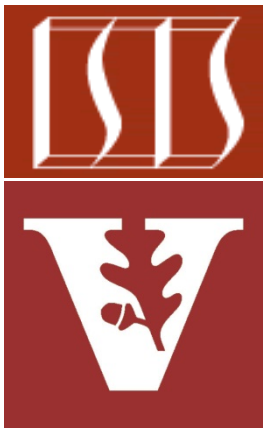
Overview of C++: Part 6

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

Object-Oriented Implementation in C++

- Problems with previous examples:
  - Changes to the implementation will require recompilation & relinking of clients
  - Extensions will require access to the source code
- Solutions
  - Combine inheritance with dynamic binding to completely decouple interface from implementation & binding time
  - This requires the use of C++ abstract base classes

Object-Oriented Implementation in C++

- Defining an abstract base class in C++

```
template <typename T>
class Stack {
public:
  virtual ~Stack (void) = 0; // Need implementation!
  virtual void push (const T &item) = 0;
  virtual void pop (void) = 0;
  virtual void top (T &item) const = 0;
  virtual bool is_empty (void) const = 0;
  virtual bool is_full (void) const = 0;
};
```

- Using pure virtual methods ensures that the C++ compiler won't instantiate objects of the abstract base class

Object-Oriented Implementation in C++

- It's entirely possible (& often desirable) to implement a non-virtual method in an abstract base class, e.g.:

```
template <typename T>
class Stack {
public:
  void top (T &item) {
    // Apply the Template Method pattern
    pop (item);
    push (item);
  }
}
```

- Subclasses of Stack *must* implement pop() & push()
- Note that this implementation is not exception-safe & isn't const!

Object-Oriented Implementation in C++

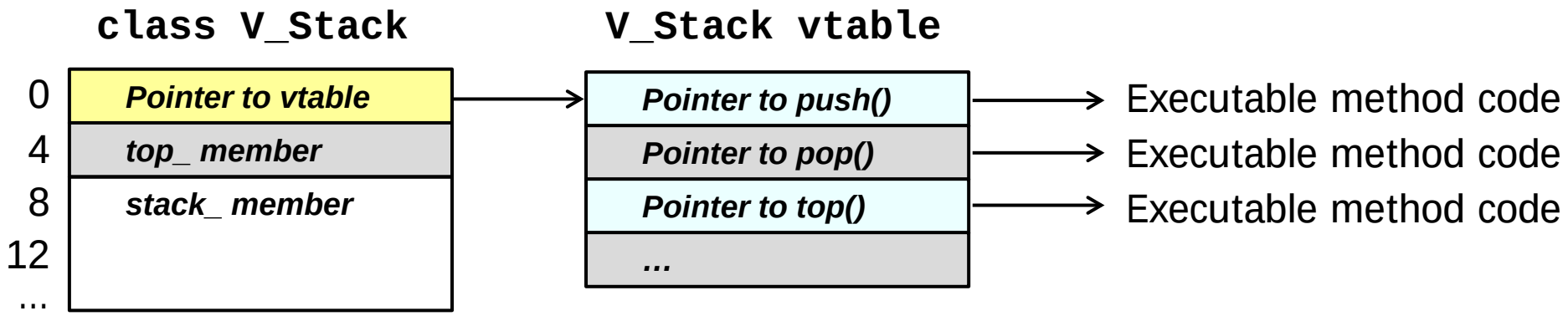
- Deriving from the Stack abstract base class to define a V_Stack subclass

```
#include "Stack.h"

template <typename T> class V_Stack : public Stack<T> {
public:
  enum { DEFAULT_SIZE = 100 };
  V_Stack (size_t size = DEFAULT_SIZE);
  V_Stack (const V_Stack &rhs);
  virtual void push (const T &item);
  virtual void pop (void);
  virtual void top (T &item) const;
  virtual void is_empty (void) const;
  virtual void is_full (void) const;
private:
  size_t top_;
  std::vector<T> stack_;
};
```

