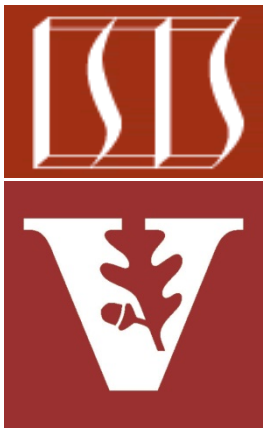
Overview of C++: Part 1

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt



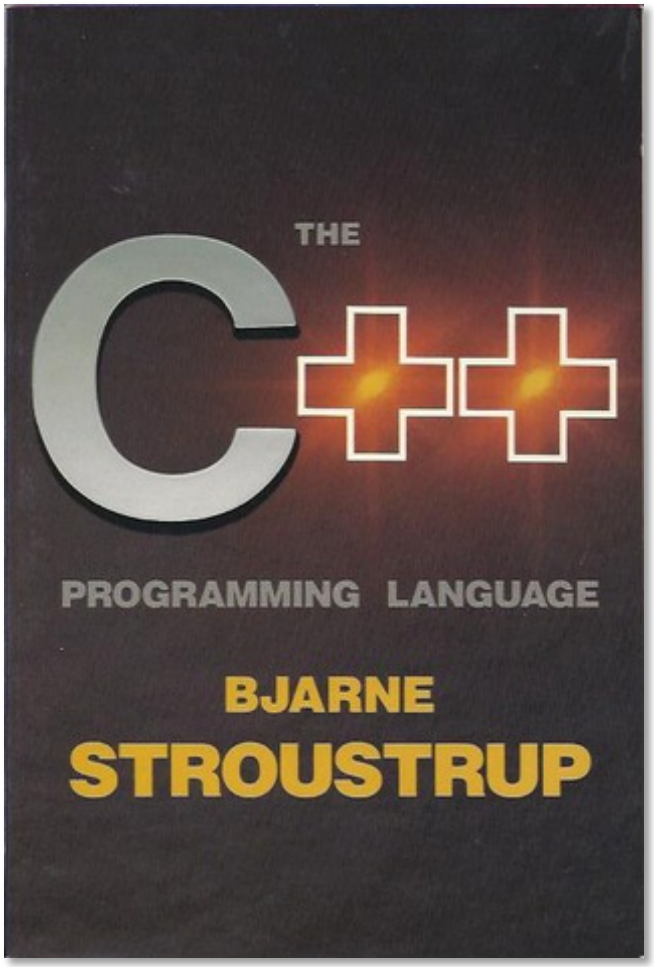
Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

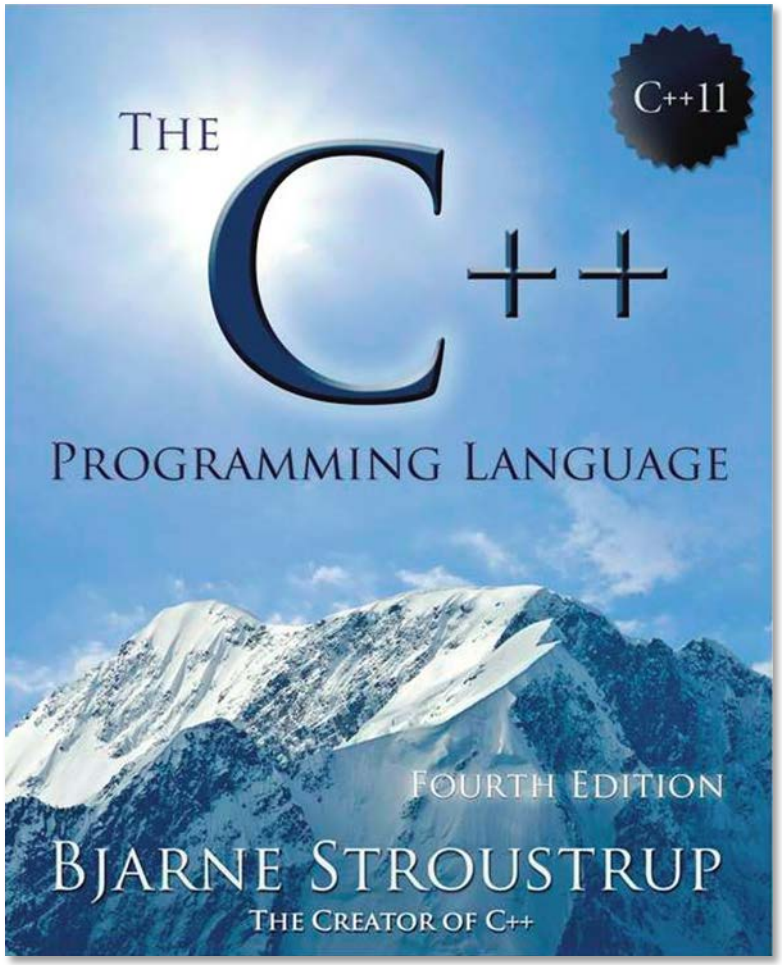
C++ Overview

- C++ was designed at AT&T Bell Labs by Bjarne Stroustrup in the early 80's – over 30 years ago!
  - The original cfront translated C++ into C for portability
    - However, this was hard to debug & was often inefficient
  - Many native host machine compilers now exist
    - e.g., GNU G++, LLVM Clang, Intel C++ Compiler, Microsoft Visual C++, Sun Studio, C++ Builder, Comeau C/C++, etc.



