>::value << std::endl;
    std::cout << "nothrow copy-asign: " << std::is_nothrow_copy_assignable<Test>::value << std::endl;
    std::cout << "nothrow move-asign: " << std::is_nothrow_move_assignable<Test>::value << std::endl;
}
```

Output:

```
nothrow default-ctor: false
nothrow copy-ctor: false
nothrow move-ctor: false
nothrow copy-asign: false
nothrow move-asign: false
```

Standard Exceptions

- These methods can also be used with a **static_assert** to force a noexcept over a templatized type.

App.cpp

```
#include <type_traits>
#include <iostream>

struct Test {
    Test(const Test&)  {}
};

template <typename T>
struct MyTemplate {
    MyTemplate() {
        static_assert(std::is_nothrow_copy_constructible<T>::value, "use noexcept !");
    }
};

int main() {
    MyTemplate<Test> t;
    return 0;
}
```

This code will not compile with the following error message:

```
error C2338: static_assert failed: 'use noexcept !'
```

Standard Exceptions

- These methods can also be used with a **static_assert** to force a noexcept over a templatized type.

App.cpp

```
#include <type_traits>
#include <iostream>

struct Test {
    Test(const Test&)  {}
};

template <typename T>
struct MyTemplate {
    MyTemplate() {
        static_assert(std::is_nothrow_copy_constructible<T>::value, "use noexcept !");
    }
};

int main() {
    MyTemplate<Test> t;
    return 0;
}
```

Standard Exceptions

- You can also use noexcept(function()) to select a logic (e.g. similar to std::vector)

App.cpp

```
#include <type_traits>
#include <iostream>

struct Test {
    Test(const Test&)  {}
};

template <typename T>
struct MyVector {
    void grow() {
        if (std::is_nothrow_copy_constructible<T>::value) {
            // use the copy ctor
        }
        else {
            // use the move ctor
        }
    }
};
int main() {
    MyVector<Test> v;
    std::cout << "Execution ok !";
    return 0;
}
```

- std::nothrow

std::nothrow

- Whenever a large amount of data it is being allocated a **bad_alloc** exception is being thrown:

App.cpp

```
#include <iostream>


int main() {
    try {
        int* x = new int[1000000000000];
    }
    catch (std::bad_alloc e) {
        std::cout << "Alloc Error: " << e.what();
    }
    return 0;
}
```

Output:

```
Alloc Error: bad allocation
```

std::nothrow

- However, sometimes you want a new operator that (in case of an exception) returns a **nullptr** instead of throwing an exception
- The solution is to use a **std::nothrow** parameter with the new operator.

App.cpp

```
#include <iostream>

int main() {

    int* x = new(std::nothrow) int[1000000000000];
    if (x == nullptr) {
        std::cout << "Unable to allocate memory !";
    }
    return 0;
}
```

Output:

```
Unable to allocate memory !
```

Modeling exception behavior

Modeling Exception Behavior

- Let's analyze the following code:

App.cpp

```
#include <iostream>
#include <Windows.h>

int main() {
    printf("Start\n");
    try
    {
        int x = GetTickCount() % 10;
        if (x == 0) throw 10;
        if (x == 1) throw true;
        if (x == 2) throw 1.2;
        if (x == 3) throw "Exception text";
        printf("x is not 0,1,2 or 3\n");
    }
    catch (int e) { printf("Exception: %d\n", e); }
    catch (bool b) { printf("Bool exception \n"); }
    catch (double d) { printf("Double exception: %lf \n",d); }
    catch (...) { printf("Generic exception !"); }
    printf("End\n");
    return 0;
}
```

Output (possible):

```
x is not 0,1,2 or 3
```

Modeling Exception Behavior

- Let's analyze the following code:

App.cpp

```
#include <iostream>
#include <Windows.h>

int main() {
    printf("Start\n");
    try
    {
        int x = GetTickCount() % 10;
        if (x == 0) throw 10;
        if (x == 1) throw true;
        if (x == 2) throw 1.2;
        if (x == 3) throw "Exception text";
        printf("x is not 0,1,2 or 3\n");
    }
    catch (int e) { printf("Exception: %d\n", e); }
    catch (bool b) { printf("Bool exception \n"); }
    catch (double d) { printf("Double exception: %lf \n",d); }
    catch (...) { printf("Generic exception !"); }
    printf("End\n");
    return 0;
}
```

ASM Code

```
call    qword ptr [GetTickCount (07FF7414E2000h)]
xor     edx,edx
mov     ecx,10
div     eax,ecx
mov     eax,edx
mov     dword ptr [rbp+4],eax
```

Modeling Exception Behavior

- Let's analyze the following code:

App.cpp

```
#include <iostream>
#include <Windows.h>

int main() {
    printf("Start\n");
    try
    {
        int x = GetTickCount() % 10;
        if (x == 0) throw 10;
        if (x == 1) throw true;
        if (x == 2) throw 1.2;
        if (x == 3) throw "Exception text";
        printf("x is not 0,1,2 or 3\n");
    }
    catch (int e) { printf("Exception: %d\n", e); }
    catch (bool b) { printf("Bool exception \n"); }
    catch (double d) { printf("Double exception: %lf \n",d); }
    catch (...) { printf("Generic exception !"); }
    printf("End\n");
    return 0;
}
```

ASM Code

```
cmp     dword ptr [rbp+4],0
jne     NEXT_IF
mov     dword ptr [rbp+144h],10
lea     rdx,[_TI1H (07FF7414DD1B0h)]
lea     rcx,[rbp+144h]
call    _CxxThrowException
```

Modeling Exception Behavior

- Let's analyze the following code:

App.cpp

```
#include <iostream>
#include <Windows.h>

int main() {
    printf("Start\n");
    try
    {
        int x = GetTickCount() % 10;
        if (x == 0) throw 10;
        if (x == 1) throw true;
        if (x == 2) throw 1.2;
        if (x == 3) throw "Exception text";
        printf("x is not 0,1,2 or 3\n");
    }
    catch (int e) { printf("Exception: %d\n", e); }
    catch (bool b) { printf("Bool exception \n"); }
    catch (double d) { printf("Double exception: %lf \n",d); }
    catch (...) { printf("Generic exception !"); }
    printf("End\n");
    return 0;
}
```

ASM Code

