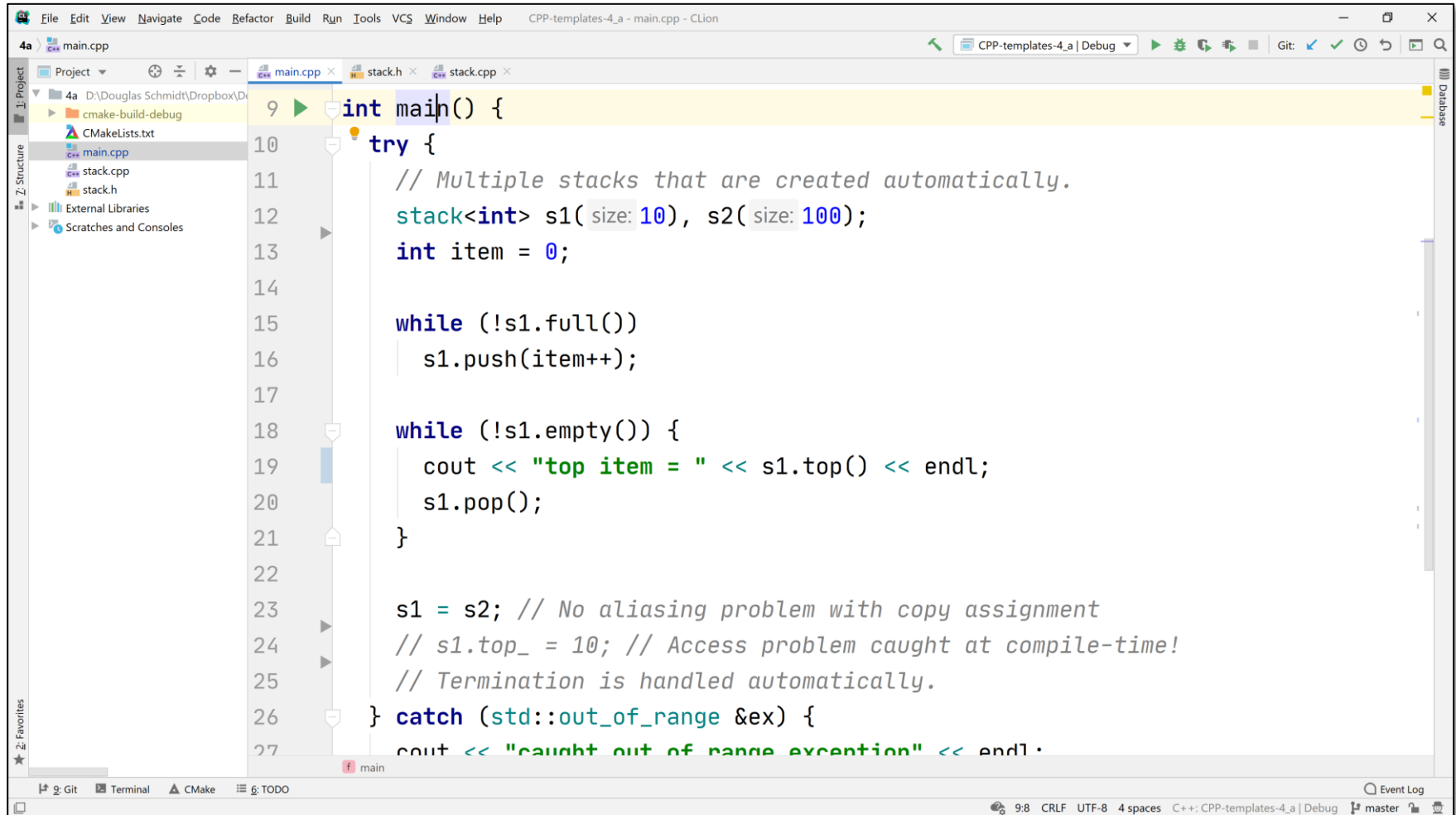


# Template Implementation in C++

- A parameterized type Stack class interface using C++



```
4a  main.cpp
CPP-templates-4_a | Debug
main.cpp x stack.h x stack.cpp x
Project
4a D:\Douglas Schmidt\Dropbox\...
  cmake-build-debug
  CMakeLists.txt
  main.cpp
  stack.cpp
  stack.h
External Libraries
Scratches and Consoles
Z: Structure
2: Favorites

9 int main() {
10     try {
11         // Multiple stacks that are created automatically.
12         stack<int> s1( size: 10), s2( size: 100);
13         int item = 0;
14
15         while (!s1.full())
16             s1.push(item++);
17
18         while (!s1.empty()) {
19             cout << "top item = " << s1.top() << endl;
20             s1.pop();
21         }
22
23         s1 = s2; // No aliasing problem with copy assignment
24         // s1.top_ = 10; // Access problem caught at compile-time!
25         // Termination is handled automatically.
26     } catch (std::out_of_range &ex) {
27         cout << "caught out of range exception" << endl;