C++ Overview

- C++ is a mostly upwardly compatible extension of C that provides:
  - Stronger typechecking
  - Support for data abstraction
  - Support for object-oriented programming
  - Support for generic programming



C++ Design Goals

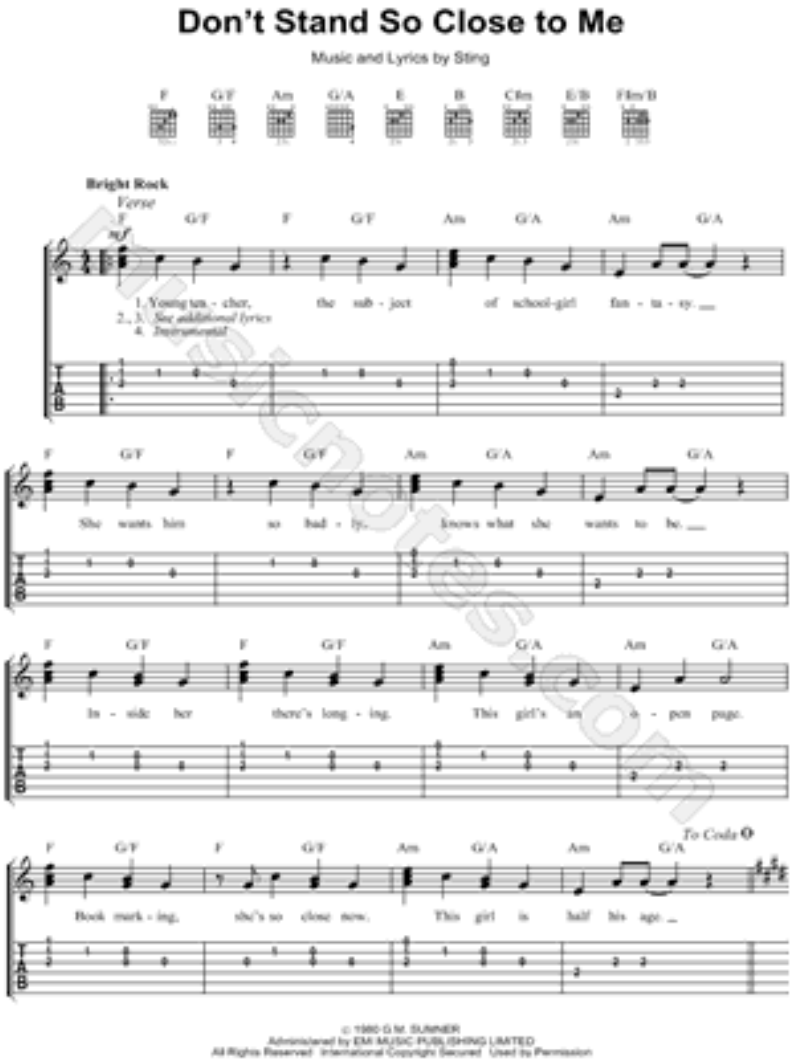
- As with C, run-time efficiency is important
  - Unlike other languages (e.g., Ada, Java, C#, etc.) complicated run-time libraries & virtual machines have not traditionally been needed for C++
  - There is no language-specific support for persistence or distribution in C++
    - Concurrency support was added in C++11

C++ Design Goals

- Compatibility with C libraries & traditional development tools is emphasized, e.g.,
  - Object code reuse
    - e.g., the storage layout of structures is compatible with C, which enables support for the standard ANSI C library, UNIX & Windows system calls via extern blocks, etc.
  - C++ works with the “make” recompilation utility