```
cmp     dword ptr [rbp+4],1
jne     NEXT_IF
mov     dword ptr [rbp+164h],1
lea     rdx,[_TI1_N (07FF7414DD208h)]
lea     rcx,[rbp+164h]
call    _CxxThrowException
```

Modeling Exception Behavior

- Let's analyze the following code:

App.cpp

```
#include <iostream>
#include <Windows.h>

int main() {
    printf("Start\n");
    try
    {
        int x = GetTickCount() % 10;
        if (x == 0) throw 10;
        if (x == 1) throw true;
        if (x == 2) throw 1.2;
        if (x == 3) throw "Exception text";
        printf("x is not 0,1,2 or 3\n");
    }
    catch (int e) { printf("Exception: %d\n", e); }
    catch (bool b) { printf("Bool exception \n"); }
    catch (double d) { printf("Double exception: %lf \n",d); }
    catch (...) { printf("Generic exception !"); }
    printf("End\n");
    return 0;
}
```

ASM Code

```
cmp     dword ptr [rbp+4],2
jne     main+0B9h (07FF7414D1A09h)
movsd   xmm0,mmword ptr [__real@3ff3333333333333
movsd   mmword ptr [rbp+188h],xmm0
lea     rdx,[_TI1N (07FF7414DD260h)]
lea     rcx,[rbp+188h]
call    _CxxThrowException
```

Modeling Exception Behavior

- Let's analyze the following code:

App.cpp

```
#include <iostream>
#include <Windows.h>

int main() {
    printf("Start\n");
    try
    {
        int x = GetTickCount() % 10;
        if (x == 0) throw 10;
        if (x == 1) throw true;
        if (x == 2) throw 1.2;
        if (x == 3) throw "Exception text";
        printf("x is not 0,1,2 or 3\n");
    }
    printf("End\n");
    return 0;
}
```

https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/cxxthrowexception?view=msvc-170

```
extern "C" void __stdcall _CxxThrowException( void* pExceptionObject,
                                              _ThrowInfo* pThrowInfo );
```

ASM Code

```
cmp         dword ptr [rbp+4],2
jne         main+0B9h (07FF7414D1A09h)
movsd       xmm0,mmword ptr [__real@3ff3333333333333
movsd       mmword ptr [rbp+188h],xmm0
lea         rdx,[_TI1N (07FF7414DD260h)]
lea         rcx,[rbp+188h]
call        _CxxThrowException
```

Pseudo-code

```
if (x==2) {
    double value = 1.2;
    _ThrowInfo* t = (_ThrowInfo*) (<some address>);
    _CxxThrowException (t, (void*) &value)
}
```

Modeling Exception Behavior

- The purpose of the _ThrowInfo structure is to:
  - Provide type information related to the thrown values (is it int, float, bool, or other type)
  - Provide a list of methods required for unwind
  - Provide a table to addresses where the catch offsets start (in our case the catch for int , the one for bool, the one for double and the generic one).

- The `CxxThrowException` analyzes the value that it receives and the throw info data and jumps to the appropriate address.

Modeling Exception Behavior

- Let's analyze the following code:

App.cpp

```
#include <iostream>
#include <Windows.h>

int main() {
    printf("Start\n");
    try
    {
        int x = GetTickCount() % 10;
        if (x == 0) throw 10;
        if (x == 1) throw true;
        if (x == 2) throw 1.2;
        if (x == 3) throw "Exception text";
        printf("x is not 0,1,2 or 3\n");
    }
    catch (int e) { printf("Exception: %d\n", e); }
    catch (bool b) { printf("Bool exception \n"); }
    catch (double d) { printf("Double exception: %lf \n",d); }
    catch (...) { printf("Generic exception !"); }
    printf("End\n");
    return 0;
}
```

Output (possible):

```
x is not 0,1,2 or 3
```

Modeling Exception Behavior

- This is a simulation on how the exception mechanism is translated into a code. It implies ZERO overhead on the normal execution (so only when **throw** is triggered we will jump to a specific address where the catch exists).

App.cpp

```
#include <iostream>
#include <Windows.h>

int main() {
    printf("Start\n");
    try
    {
        int x = GetTickCount() % 10;
        if (x == 0) throw 10;
        if (x == 1) throw true;
        if (x == 2) throw 1.2;
        if (x == 3) throw "Exception text";
        printf("x is not 0,1,2 or 3\n");
    }
    catch (int e) { … }
    catch (bool b) { … }
    catch (double d) { … }
    catch (...) { … }
    printf("End\n");
    return 0;
}
```

Line	Pseudo code
1	`int main() {`
2	`    printf("Start\n");`
3	`    int x = GetTickCount() % 10;`
4	`    if (x == 0) GOTO LINE 10;`
5	`    if (x == 1) GOTO LINE 12;`
6	`    if (x == 2) GOTO LINE 14;`
7	`    if (x == 3) GOTO LINE 16;`
8	`    printf("x is not 0,1,2 or 3\n");`
9	`    GOTO LINE 18;`
10	`    printf("Exception: %d\n", 10);`
11	`    GOTO LINE 18;`
12	`    printf("Bool exception \n");`
13	`    GOTO LINE 18;`
14	`    printf("Double exception: %lf \n",1.2);`
15	`    GOTO LINE 18;`
16	`    printf("Generic exception !");`
17	`    GOTO LINE 18;`
18	`    printf("End\n");`
19	`    return 0;`
20	`}`
21	

Modeling Exception Behavior

- This is a simulation on how the exception mechanism is translated into a code. It implies ZERO overhead on the normal execution (so only when **throw** is triggered we will jump to a specific address where the catch exists).

App.cpp