```

See [CPlusPlus/tree/master/overview/capabilities/4-C++-templates](https://github.com/Douglas-Schmidt/CPlusPlus/tree/master/overview/capabilities/4-C++-templates)

# Pros of Template Implementation in C++

- All the benefits of C++ data abstraction, plus it is simple to generalize by the type
- We also showcased core patterns/idioms for writing exception-safe C++ code



# Cons of Template Implementation in C++

- Requires programmers to call `full()` & `empty()` explicitly, which means errors can silently creep in..
  - We'll fix this with C++ exceptions features
- Can't customize the implementation at runtime
  - We'll fix this with C++ object-oriented programming features



---

# End of C++ Generic Programming Stack Implementation

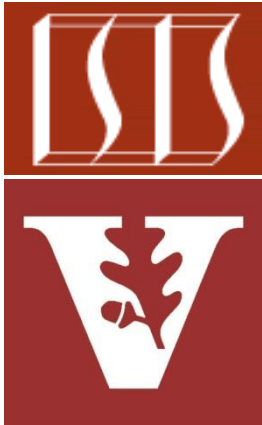
---

# Evolution of Programming Abstraction Mechanisms: C++ Exception Handling

**Douglas C. Schmidt**

**[d.schmidt@vanderbilt.edu](mailto:d.schmidt@vanderbilt.edu)**

**[www.dre.vanderbilt.edu/~schmidt](http://www.dre.vanderbilt.edu/~schmidt)**



**Professor of Computer Science**

**Institute for Software  
Integrated Systems**