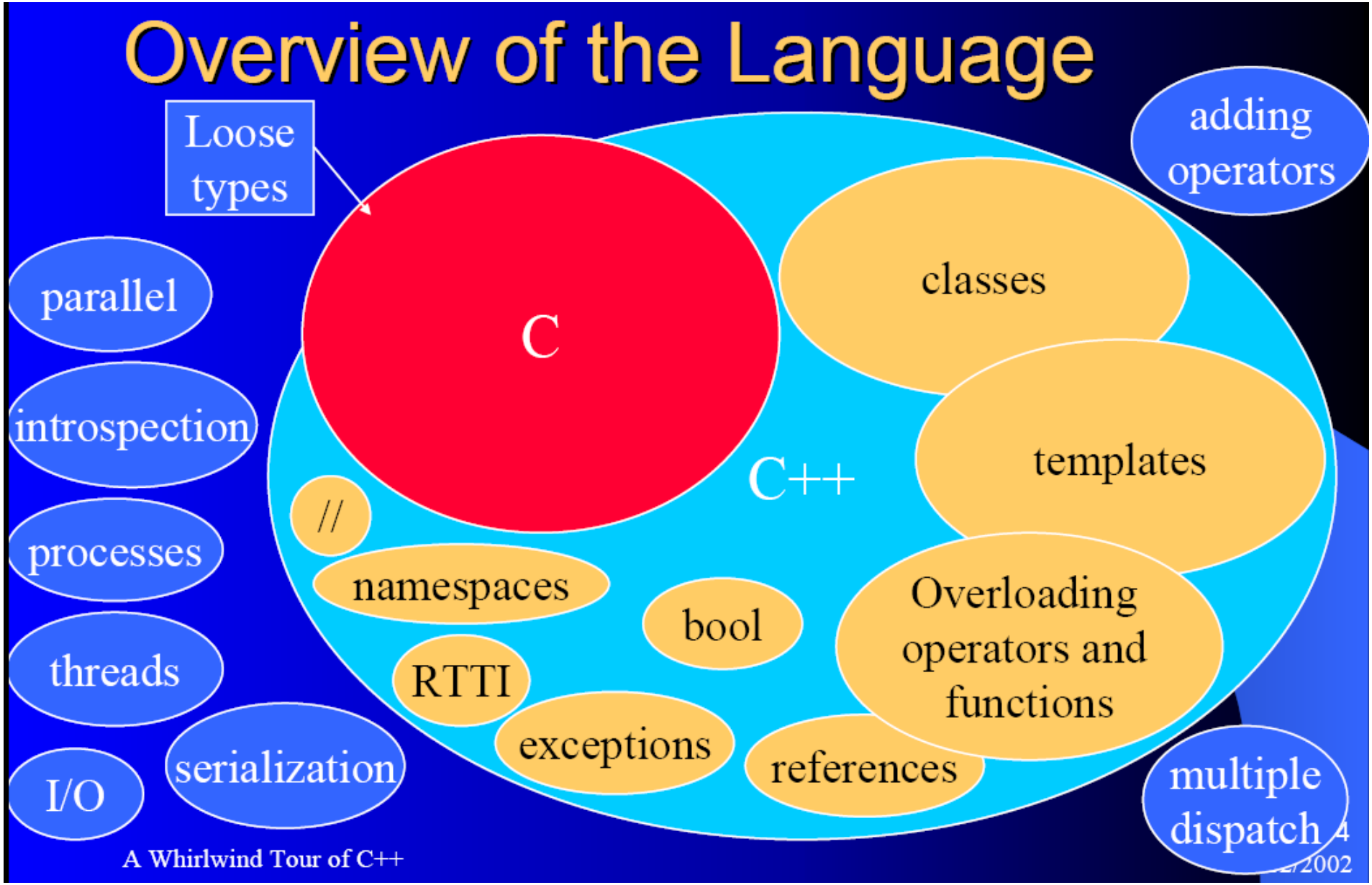
C++ Design Goals

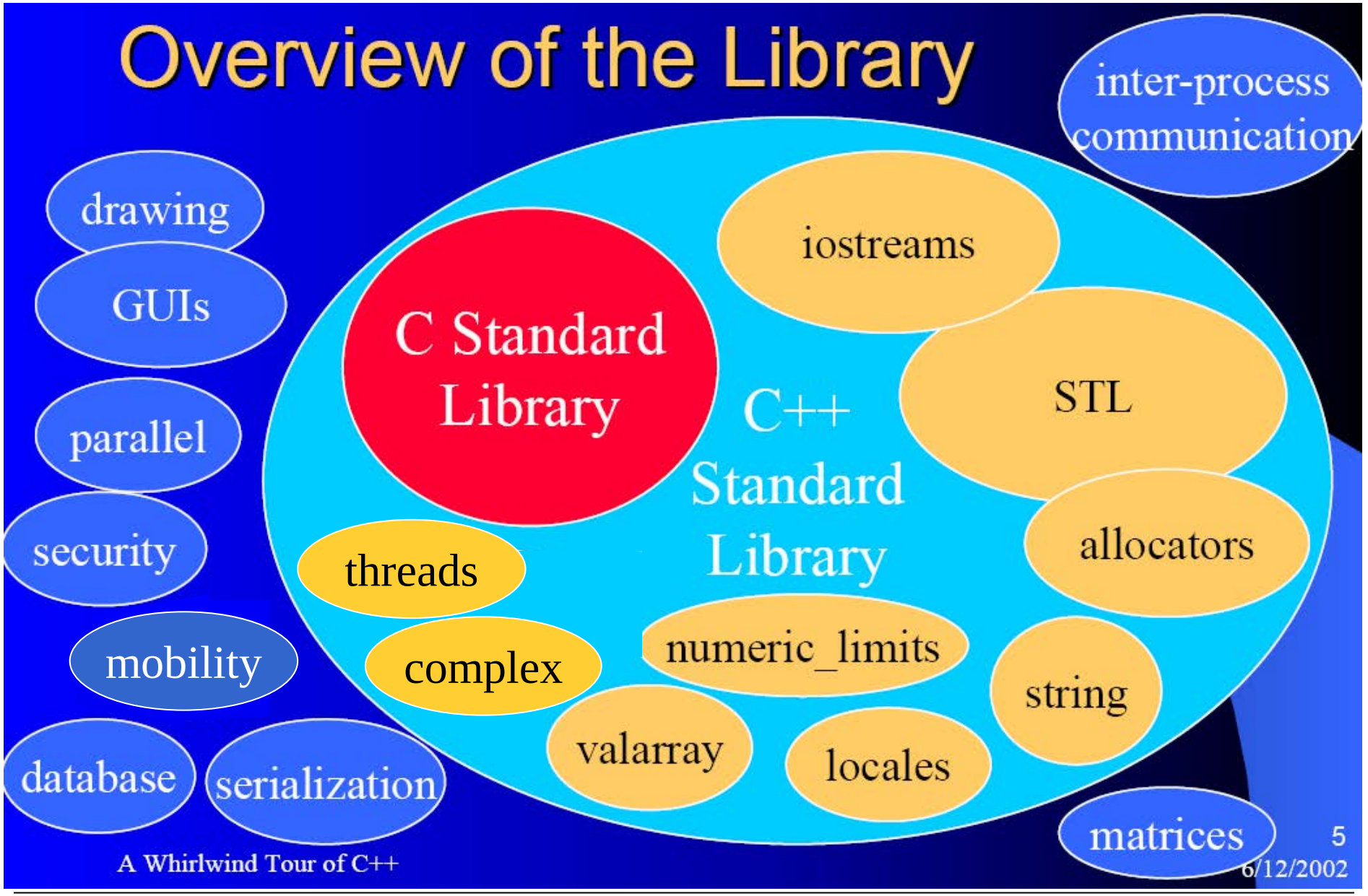
- "As close to C as possible, but no closer"
  - i.e., C++ is not a proper superset of C
    - Backwards compatibility with C is not entirely maintained
    - Typically not a problem in practice...