```
#include <iostream>
#include <Windows.h>

int main() {
    printf("Start\n");
    try
    {
        int x = GetTickCount() % 10;
        if (x == 0) throw 10;
        if (x == 1) throw true;
        if (x == 2) throw 1.2;
        if (x == 3) throw "Exception text";
        printf("x is not 0,1,2 or 3\n");
    }
    catch (int e) { … }
    catch (bool b) { … }
    catch (double d) { … }
    catch (...) { … }
    printf("End\n");
    return 0;
}
```

Line	Pseudo code
1	`int main() {`
2	`    printf("Start\n");`
3	`    int x = GetTickCount() % 10;`
4	`    if (x == 0) GOTO LINE 10;`
5	`    if (x == 1) GOTO LINE 12;`
6	`    if (x == 2) GOTO LINE 14;`
7	`    if (x == 3) GOTO LINE 16;`
8	`    printf("x is not 0,1,2 or 3\n");`
9	`    GOTO LINE 18;`
10	`    printf("Exception: %d\n", 10);`
11	`    GOTO LINE 18;`
12	`    printf("Bool exception \n");`
13	`    GOTO LINE 18;`
14	`    printf("Double exception: %lf \n",1.2);`
15	`    GOTO LINE 18;`
16	`    printf("Generic exception !");`
17	`    GOTO LINE 18;`
18	`    printf("End\n");`
19	`    return 0;`
20	`}`
21	

Modeling Exception Behavior

- This is a simulation on how the exception mechanism is translated into a code. It implies ZERO overhead on the normal execution (so only when **throw** is triggered we will jump to a specific address where the catch exists).

App.cpp

```
#include <iostream>
#include <Windows.h>

int main() {
    printf("Start\n");
    try
    {
        int x = GetTickCount() % 10;
        if (x == 0) throw 10;
        if (x == 1) throw true;
        if (x == 2) throw 1.2;
        if (x == 3) throw "Exception text";
        printf("x is not 0,1,2 or 3\n");
    }
    catch (int e) { … }
    catch (bool b) { … }
    catch (double d) { … }
    catch (...) { … }
    printf("End\n");
    return 0;
}
```

Pseudo code

```
 1  int main() {
 2      printf("Start\n");
 3      int x = GetTickCount() % 10;
 4      if (x == 0) GOTO LINE 10;
 5      if (x == 1) GOTO LINE 12;
 6      if (x == 2) GOTO LINE 14;
 7      if (x == 3) GOTO LINE 16;
 8      printf("x is not 0,1,2 or 3\n");
 9      GOTO LINE 18;
10      printf("Exception: %d\n", 10);
11      GOTO LINE 18;
12      printf("Bool exception \n");
13      GOTO LINE 18;
14      printf("Double exception: %lf \n",1.2);
15      GOTO LINE 18;
16      printf("Generic exception !");
17      GOTO LINE 18;
18      printf("End\n");
19      return 0;
20  }
21
```

Modeling Exception Behavior

- This is a simulation on how the exception mechanism is translated into a code. It implies ZERO overhead on the normal execution (so only when **throw** is triggered we will jump to a specific address where the catch exists).

App.cpp

```
#include <iostream>
#include <Windows.h>

int main() {
    printf("Start\n");
    try
    {
        int x = GetTickCount() % 10;
        if (x == 0) throw 10;
        if (x == 1) throw true;
        if (x == 2) throw 1.2;
        if (x == 3) throw "Exception text";
        printf("x is not 0,1,2 or 3\n");
    }
    catch (int e) { … }
    catch (bool b) { … }
    catch (double d) { … }
    catch (...) { … }
    printf("End\n");
    return 0;
}
```

Pseudo code

```
 1  int main() {
 2      printf("Start\n");
 3      int x = GetTickCount() % 10;
 4      if (x == 0) GOTO LINE 10;
 5      if (x == 1) GOTO LINE 12;
 6      if (x == 2) GOTO LINE 14;
 7      if (x == 3) GOTO LINE 16;
 8      printf("x is not 0,1,2 or 3\n");
 9      GOTO LINE 18;
10      printf("Exception: %d\n", 10);
11      GOTO LINE 18;
12      printf("Bool exception \n");
13      GOTO LINE 18;
14      printf("Double exception: %lf \n",1.2);
15      GOTO LINE 18;
16      printf("Generic exception !");
17      GOTO LINE 18;
18      printf("End\n");
19      return 0;
20  }
21
```

Modeling Exception Behavior

- This is a simulation on how the exception mechanism is translated into a code. It implies ZERO overhead on the normal execution (so only when **throw** is triggered we will jump to a specific address where the catch exists).

App.cpp

```
#include <iostream>
#include <Windows.h>
```

```
    }
    catch (int e) { … }
    catch (bool b) { … }
    catch (double d) { … }
    catch (...) { … }
    printf("End\n");
    return 0;
}
```

Line — Pseudo code

```
 1  int main() {
 2      printf("Start\n");
 3      int x = GetTickCount() % 10;
 4      if (x == 0) GOTO LINE 10;
 5      if (x == 1) GOTO LINE 12;
 6      if (x == 2) GOTO LINE 14;
 7      if (x == 3) GOTO LINE 16;
 8      printf("x is not 0,1,2 or 3\n");
 9      GOTO LINE 18;
10      printf("Exception: %d\n", 10);
```

