Template Implementation in C++

- A parameterized type Stack class interface using C++

```cpp
int main() {
    try {
        // Multiple stacks that are created automatically.
        stack<int> s1(size: 10), s2(size: 100);
        int item = 0;

        while (!s1.full())
            s1.push(item++);

        while (!s1.empty()) {
            cout << "top item = " << s1.top() << endl;
            s1.pop();
        }

        s1 = s2; // No aliasing problem with copy assignment
        // s1.top_ = 10; // Access problem caught at compile-time!
        // Termination is handled automatically.
    } catch (std::out_of_range &ex) {
        cout << "caught out of range exception" << endl;
    }
}
```

See 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
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

```cpp
int main() {
    stack<int> s1(size: 10);
    auto item = 0;

    try {
        for (;;) {
            s1.push(item++);
        }
        catch (stack<int>::overflow &ex) {
            cout << ex.what() << endl;
        }
    }
    catch (stack<int>::underflow &ex) {
        cout << ex.what() << endl;
    }
}
```

See 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.

```cpp
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;
}
```

See CPlusPlus/tree/master/overview/capabilities/3-C+++-data-abstraction
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

```cpp
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

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

See CPlusPlus/tree/master/overview/capabilities/4-C++-templates
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

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

See CPlusPlus/tree/master/overview/capabilities/4-C++-templates
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