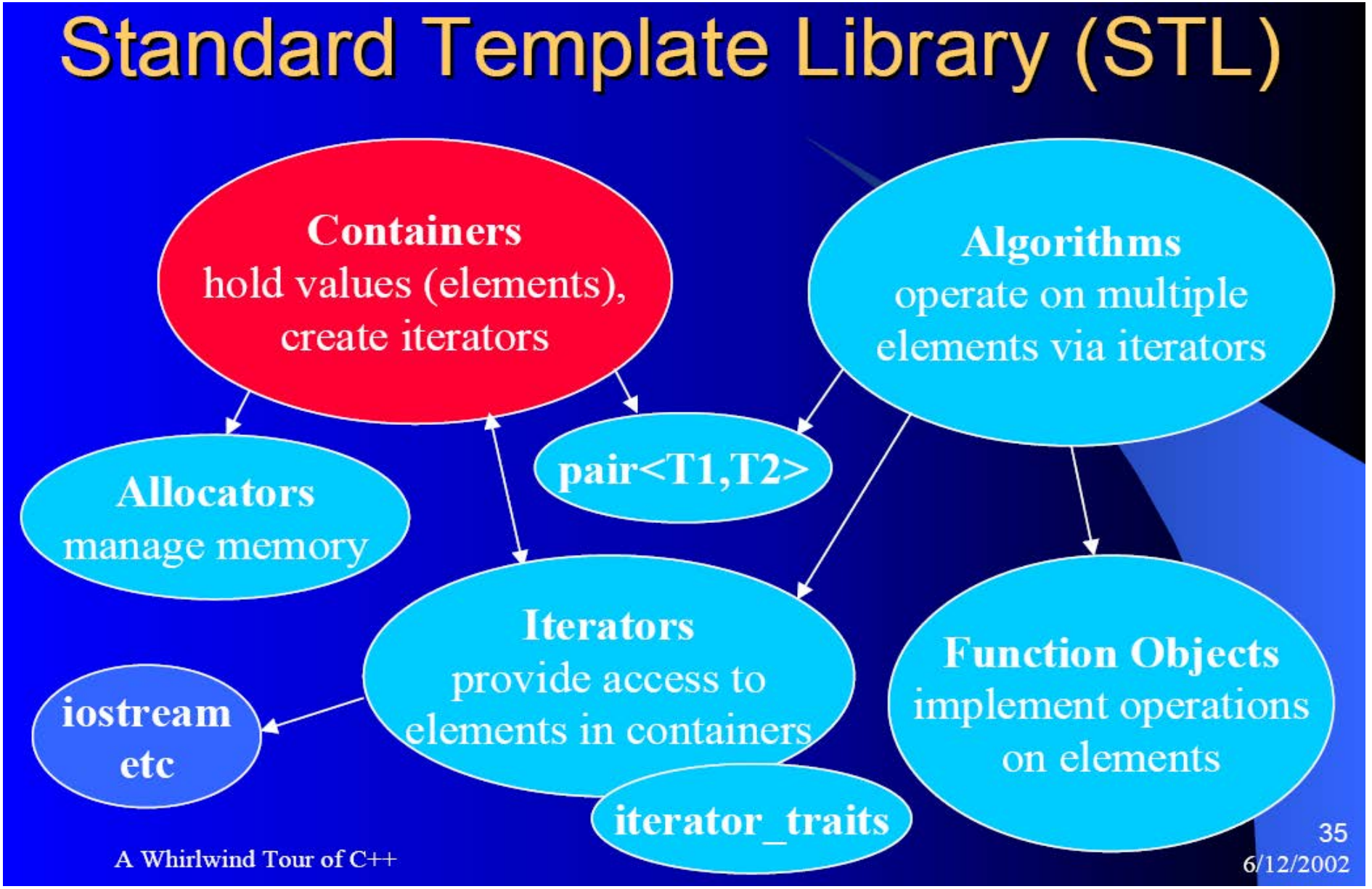


C++ Design Goals

- Certain C++ design goals conflict with modern techniques for:
  - *Compiler optimization*
    - e.g., pointers to arbitrary memory locations complicate register allocation & garbage collection
  - *Software engineering*
    - e.g., separate compilation complicates inlining due to difficulty of interprocedural analysis
    - Dynamic memory management is error-prone