```
,1.2);
```

```
21
```

Please note that GOTO LINE reflect a call to `CxxThrowException` that implies:

1. Look at the type of the expression
2. Based on the type identify the offset (in our case the line number) where the execution should jump.

Type	Line number
int	10
bool	12
double	14
<everything else>	16

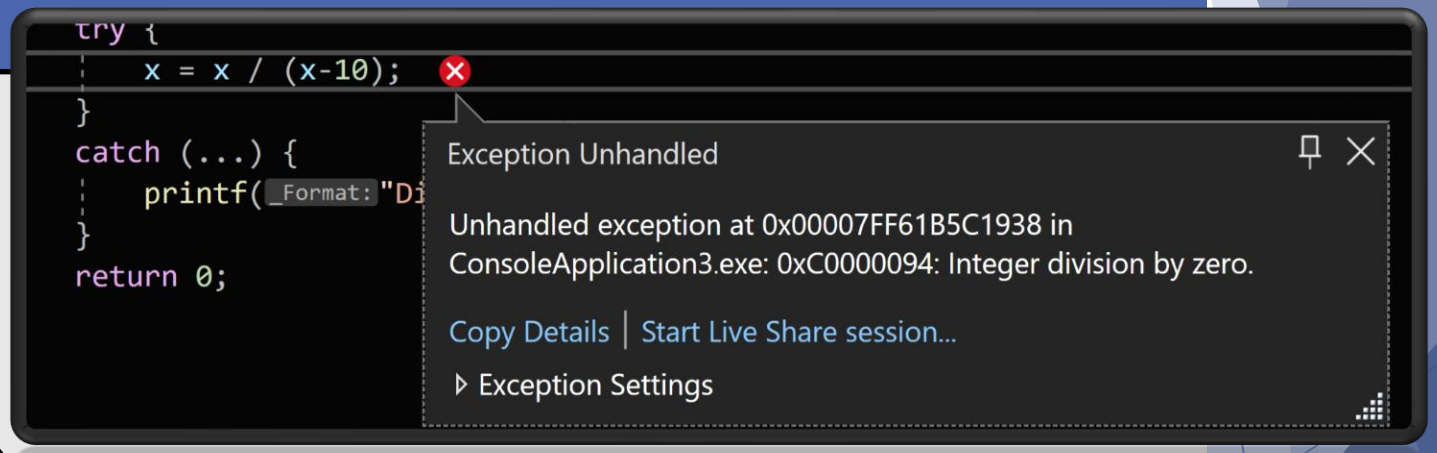
SEH (Structured Exception Handling)

SEH

- Let’s analyze the following code:

App.cpp

```
#include <iostream>

int main() {
     int x = 10;
     try {
          x = x / (x-10);
     }
     catch (...) {
          printf("Division by 0");
     }
     return 0;
}
```



- **If we have used a try…catch block, why a division by 0 exception was not caught ?** (especially since we have used catch(…) format – that should have matched every exception).

SEH

- C++ exceptions are designed to work with exceptions that originates form a throw instruction (exceptions that are managed by the C++ runtime / compiler).
- At the same time, there are hardware exceptions (such as **Division By 0**) that are handled by the OS directly through a different mechanism
- Other exceptions that fall into the same category:
  - Invalid assembly opcode that is about to be executed
  - Memory access violation (trying to access a memory address that is invalid / not allocated)
  - Stack overflow
  - Hardware breakpoint hit
  - etc

SEH

- To handle those extensions, use the **__try** and **__except** keywords with the following format:

```
__try  { // cod to be executed }
__except (<expression>) { // exception block }
          or
__finally  { // usually cleanup code }
```

- where <expression> should evaluate into one of the following:

Value	Meaning
EXCEPTION_EXECUTE_HANDLER	Execute he code in the __except block
EXCEPTION_CONTINUE_SEARCH	Exception will not be handled. Instead, it will be passed to another exception handle from the SEH structure.
EXCEPTION_CONTINUE_EXECUTION	Ignore exception and **resume execution** at the point where the exception occurred. ***This is very dangerous and never recommended !!!!***

SEH

- The original code can be converted as follows:

App.cpp

```
#include <iostream>
#include <windows.h>

int main() {
     int x = 10;
     __try {
          x = x / (x-10);
     }
     __except (EXCEPTION_EXECUTE_HANDLER) {
          printf("Division by 0");
     }
     return 0;
}
```

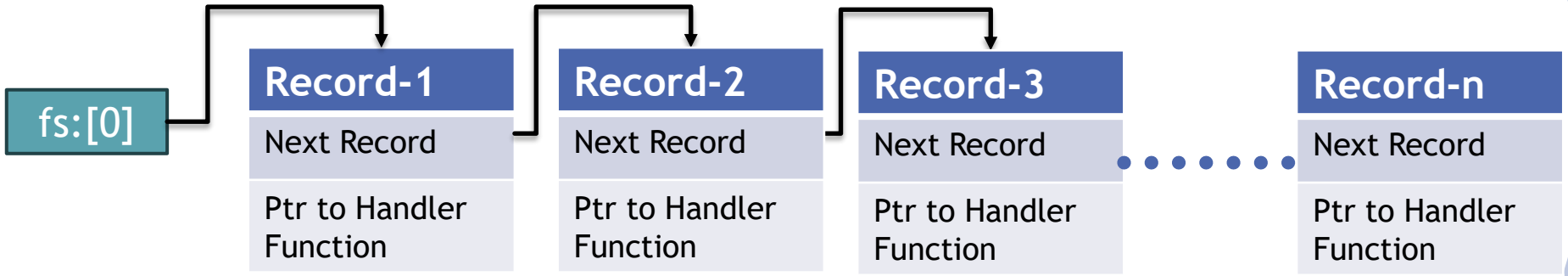
Output:

```
Division by 0
```

- Compiled with: /JMC /permissive- /ifcOutput "x64\Debug\" /GS /W3 /Zc:wchar_t /ZI /Gm- /Od /sdl /Fd"x64\Debug\vc143.pdb" /Zc:inline /fp:precise /D "_DEBUG" /D "_CONSOLE" /D "_UNICODE" /D "UNICODE" /errorReport:prompt /WX- /Zc:forScope /RTC1 /Gd /MDd /std:c++14 /FC /Fa"x64\Debug\" **/EHa** /nologo /Fo"x64\Debug\" /Fp"x64\Debug\ConsoleApplication3.pch" /diagnostics:column

SEH

- This type of exceptions are handled in Windows OS via SEH (**S**tructured **E**xception **H**andling)



- In practice this is a single-linked list
- When a hardware exception is being triggered, the compiler passes through this list until the exception is being handled (based on those 3 values that an __except keywords receives).

SEH

- Let’s analyze the following code:

App.cpp

