

C++ Overview

- C++ was designed at AT&T Bell Labs by Bjarne Stroustrup in the early 80's
 - The original *cf*ront translated C++ into C for portability
 - * However, this was difficult to debug and potentially inefficient
 - Many native host machine compilers now exist
 - * e.g., Borland, DEC, GNU, HP, IBM, Microsoft, Sun, Symantec, etc.
- C++ is a *mostly* upwardly compatible extension of C that provides:
 1. *Stronger typechecking*
 2. *Support for data abstraction*
 3. *Support for object-oriented programming*
 4. *Support for generic programming*

An Overview of C++

Douglas C. Schmidt

Professor

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt/

Department of EECS

Vanderbilt University

(615) 343-8197



December 11, 2006

C++ Design Goals (cont'd)

- “As close to C as possible, but no closer”
 - *i.e.*, C++ is not a proper superset of C → backwards compatibility is not entirely maintained
 - * Typically not a problem in practice...
- Note, certain C++ design goals conflict with modern techniques for:
 1. *Compiler optimization*
 - *e.g.*, pointers to arbitrary memory locations complicate register allocation & garbage collection
 2. *Software engineering*
 - *e.g.*, separate compilation complicates inlining due to difficulty of interprocedural analysis
 - Dynamic memory management is error-prone

C++ Design Goals

- As with C, run-time efficiency is important
 - Unlike other languages (*e.g.*, Ada) complicated run-time libraries have not traditionally been required for C++
 - * Note, that there is no language-specific support for concurrency, persistence, or distribution in C++
- Compatibility with C libraries & traditional development tools is emphasized, *e.g.*,
 - Object code reuse
 - * The storage layout of structures is compatible with C
 - * *e.g.*, support for X-windows, standard ANSI C library, & UNIX/WIN32 system calls via `extern` block
 - C++ works with the `make` recompilation utility

Important Minor Enhancements

- C++ enforces type checking via *function prototypes*
- Provides type-safe linkage
- Provides inline function expansion
- Declare constants that can be used to define static array bounds with the `const` type qualifier
- Built-in dynamic memory management via `new` & `delete` operators
- Namespace control

Major C++ Enhancements

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

Questionable Enhancements

- Default values for function parameters
- Operator & function overloading
- Variable declarations may occur anywhere statements may appear within a block
- Allows user-defined conversion operators
- Static data initializers may be arbitrary expressions

Useful Minor Enhancements

- The name of a `struct`, `class`, `enum`, or `union` is a type name
- References allow “call-by-reference” parameter modes
- New type-secure extensible *iostreams* I/O mechanism
- “Function call”-style cast notation
- Several different commenting styles
- New `mutable` type qualifier
- New `bool` boolean type

Strategies for Learning C++

- Focus on concepts & programming techniques
 - Don't get lost in language features
- Learn C++ to become a better programmer
 - More effective at designing & implementing
 - Design Patterns
- C++ supports many different programming styles
- Learn C++ gradually
 - Don't have to know every detail of C++ to write a good C++ program

Language Features Not Part of C++

1. *Concurrency*
 - “Concurrent C” by Gehani
 - Actor++ model by Lavender & Kafura
2. *Persistence*
 - Object Store, Versant, Objectivity
 - Exodus system & E programming language
3. *Garbage Collection*
 - USENIX C++ 1994 paper by Ellis & Detlefs
 - GNU g++
4. *Distribution*
 - CORBA, RMI, COM+, SOAP, etc.

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);
```

Stack Example

- The following slides examine several alternative methods of implementing a Stack
 - Begin with C & evolve up to various C++ implementations
- First, consider the “bare-bones” 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 not very abstract...

Data Hiding Implementation in C (cont'd)

- Use case

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

- Main problems:

1. The programmer must call `create()` first & `destroy()` last!
2. There is only *one* stack & only *one* type of stack
3. Name space pollution...
4. Non-reentrant

Data Hiding Implementation in C (cont'd)

- `/* 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 Abstraction Implementation in C (cont'd)

- 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_);
    s->top_ = 0; s->size_ = 0; s->stack_ = 0;
}
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

- 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 *);
/* ... */
```


Main problems with Data Abstraction in C

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

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

Data Abstraction Implementation in C (cont'd)

- 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++ (cont'd)

- 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];
    }
```

```
void Stack::operator = (const Stack &s) {
    if (this == &s) return;
    T *temp_stack = new T[s.size_]; delete [] stack_;
    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++

- We can get encapsulation *and* 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);
    int is_empty (void) const;
    int is_full (void) const;
private:
    size_t top_, size_;
    T *stack_;
};
```

Data Abstraction Implementation in C++ (cont'd)

- 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);
    // Access violation caught at compile-time!
    s2.top_ = 10;
    // Termination is handled automatically.
}
```

Data Abstraction Implementation in C++ (cont'd)

- Accessor & worker operations

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

int 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_]; }
```

Drawbacks with C++ Data Abstraction Implementation

1. Error handling is obtrusive
 - Use exception handling to solve this (but be careful)!
2. 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
3. Function call overhead
 - We can use C++ inline functions to remove this overhead

Benefits of C++ Data Abstraction Implementation

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

```
Stack s1 (200);  
s1.top_ = 10 // Error flagged by compiler!
```
2. The ability to declare multiple stack objects

```
Stack s1 (10), s2 (20), s3 (30);
```
3. Automatic initialization & termination

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

Exception Handling Implementation in C++ (cont'd)

- Stack.cpp