Overview of the Language
Loose types
adding operators
parallel
C
classes
introspection
templates
C++
//
processes
namespaces
bool
Overloading operators and functions
threads
RTTI
exceptions
references
serialization
I/O
multiple dispatch
A Whirlwind Tour of C++
4
2002


Overview of the Library
inter-process communication
drawing
GUIs
parallel
security
mobility
database
serialization
C Standard Library
C++ Standard Library
iostreams
STL
allocators
threads
complex
numeric_limits
string
valarray
locales
matrices
A Whirlwind Tour of C++
5
6/12/2002

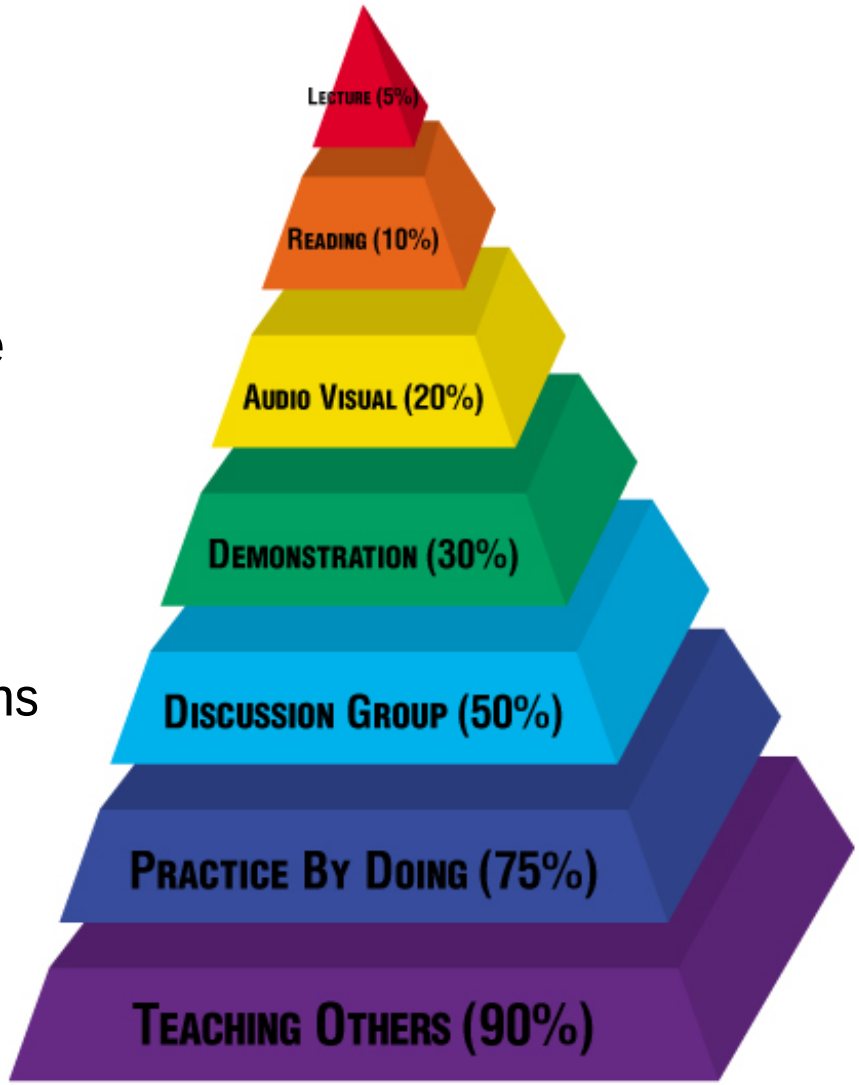

Standard Template Library (STL)
Containers
hold values (elements), create iterators
Algorithms
operate on multiple elements via iterators
Allocators
manage memory
pair<T1,T2>
Iterators
provide access to elements in containers
Function Objects
implement operations on elements
iostream
etc
iterator_traits
A Whirlwind Tour of C++
35
6/12/2002

Key C++ Features

- C++ supports data abstraction & encapsulation
  - e.g., the class mechanism & name spaces
- C++ supports object-oriented programming features
  - e.g., abstract classes, inheritance, & virtual methods
- C++ supports generic programming
  - e.g., parameterized types
- C++ supports sophisticated error handling
  - e.g., exception handling
- C++ supports identifying an object's type at runtime
  - e.g., Run-Time Type Identification (RTTI)