```
#include <iostream>
#include <windows.h>

int main() {
     int* x = nullptr;
     __try {
          *x = 100;
     }
     __except (EXCEPTION_EXECUTE_HANDLER) {
          printf("Invalid memory access");
     }
     return 0;
}
```

Output:

```
Invalid memmory access
```

SEH

- Let’s analyze the following code:

App.cpp

```
#include <iostream>
#include <windows.h>

int main() {
    int* x = nullptr;
    __try {
        *x = 100;
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        printf("Invalid memory access");
    }
    return 0;
}
```

ASM Code

```
mov         x, 2
```

SEH

- Let's analyze the following code:

App.cpp

```
#include <iostream>
#include <windows.h>

int main() {
     int* x = nullptr;
     __try {
          *x = 100;
     }
     __except (EXCEPTION_EXECUTE_HANDLER) {
          printf("Invalid memory access");
     }
     return 0;
}
```

ASM Code

```
push    __except_handler
push    fs:[0]
mov     fs:[0], esp
```

Where **__except_handler** is a runtime code that will execute the code from the __except block.

SEH

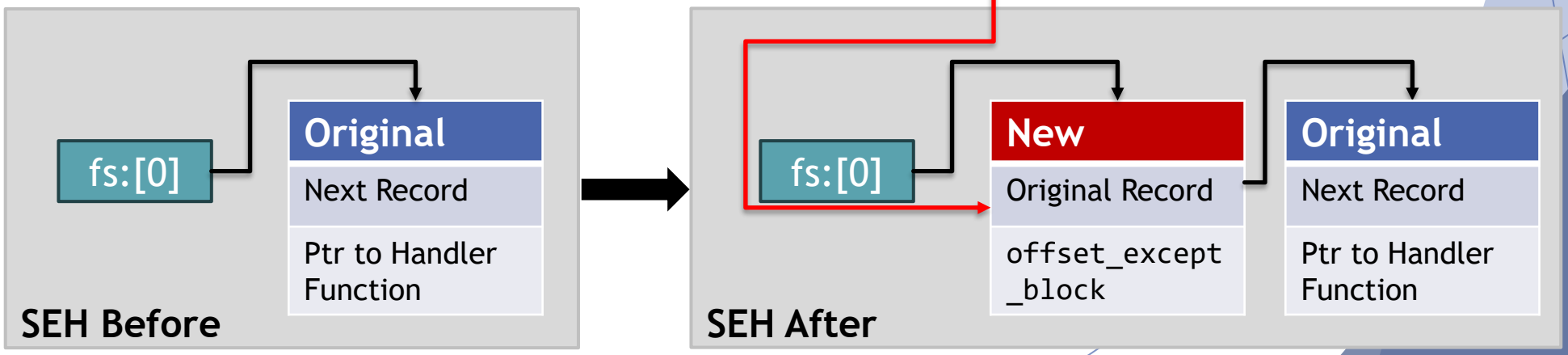
- Let's analyze the following code:

App.cpp

```
#include <iostream>
#include <windows.h>

int main() {
     int* x = nullptr;
     __try {
     ….
```

ASM Code

```
push    offset_except_block
push    fs:[0]
mov     fs:[0], esp
```



SEH

- Let’s analyze the following code:

App.cpp

```
#include <iostream>
#include <windows.h>

int main() {
     int* x = nullptr;
       try {
          *x = 100;
     }
     __except (EXCEPTION_EXECUTE_HANDLER) {
          printf("Invalid memory access");
     }
     return 0;
}
```

ASM Code

```
mov          [x], 100
```

This is where the exception was triggered (as “x” is nullptr)

SEH

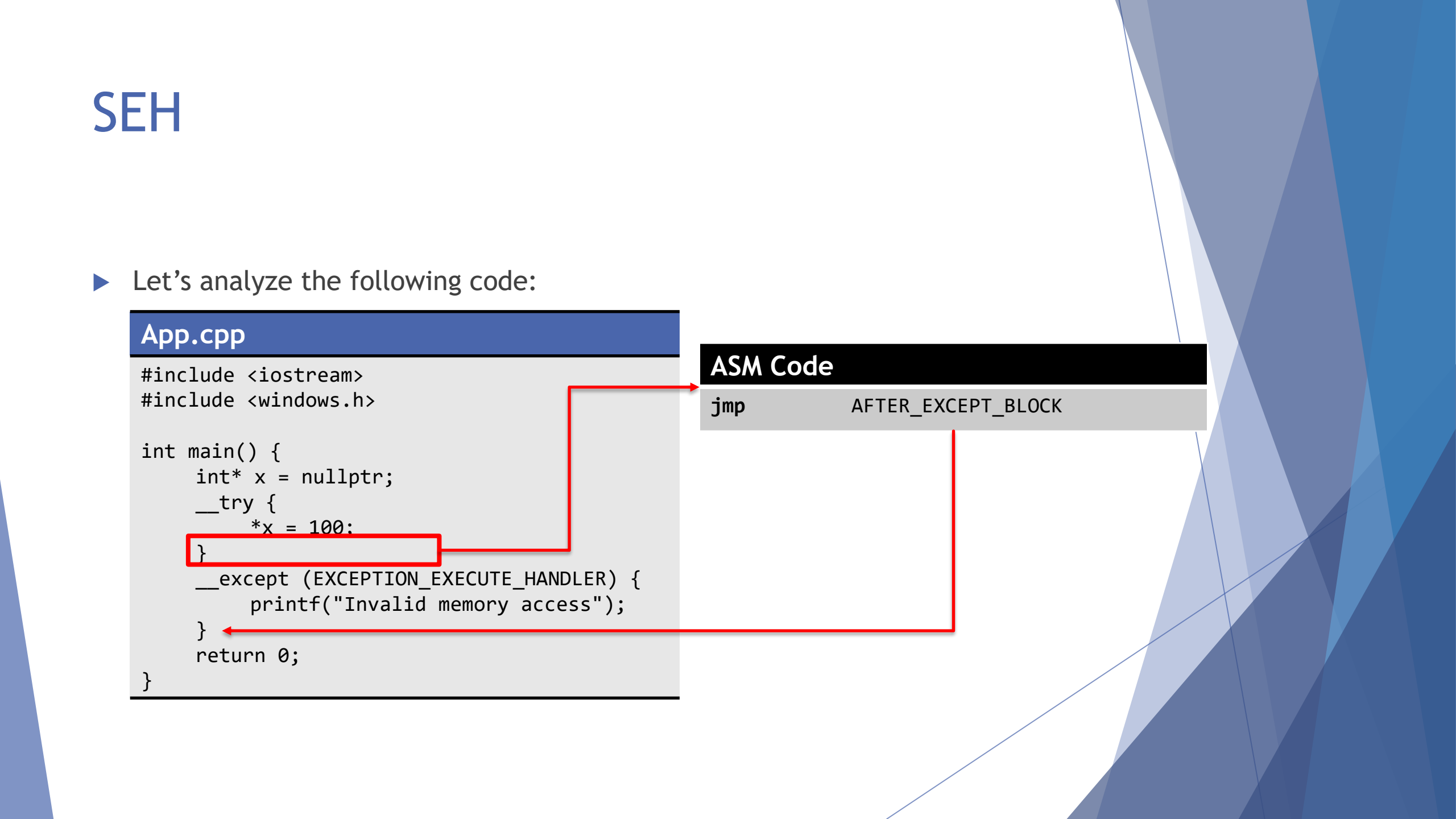
- Let’s analyze the following code:

App.cpp