**Vanderbilt University  
Nashville, Tennessee, USA**



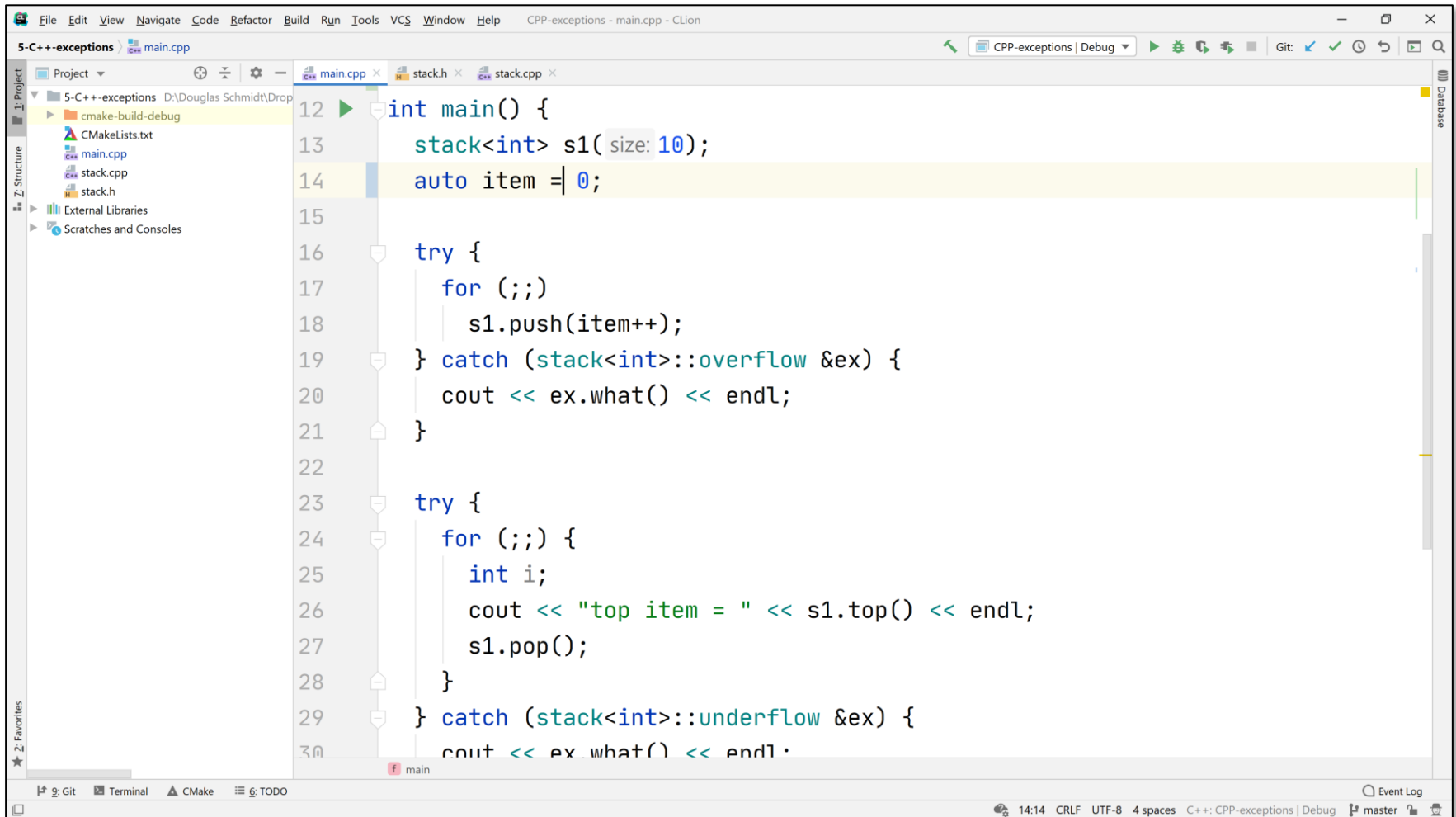
---

# C++ Exception Handling Stack Implementation

---

# Exception Handling Implementation in C++

- C++ exceptions separate error handling from normal processing



The screenshot shows a C++ IDE with a project named "5-C++-exceptions". The main.cpp file is open, showing the following code:

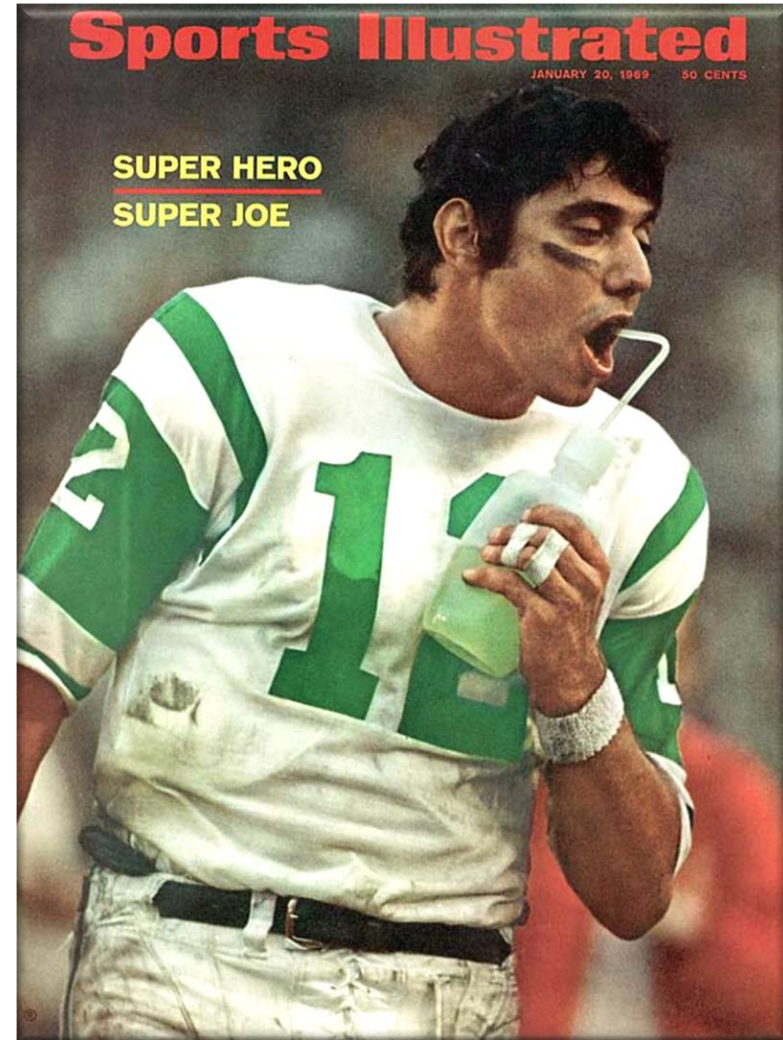
```
12 int main() {
13     stack<int> s1( size: 10);
14     auto item = 0;
15
16     try {
17         for (;;)
18             s1.push(item++);
19     } catch (stack<int>::overflow &ex) {
20         cout << ex.what() << endl;
21     }
22
23     try {
24         for (;;) {
25             int i;
26             cout << "top item = " << s1.top() << endl;
27             s1.pop();
28         }
29     } catch (stack<int>::underflow &ex) {
30         cout << ex.what() << endl;
31     }
```