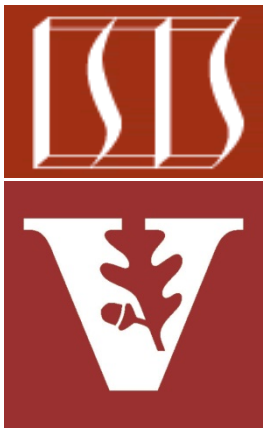
Strategies for Learning C++

- Focus on concepts & design & programming techniques
  - Don't get lost in C++ language features
- Learn C++ to become a better software developer
  - i.e., become more effective at designing & implementing modular & robust programs
- Learn & apply software patterns & idioms
  - C++ supports many different programming styles
- Learn C++ gradually
  - Don't have to know every detail of C++ to write a good C++ program



Overview of C++: Part 2

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

Stack Example

- The rest of this overview examines several alterative methods of implementing a Stack
  - We'll begin with C & evolve up to various C++ implementations
- First, consider the "bare-bones" C implementation:

```
typedef int T;
#define MAX_STACK 100 /* const int MAX_STACK = 100; */
T stack[MAX_STACK];
int top = 0;
T item = 10;
stack[top++] = item; // push
...
item = stack[--top]; // pop
```

- Obviously, this solution is not very abstract...

Data Hiding Implementation in C

- Define the interface to a Stack of integers in C in Stack.h:

```
/* Type of Stack element. */
typedef int T;

/* Stack interface. */
int create (int size);
int destroy (void);
void push (T new_item);
void pop (T *old_top);
void top (T *cur_top);
int is_empty (void);
int is_full (void);
```

Data Hiding Implementation in C

- /* File stack.c */

```
#include "stack.h"
static int top_, size_; /* Hidden within this file. */
static T *stack_;

int create (int size) {
  top_ = 0; size_ = size;
  stack_ = malloc (size * sizeof (T));
  return stack_ == 0 ? -1 : 0;
}
void destroy (void) { free ((void *) stack_); }
void push (T item) { stack_[top_++] = item;}
void pop (T *item) { *item = stack_[--top_]; }
void top (T *item) { *item = stack_[top_ - 1]; }
int is_empty (void) { return top_ == 0; }
int is_full (void) { return top_ == size_; }
```

Data Hiding Implementation in C

- Use case

```
#include "stack.h"
void foo (void) {
  T i;
  push (10); /* Oops, forgot to call create! */
  push (20);
  pop (&i);
  destroy ();
}
```

- Main problems:
  - The programmer must call create() first & destroy() last!
  - There is only one stack & only one type of stack
  - Name space pollution...
  - Non-reentrant

Data Abstraction Implementation in C

- An ADT Stack interface in C:

```
typedef int T;
typedef struct { size_t top_, size_; T *stack_; } Stack;

int Stack_create (Stack *s, size_t size);
void Stack_destroy (Stack *s);
void Stack_push (Stack *s, T item);
void Stack_pop (Stack *, T *item);

/* Must call before pop'ing */
int Stack_is_empty (Stack *);
/* Must call before push'ing */
int Stack_is_full (Stack *);
/* ... */
```

Data Abstraction Implementation in C

- An ADT Stack implementation in C:

```
#include "stack.h"
int Stack_create (Stack *s, size_t size) {
  s->top_ = 0; s->size_ = size;
  s->stack_ = malloc (size * sizeof (T));
  return s->stack_ == 0 ? -1 : 0;
}
void Stack_destroy (Stack*s) { free ((void *)s->stack_);}

void Stack_push (Stack *s, T item)
{ s->stack_[s->top_++] = item; }
void Stack_pop (Stack *s, T *item)
{ *item = s->stack_[--s->top_]; }

int Stack_is_empty (Stack *s) { return s->top_ == 0; }
```

Data Abstraction Implementation in C

- Use case

```
void foo (void) {
  Stack s1, s2, s3; /* Multiple stacks! */
  T item;
  Stack_pop (&s2, &item); /* Pop'd empty stack */

  /* Forgot to call Stack_create! */
  Stack_push (&s3, 10);

  s2 = s3; /* Disaster due to aliasing!!! */

  /* Destroy uninitialized stacks! */
  Stack_destroy (&s1); Stack_destroy (&s2);
}
```

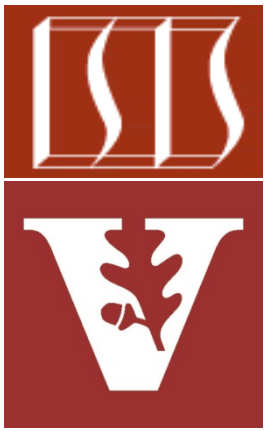
Data Abstraction Implementation in C

- No guaranteed initialization, termination, or assignment
- Still only one type of stack supported
- Too much overhead due to function calls
- No generalized error handling...
- The C compiler does not enforce information hiding, e.g.,

```
s1.top_ = s2.stack_[0]; /* Violate abstraction */
s2.size_ = s3.top_; /* Violate abstraction */
```

Overview of C++: Part 3

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

Data Abstraction Implementation in C++

- We can get encapsulation & more than one stack:

```
typedef int T;

class Stack {
public:
  Stack (size_t size);
  Stack (const Stack &s);
  void operator= (const Stack &);
  ~Stack (void);
  void push (const T &item);
  void pop (T &item);
  bool is_empty (void) const;
  bool is_full (void) const;

private:
  size_t top_, size_; T *stack_;
};
```