```
#include <iostream>
#include <windows.h>

int main() {
     int* x = nullptr;
     __try {
          *x = 100;
     }
     __except (EXCEPTION_EXECUTE_HANDLER) {
          printf("Invalid memory access");
     }
     return 0;
}
```

ASM Code

```
jmp          AFTER_EXCEPT_BLOCK
```



SEH

- Let's analyze the following code:

App.cpp

```
#include <iostream>
#include <windows.h>

int main() {
     int* x = nullptr;
     __try {
          *x = 100;
     }
     __except (EXCEPTION_EXECUTE_HANDLER) {
          printf("Invalid memory access");
     }
     return 0;
}
```

ASM Code

```
push   offset of "Invalid memory access"
call   printf
jmp    AFTER_EXCEPT_BLOCK
```

This code is being called indirectly from the runtime C/C++ framework **IF AN EXCEPTION HAPPENS**

SEH

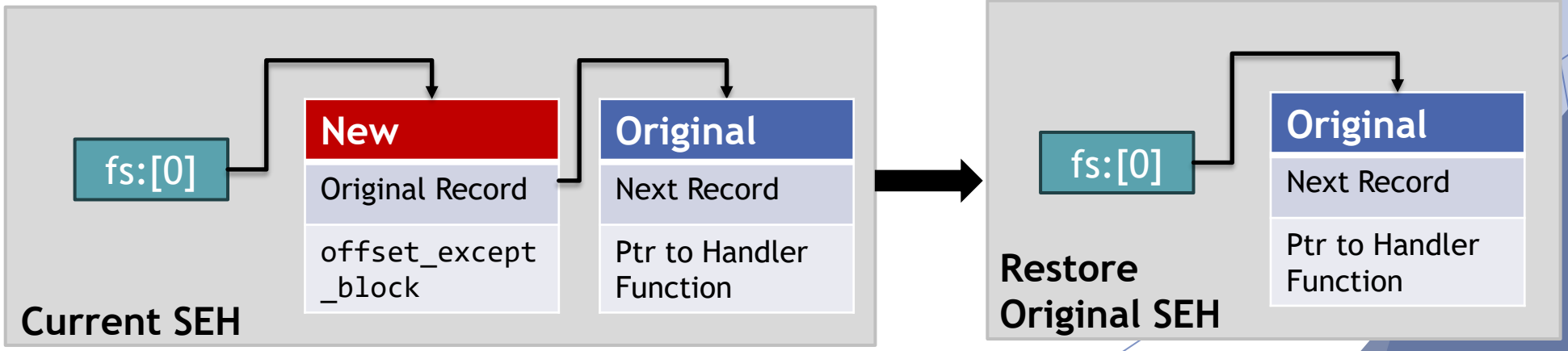
- Let's analyze the following code:

App.cpp

```
int main() {
     int* x = nullptr;
     __try {…}
     __except (…}
     return 0;
}
```

ASM Code

```
mov     eax, dword ptr [esp]
mov     fs:[0], eax
add     esp, 8
-----------------------------------------
xor     eax,eax
```



SEH

- This SEH mechanism is specific to Windows (32 bytes)
- A similar mechanisms exists for a **64 bytes** but implies a different code generation mechanism (exception data is store in .pdata and .xdata sections)
- It is **not a zero abstraction** (some code will be added and executed even if the exception is not triggered).

SEH

- Let’s analyze the following code:

App.cpp