The code demonstrates two scenarios of stack overflow and underflow. The first try-catch block pushes elements until an overflow occurs, and the second try-catch block pops elements until an underflow occurs. Both scenarios are caught and the error message is printed.

See [CPlusPlus/tree/master/overview/capabilities/5-C++-exceptions](https://github.com/DouglasC-Schmidt/CPlusPlus/tree/master/overview/capabilities/5-C++-exceptions)

# Exception Handling Implementation in C++

- There are several types of exception handling “guarantees”



See [www.boost.org/community/exception\\_safety.html](http://www.boost.org/community/exception_safety.html) for more info



# Exception Handling Implementation in C++

- There are several types of exception handling “guarantees”
  - *No* guarantee – memory can be leaked, invariants of a component are not preserved, etc.

```
template<typename T>
stack<T> &
stack<T>::operator=(const
stack<T> &s) {
    if (this != &s) {
        T *t = new T[s.size_];
        for (size_t i = 0; i <
            s.size_; ++i)
            t[i] = s.stack_[i];

        delete [] stack_;
        stack_ = t;
        top_ = s.top_;
        size_ = s.size_;
    }
    return *this;
}
```

# Exception Handling Implementation in C++

- There are several types of exception handling “guarantees”
  - *No* guarantee – memory can be leaked, invariants of a component are not preserved, etc.
  - The *basic* guarantee – the invariants of a component are preserved & no resources are leaked

```
template<typename T>
stack<T> &
stack<T>::operator=(const
stack<T> &s) {
    if (this != &s) {
        try {
            T *t = new T[s.size_];
            for (size_t i = 0; i <
                s.size_; ++i)
                t[i] = s.stack_[i];

            delete [] stack_;
            stack_ = t;
            top_ = s.top_;
            size_ = s.size_;
        } catch (exception &ex) {
            delete [] t;
        } ...
    }
```

# Exception Handling Implementation in C++

- There are several types of exception handling “guarantees”
  - *No* guarantee – memory can be leaked, invariants of a component are not preserved, etc.
  - The *basic* guarantee – the invariants of a component are preserved & no resources are leaked
  - The *strong* guarantee – the operation either completes successfully or throws an exception, leaving the program state exactly as it was before the operation started

```
template<typename T>
stack<T> &
stack<T>::operator=(const
                    stack<T> &rhs) {
    if (this != &rhs)
        stack<T>(rhs).swap(*this);
    return *this;
}
```

# Exception Handling Implementation in C++

- There are several types of exception handling “guarantees”
  - *No* guarantee – memory can be leaked, invariants of a component are not preserved, etc.
  - The *basic* guarantee – the invariants of a component are preserved & no resources are leaked
  - The *strong* guarantee – the operation either completes successfully or throws an exception, leaving the program state exactly as it was before the operation started
  - The *no-throw* guarantee – that the operation will not throw an exception

```
template<typename T>
class stack {
public:
    class overflow {};
    class underflow {};
    ...
    stack(stack &&rhs)
        noexcept;

    stack &operator=(stack &&rhs)
        noexcept;

    void swap(stack &rhs)
        noexcept;
} ...
```

# Pros of Exception Handling Implementation

- **Pros**

- Exception handling provides a disciplined way of dealing with erroneous run-time problems by separating error handling from normal code
- Exception handling makes it possible to deal with constructor failures in C++



# Cons of Exception Handling Implementation

- **Cons**

- Exceptions are hard to program correctly if you don't apply the patterns/idioms we've discussed
  - e.g., due to the chances for resource leaks and/or corruption
- Exceptions can yield increased time/space overhead in programs



---

# End of C++ Exception Handling Stack Implementation

---