Data Abstraction Implementation in C++

- Manager operations

```
Stack::Stack (size_t s)
  : top_ (0), size_ (s), stack_ (new T[s]) {}

Stack::Stack (const Stack &s)
  : top_(s.top_), size_(s.size_),stack_(new T[s.size_])
{
  for (size_t i = 0; I < s.size_; ++i)
    stack_[i] = s.stack_[i];
}
```

Data Abstraction Implementation in C++

- Manager operations

```
void Stack::operator= (const Stack &s)
{
  if (this == &s)
    return;
  T *temp_stack = new T[s.size_];
  delete [] stack_; stack_ = 0;
  for (size_t i = 0; i < s.size_; ++i)
    temp_stack[i] = s.stack_[i];
  stack_ = temp_stack; top_ = s.top_; size_ = s.size_;
}

Stack::~Stack (void)
{
  delete [] stack_;
}
```

Data Abstraction Implementation in C++

- Accessor & worker operations

```
bool Stack::is_empty (void) const {
  return top_ == 0;
}

bool Stack::is_full (void) const {
  return top_ == size_;
}

void Stack::push (const T &item) {
  stack_[top_++] = item;
}

void Stack::pop (T &item) {
  item = stack_[--top_];
}
```

Data Abstraction Implementation in C++

- Use case

```
#include "Stack.h"
void foo (void) {
  Stack s1 (1), s2 (100);
  T item;
  if (!s1.is_full ())
    s1.push (473);
  if (!s2.is_full ())
    s2.push (2112);
  if (!s2.is_empty ())
    s2.pop (item);
  s2.top_ = 10; // Access problem caught at compile-time!
  s2 = s1; // No aliasing problem with assignment
  // Termination is handled automatically.
}
```

Pros of Data Abstraction in C++

- Data hiding & data abstraction, e.g.,

```
Stack s1 (200);
s1.top_ = 10 // Error flagged by compiler!
```

- The ability to declare multiple stack objects

```
Stack s1 (10), s2 (20), s3 (30);
```

- Automatic initialization & termination

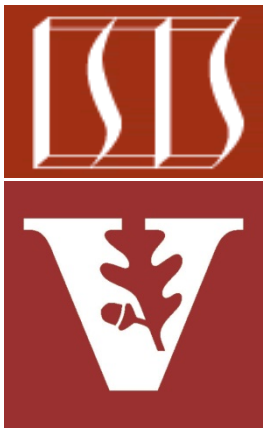
```
{
  Stack s1 (1000); // constructor called automatically.
  // ...
  // Destructor called automatically
}
```

Cons of Data Abstraction in C++

- Error handling is obtrusive
  - Use exception handling to solve this (but be careful)!
- The example is limited to a single type of stack element (int in this case)
  - We can use C++ “parameterized types” to remove this limitation
- Function call overhead
  - We can use C++ inline functions to remove this overhead

Overview of C++: Part 4

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

Exception Handling Implementation in C++

- C++ exceptions separate error handling from normal processing

```
typedef .... T; // Where "...." is a placeholder
class Stack {
public:
  class Underflow { /* ... */ };
  class Overflow { /* ... */ };
  Stack (size_t size);
  Stack (const Stack &rhs);
  void operator= (const Stack &rhs);
  ~Stack (void);
  void push (const T &item) throw (Overflow);
  void pop (T &item) throw (Underflow);
  // ...
private:
  size_t top_, size_; T *stack_;
};
```

Exception Handling Implementation in C++

- Stack.cpp

```
Stack::Stack (size_t s)
  : top_ (s), size_ (s), stack_ (new T[s]) {}

Stack::~Stack () { delete [] stack_; }

void Stack::push (const T &item) throw (Stack::Overflow) {
  if (is_full ()) throw Stack::Overflow ();
  stack_[top_++] = item;
}

void Stack::pop (T &item) throw (Stack::Underflow) {
  if (is_empty ()) throw Stack::Underflow ();
  item = stack_[--top_];
}
```

Exception Handling Implementation in C++

- Stack.cpp

```
Stack::Stack (const Stack &s):
  : top_ (s.top_), size_ (s.size_), stack_ (0)
{
  scoped_array<T> temp_stack (new T[s.size_]);

  for (size_t i = 0; i < s.size_; ++i)
    temp_stack[i] = s.stack_[i];

  temp_stack.swap (stack_);
}
```

Exception Handling Implementation in C++

- Stack.cpp

```
void Stack::operator= (const Stack &s)
{
  if (this == &s) return; // Check for self-assignment
  scoped_array<T> temp_stack (new T[s.size_]);

  for (size_t i = 0; i < s.size_; ++i)
    temp_stack[i] = s.stack_[i];

  top_ = s.top_; size_ = s.size_;
  temp_stack.swap (stack_);
}
```