Object-Oriented Implementation in C++

- Note that each class with virtual methods has a so-called “vtable”



Object-Oriented Implementation in C++

- class V_Stack implementation

```
template <typename T>
V_Stack<T>::V_Stack (size_t size)
  : top_ (0), stack_ (size) {}

template <typename T>
V_Stack<T>::V_Stack (const V_Stack &rhs)
  : top_ (rhs.top_), stack_ (rhs.stack_) {}
```

Object-Oriented Implementation in C++

- class V_Stack implementation

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

template <typename T> void
V_Stack<T>::pop (void) { --top_; }

template <typename T> void
V_Stack<T>::top (T &item) const { item = stack_[top_-1];
}

template <typename T> int
V_Stack<T>::is_full (void) const
{ return top_ >= stack_.size (); }

template <typename T> int
V_Stack<T>::is_empty (void) const { return top_ == 0; }
```

Object-Oriented Implementation in C++

- Inheritance can also create an linked list stack (L_Stack):

```
template <typename T> class Node; // forward declaration.

template <typename T> class L_Stack : public Stack<T> {
public:
  L_Stack (size_t size_hint = 0);
  virtual ~L_Stack (void);
  virtual void push (const T &new_item);
  virtual void pop (void);
  virtual void top (T &item) const;
  virtual bool is_empty (void) const;
  virtual bool is_full (void) const;

private:
  Node<T> *head_; // Head of linked list of Node<T>'s
};
```

Object-Oriented Implementation in C++

- class Node implementation

```
template <typename T> class Node
{
  friend template <typename T> class L_Stack;
public:
  Node (T i, Node<T> *n = 0): item_ (i), next_ (n) {}

private:
  T item_;
  Node<T> *next_;
};
```

- Note that the use of the “Cheshire cat” idiom allows the library writer to completely hide the representation of class V Stack...

Object-Oriented Implementation in C++

- class L_Stack implementation:

```
template <typename T> L_Stack<T>::L_Stack (size_t)
  : head_ (0) {}

template <typename T> void L_Stack<T>::push(const T&it) {
  Node<T> *t = new Node<T> (it, head_); head_ = t;
}

template <typename T> void L_Stack<T>::pop () {
  Node<T> *t = head_; head_ = head_->next_; delete t;
}

template <typename T> void L_Stack<T>::top (T &item)
                                                const {
  item = head_->item_;
}
```

Object-Oriented Implementation in C++

- class L_Stack implementation:

```
template <typename T> L_Stack<T>::~L_Stack (void)
{
  while (!is_empty ())
    pop ();
}

template <typename T> L_Stack<T>::is_empty (void) const
{
  return head_ == 0;
}

template <typename T> L_Stack<T>::is_full (void) const
{
  ... // You fill in here ... ;-)
}
```

Object-Oriented Implementation in C++

- Using our abstract base class, it is possible to write code that does not depend on the stack implementation, e.g.,

```
Stack<int> *make_stack (bool v_Stack) {
  return v_Stack ? new V_Stack<int> : new L_Stack<int>;
}

void print_top (Stack<int> *sp) {
  std::cout << "top = "
            << sp->top () // (*sp->vptr[3])(sp);
            << std::endl;
}

int main (int argc, char **) {
  std::auto_ptr <Stack<int>> sp (make_stack (argc > 1));
  sp->push (10);
  print_top (sp.get ());
}
```

Object-Oriented Implementation in C++

- Moreover, we can make changes at run-time without modifying, recompiling, or relinking existing code via dynamic linking:

```
char stack_symbol[MAXNAMLEN];
char stack_file[MAXNAMLEN];
cin >> stack_file >> stack_factory;

void *handle = ACE_OS::dlopen (stack_file);
Stack<int> *(*factory)(bool) =
  ACE_OS::dlsym (handle, stack_factory);

std::auto_ptr <Stack<int>> sp ((*factory) (argc > 1));
sp->push (10);
print_top (sp.get ());
```

- Note, no need to stop, modify, & restart an executing application!
  - Naturally, this requires careful configuration management...