```
#include <iostream>
#include <windows.h>

int main() {
     int* x = nullptr;
     __try {
          *x = 100;
          printf("Pointer set up corectly \n");
     }
     __except (EXCEPTION_EXECUTE_HANDLER) {
          printf("Invalid memory access \n");
     }
     __finally {
          printf("Finally block\n");
     }
     return 0;
}
```

Code will not compile (we can not have __except and __finally at the same time).

SEH

- If we remove the **__finally** block, the code compiles:

App.cpp

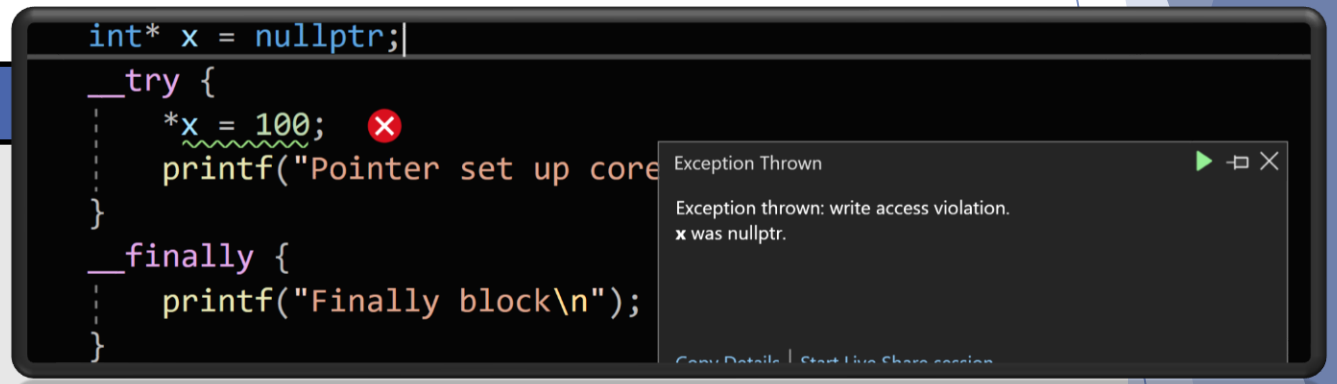
```
#include <iostream>
#include <windows.h>

int main() {
     int* x = nullptr;
     __try {
          *x = 100;
          printf("Pointer set up corectly \n");
     }
     __except (EXCEPTION_EXECUTE_HANDLER) {
          printf("Invalid memory access \n");
     }
     return 0;
}
```

Output:

```
Invalid memmory access
```

SEH

- If we remove the **__except** block, the code compiles, but crashed on runtime as the exception is not handled.



App.cpp

```
#include <iostream>
#include <windows.h>

int main() {
     int* x = nullptr;
     __try {
          *x = 100;
          printf("Pointer set up corectly \n");
     }
     __finally {
          printf("Finally block\n");
     }
     return 0;
}
```

SEH

- To create a code where we can use both **__except** and **__finally** we need to build two nested **__try** blocks.

```
__try {
    __try {
        // cod to be executed
    }
    __except (<expression>) {
        // exception block
    }
}
__finally {
    // usually cleanup code
}
```

SEH

- The modified code that has both **__except** and **__finally** will look as follows:

App.cpp

```
#include <windows.h>
#include <stdio.h>

int main() {
    __try {
        __try {
            int* x = nullptr;
            *x = 100;
            printf("Pointer set up corectly \n");
        }
        __except (EXCEPTION_EXECUTE_HANDLER) {
            printf("Invalid memory access \n");
        }
    }
    __finally {
        printf("Finally block\n");
    }
    return 0;
}
```

Output:

```
Invalid memmory access
Finally block
```

SEH

- Changing ”x” to a valid pointer:

App.cpp

```
#include <windows.h>
#include <stdio.h>

int main() {
    __try {
        __try {
            int* x = = new int;
            *x = 100;
            printf("Pointer set up corectly \n");
        }
        __except (EXCEPTION_EXECUTE_HANDLER) {
            printf("Invalid memory access \n");
        }
    }
    __finally {
        printf("Finally block\n");
    }
    return 0;
}
```

Output:

```
Pointer set up corectly
Finally block
```

- RAII

RAII

- Let's analyze the following code:

App.cpp

```
#include <iostream>

struct A {
    A() { std::cout << "A-CTOR\n"; }
    ~A() { std::cout << "A-DTOR\n"; }
};
struct B {
    B() { std::cout << "B-CTOR\n"; }
    ~B() { std::cout << "B-DTOR\n"; }
};
struct C {
    A* a;
    B* b;
    C() {
        a = new A();
        b = new B();
        std::cout << "C-CTOR\n";
    }
    ~C() {
        delete a;
        delete b;
        std::cout << "C-DTOR\n";
    }
};
```

```
int main() {
    try {
        C c;
    }
    catch (int){
        std::cout << "Exception\n";
    }
    return 0;
}
```

Output:

```
A-CTOR
B-CTOR
C-CTOR
A-DTOR
B-DTOR
C-DTOR
```

What would happen if during the constructor or destructor of class C an exception will be thrown ?

RAII

- Let's analyze the following code:

App.cpp

```
#include <iostream>

struct A {…};
struct B {…};
struct C {
    A* a;
    B* b;
    C() {
        a = new A();

        throw 10;

        b = new B();
        std::cout << "C-CTOR\n";
    }
    ~C() {
        delete a;
        delete b;
        std::cout << "C-DTOR\n";
    }
};

int main() {
    try {
        C c;
    }
    catch (int){
        std::cout << "Exception\n";
    }
    return 0;
}
```

Output:
A-CTOR
Exception

Noticed that we end up in an impossible state where only part of the C object is created (data member "a") and it is not removed from the memory (the destructor is not called).

MEMORY LEAK

RAII

- What if data member “a” is created directly (not on the heap):

App.cpp

```
#include <iostream>

struct A {…};
struct B {…};
struct C {
    A a;
    B* b;
    C() {
        throw 10;
        b = new B();
        std::cout << "C-CTOR\n";
    }
    ~C() {
        delete b;
        std::cout << “C-DTOR\n";
    }
};

int main() {
    try {
        C c;
    }
    catch (int){
        std::cout << "Exception\n";
    }
    return 0;
}
```

Output:

```
A-CTOR
A-DTOR
Exception
```

The destructor for object C is not called. However, the destructor for data member “a” is being called.

ALL OK

RAII

- In C++, even if a constructor throws, **members that were constructed before the throw** will be destroyed when their lifetime ends (their destructor – if any will be called).
- This rule applies to:
  - Non-pointer members (pretty much any data member that is not a pointer)
  - Base classes
  - RAII wrappers (e.g. unique_ptr)

RAII

- RAII stands for **R**esource **A**cquisition **I**s **I**nitialization and is a C++ idiom that is based on the following ideas:
  - When an object is created, it **acquires** a resource (a memory where it resides, handled, mutexes, etc)
  - When the object is destroyed (goes out of scope), its **destructor releases** the resource
- This means that automatic cleanup is possible – no matter how a function / method exits (normal or exception)
- RAII ensures that:
  - Memory / resource leaks are prevented
  - Provides exception safety (cleanup)

RAII

- The original code (using RAII) will look like this:

App.cpp