Exception Handling Implementation in C++

- scoped_array extends auto_ptr to destroy built-in arrays

```
template <typename T> class scoped_array {
public:
  explicit scoped_array (T *p = 0) : ptr_ (p) {}
  ~scoped_array () { delete [] ptr_; }
  T &operator[](std::ptrdiff_t i) const {return ptr_[i];}
  T *get() const { return ptr_; }
  void swap (T *&b) { T *tmp = b; b = ptr_; ptr_ = tmp; }
  void swap (scoped_array<T> &b)
  { T *tmp=b.ptr_; b.ptr_=this->ptr_; this->ptr_=tmp; }

private:
  T *ptr_;
  scoped_array (const scoped_array<T> &);
  scoped_array &operator=(const scoped_array<T> &);
};
```

Exception Handling Implementation in C++

- There's a better way to implement operator=():

```
void Stack::operator= (const Stack &s) {
  if (this == &s) return; // Check for self-assignment
  Stack t (s);
  std::swap (t.top_, top_); std::swap (t.size_, size_);
  std::swap (t.stack_, stack_);
}
```

- The old way:

```
void Stack::operator= (const Stack &s) {
  if (this == &s) return; // Check for self-assignment
  scoped_array<T> temp_stack (new T[s.size_]);
  for (size_t i = 0; i < s.size_; ++i)
    temp_stack[i] = s.stack_[i];
  top_ = s.top_; size_ = s.size_;
  temp_stack.swap (stack_);
}
```

Exception Handling Implementation in C++

- & yet an even better way to implement an exception-safe Stack:

```
class Stack { // ...
private: // ...
  scoped_array<T> stack_;
  void swap (Stack &);
};

// ...
Stack::Stack (const Stack &s)
: top_(s.top_), size_(s.size_), stack_(new T[s.size_]) {
   for (size_t i = 0; i < s.size_; ++i)
     stack_[i] = s.stack_[i];
}

Stack::~Stack () { /* no-op! */ }
```

Exception Handling Implementation in C++

- & yet an even better way to implement operator=():

```
void Stack::operator= (const Stack &s) {
  if (this == &s) return; // Check for self-assignment
  Stack temp_stack (s);
  swap (temp_stack);
}

void Stack::swap (Stack &t) {
  std::swap (t.top_, top_);
  std::swap (t.size_, size_);
  t.stack_.swap (stack_);
}
```

- This solution is easy to generalize!

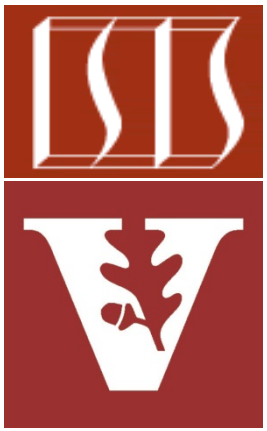
Exception Handling Implementation in C++

- Use case

```
#include "Stack.h"
void foo (void) {
  Stack s1 (1), s2 (100);
  try {
    T item;
    s1.push (473);
    s1.push (42); // Exception, push'd full stack!
    s2.pop (item); // Exception, pop'd empty stack!
    s2.top_ = 10; // Access violation caught!
  } catch (Stack::Underflow)
  { /* Handle underflow... */ }
  catch (Stack::Overflow) { /* Handle overflow... */ }
  catch (...) { /* Catch anything else... */ throw; }
  } // Termination is handled automatically.
}
```

Overview of C++: Part 5

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

Template Implementation in C++

- A parameterized type Stack class interface using C++

```
template <typename T> class Stack {
public:
  Stack (size_t size);
  Stack (const Stack<T> &rhs);
  void operator= (const Stack<T> &rhs);
  ~Stack (void)
  void push (const T &item);
  void pop (T &item);
  bool is_empty (void) const;
  bool is_full (void) const;

private:
  size_t top_, size_; scoped_array<T> stack_;
  void swap (scoped_array<T> &t);
};
```

Template Implementation in C++

- A parameterized type Stack class implementation using C++

```
template <typename T> inline
Stack<T>::Stack (size_t size)
: top_ (0), size_ (size), stack_ (new T[size]) { }

template <typename T> inline
Stack<T>::~Stack (void) { /* no-op! */ }

template <typename T> inline void
Stack<T>::push (const T &item) {
  stack_[top_++] = item;
}

template <typename T> inline void
Stack<T>::pop (T &item) {
  item = stack_[--top_];
}
```

Template Implementation in C++

- Note minor changes to accommodate parameterized types

```
#include "Stack.h"
void foo (void) {
  Stack<int> s1 (1000);
  Stack<float> s2;
  Stack< Stack <Activation_Record> *> s3;

  s1.push (-291);
  s2.top_ = 3.1416; // Access violation caught!
  s3.push (new Stack<Activation_Record>);
  Stack <Activation_Record> *sar;
  s3.pop (sar);

  delete sar;
  // Termination of s1, s2, & s3 handled automatically
}
```

Template Implementation in C++

- Another parameterized type Stack class

```
template <typename T, size_t SIZE> class Stack {
public:
  Stack (void);
  ~Stack (void)
  void push (const T &item);
  void pop (T &item);
private:
  size_t top_, size_;
  T stack_[SIZE];
};
```

- No need for dynamic memory, though SIZE must be const, e.g.,

```
Stack<int, 200> s1;
```