```
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++ (cont'd)

- C++ exceptions separate error handling from normal processing

```
typedef int T;
class Stack {
public:
    class Underflow { /* ... */ };
    class Overflow { /* ... */ };
    Stack (size_t size);
    ~Stack (void);
    void push (const T &item) throw (Overflow);
    void pop (T &item) throw (Underflow);
    int is_empty (void);
    int is_full (void);
private:
    size_t top_, size_;
    T *stack_;
};
```

Exception Handling Implementation in C++ (cont'd)

- `scoped_array` extends `auto_ptr` to arrays
- Deletion of array is guaranteed on destruction of `scoped_array`
- This implementation is based on Boost `scoped_array` class

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

Exception Handling Implementation in C++ (cont'd)

- `Stack.cpp`

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

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

Template Implementation in C++

- A parameterized type Stack class interface using C++

```
template <typename T> class Stack {
public:
    Stack (size_t size);
    ~Stack (void)
    void push (const T &item);
    void pop (T &item);
    int is_empty (void);
    int is_full (void);
private:
    size_t top_, size_;
    T *stack_;
};
```
- To simplify the following examples we'll omit exception handling, but note that it's important to ensure exception-safety guarantees!

Exception Handling Implementation in C++ (cont'd)

- 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.
}
```

Template Implementation in C++ (cont'd)

- 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 is handled automatically
}
```

Template Implementation in C++ (cont'd)

- 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) { delete [] stack_; }

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_]; }
```


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*

Template Implementation in C++ (cont'd)

- 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];
};
```
- Note, there's no longer any need for dynamic memory, though SIZE must be a constant, e.g.,

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

Object-Oriented Implementation in C++ (cont'd)

- Inherit to create a specialized “bounded” stack:

```
#include "Stack.h"
#include "vector"

template <typename T> class B_Stack : public Stack<T> {
public:
    enum { DEFAULT_SIZE = 100 };
    B_Stack (size_t size = DEFAULT_SIZE);
    virtual void push (const T &item);
    virtual void pop (T &item);
    virtual int is_empty (void) const;
    virtual int is_full (void) const;
private:
    size_t top_; // built-in
    std::vector<T> stack_; // user-defined
};
```

Object-Oriented Implementation in C++ (cont'd)

- Defining an abstract base class in C++

```
template <typename T>
class Stack {
public:
    virtual void push (const T &item) = 0;
    virtual void pop (T &item) = 0;
    virtual int is_empty (void) const = 0;
    virtual int is_full (void) const = 0;
    void top (T &item) { // Template Method
        pop (item); push (item);
    }
};
```

- By using “pure virtual methods,” we can guarantee that the compiler won’t allow instantiation!

Object-Oriented Implementation in C++ (cont'd)

- Inheritance can also create an “unbounded” stack:

```
template <typename T> class Node; // forward declaration.
template <typename T> class UB_Stack : public Stack<T> {
public:
    enum { DEFAULT_SIZE = 100 };
    UB_Stack (size_t hint = DEFAULT_SIZE);
    ~UB_Stack (void);
    virtual void push (const T &new_item);
    virtual void pop (T &top_item);
    virtual int is_empty (void) const { return head_ == 0; }
    virtual int is_full (void) const { return 0; }
private:
    // Head of linked list of Node<T>'s.
    Node<T> *head_;
};
```

Object-Oriented Implementation in C++ (cont'd)

- class B_Stack implementation

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

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

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

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

Object-Oriented Implementation in C++ (cont'd)

- class UB_Stack implementation:

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

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

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

template <typename T> UB_Stack<T>::~~UB_Stack (void)
{ for (T t; head_ != 0; pop (t)) continue; }
```

Object-Oriented Implementation in C++ (cont'd)

- class Node implementation

```
template <typename T> class Node {
    friend template <typename T> class UB_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 UB_Stack...

Object-Oriented Implementation in C++ (cont'd)

- Moreover, we can make changes at run-time without modifying, recompiling, or relinking existing code
 - *i.e.*, can use “dynamic linking” to select stack representation at run-time, *e.g.*,


```
char stack_symbol[MAXNAMLEN];
char stack_file[MAXNAMLEN];
cin >> stack_file >> stack_symbol;
void *handle = ACE_OS::dlopen (stack_file);
void *sym = ACE_OS::dlsym (handle, stack_symbol);
if (Stack<int> *sp = // Note use of RTTI
    dynamic_cast <Stack<int> *> (sym)) foo (sp);
```
- Note, no need to stop, modify, & restart an executing application!
 - Naturally, this requires careful configuration management...

Object-Oriented Implementation in C++ (cont'd)

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


```
template <typename T> Stack<T> *make_stack (int use_B_Stack) {
    if (use_B_Stack) return new B_Stack<T>;
    else return new UB_Stack<T>;
}

void foo (Stack<int> *stack) {
    int i;
    stack->push (100);
    stack->pop (i);
    // ...
}

foo (make_stack<int> (0));
```

Summary

- A major contribution of C++ is its support for defining abstract data types (ADTs) & for generic programming
 - *e.g., classes, parameterized types, & exception handling*
- For some systems, C++'s ADT support is more important than using the OO features of the language
- For other systems, the use of C++'s OO features is essential to build highly flexible & extensible software
 - *e.g., inheritance, dynamic binding, & RTTI*