```
#include <iostream>

struct A {…};
struct B {…};
struct C {
    std::unique_ptr<A> a;
    std::unique_ptr<B> b;
    C() {
        a = std::make_unique<A>();
        throw 10;
        b = std::make_unique<B>();
        std::cout << "C-CTOR\n";
    }
    ~C() {
        std::cout << "C-DTOR\n";
    }
};

int main() {
    try {
        C c;
    }
    catch (int){
        std::cout << "Exception\n";
    }
    return 0;
}
```

Output:

```
A-CTOR
A-DTOR
Exception
```

Now we have a correct clean up of allocated memory (data member “a” was the only one that was created, it was also the only one destroyed)

ALL OK

RAII

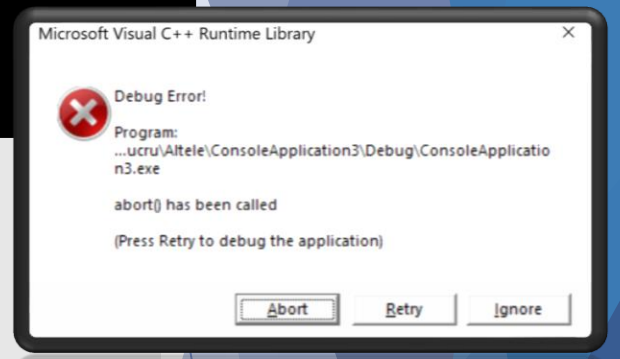
- Let's throw an exception, but as part of the destructor:

App.cpp

```
#include <iostream>

struct A {…};
struct B {…};
struct C {
    A* a;
    B* b;
    C() {
        a = new A();
        b = new B();
        std::cout << "C-CTOR\n";
    }
    ~C() {
        delete a;

        throw 10;

        delete b;
        std::cout << "C - DTOR\n";
    }
};

int main() {
    try {
        C c;
    }
    catch (int){
        std::cout << "Exception\n";
    }
    return 0;
}
```

Output:

```
A-CTOR
B-CTOR
C-CTOR
A-DTOR
```



Notice that we don't get to the point where the code from the catch is being called ("Exception" is not printed on the screen). Instead we get a runtime crash !

RUNTIME CRASH

RAII

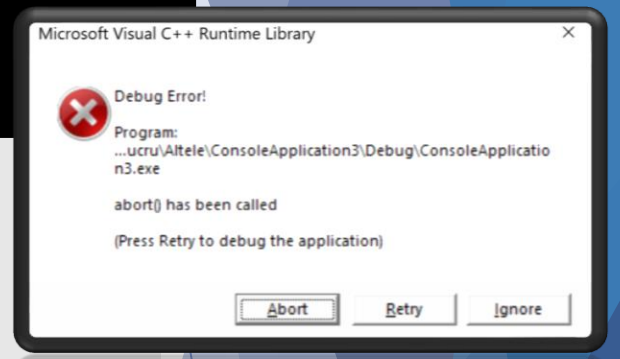
- Switching to a RAII model produces the same result.

App.cpp

```
#include <iostream>

struct A {…};
struct B {…};
struct C {
    std::unique_ptr<A> a;
    std::unique_ptr<B> b;
    C() {
        a = std::make_unique<A>();
        b = std::make_unique<B>();
        std::cout << "C-CTOR\n";
    }
    ~C() {
        throw 10;
        std::cout << "C-DTOR\n";
    }
};

int main() {
    try {
        C c;
    }
    catch (int){
        std::cout << "Exception\n";
    }
    return 0;
}
```

Output:

```
A-CTOR
B-CTOR
C-CTOR
```



Notice that we don't get to the point where the code from the catch is being called ("Exception" is not printed on the screen). Instead we get a runtime crash !

RUNTIME CRASH

RAII

- Throwing exceptions from a destructor is an undefined behavior, and most of the time it will create a runtime crash
- In practice, a **std::terminate()** is being called – this also happens if you throw and exception while another exception is already propagating.
- The solution is to make sure your don't throw anything from a destructor (e.g. you check the operations that you want to perform before performing them).

RAII

- We can also use **std::set_terminate()** to get notification when an application is being closed due to a std::terminate() call (in our case from a destructor).

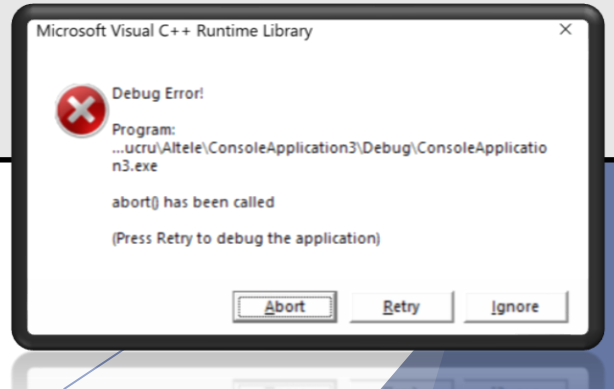
App.cpp

```
#include <iostream>

struct A {…};
struct B {…};
struct C {
    std::unique_ptr<A> a;
    std::unique_ptr<B> b;
    C() {
        a = std::make_unique<A>();
        b = std::make_unique<B>();
        std::cout << "C-CTOR\n";
    }
    ~C() {
        throw 10;
        std::cout << "C-DTOR\n";
    }
};

void notify_terminate() {
    std::cerr << "std::terminate()\n";
}

int main() {
    std::set_terminate(notify_terminate);
    try {
        C c;
    }
    catch (int){
        std::cout << "Exception\n";
    }
    return 0;
}
```

Output:

```
A-CTOR
B-CTOR
C-CTOR
Std::terminate()
```

RUNTIME CRASH



Q & A