The C++ Programming Language

An Overview of C++

Outline

C++ Overview

- C++ was designed at AT&T Bell Labs by Bjarne Stroustrup in the early 80's
  - The original *cfront* 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*
   - C++ *supports* the Object-Oriented paradigm but does not *require* it

C++ Design Goals

- As with C, run-time efficiency is important
  - *e.g.*, unlike 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 and UNIX tools is emphasized, *e.g.*,
  - Object code reuse
    - ▷ The storage layout of structures is compatible with C
    - ▷ Support for X-windows, standard ANSI C library, UNIX system calls via **extern** block
  - C++ works with the `make` recompilation utility

C++ Design Goals (cont'd)

- "As close to C as possible, but no closer"
  - *i.e.*, C++ is not a proper superset of C, so that 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 and garbage collection
2. *Software engineering*
   - *e.g.*, separate compilation complicates inlining due to difficulty of interprocedural analysis

Major C++ Enhancements

1. C++ supports object-oriented programming features
   - *e.g.*, single and multiple inheritance, abstract base classes, and virtual functions
2. C++ facilitates data abstraction and encapsulation that hides representations behind abstract interfaces
   - *e.g.*, the class mechanism and parameterized types
3. C++ provides enhanced error handling capabilities
   - *e.g.*, exception handling
4. C++ provides a means for identifying an object's type at runtime
   - *e.g.*, Run-Time Type Identification (RTTI)

Other Enhancements

- C++ enforces type checking via *function prototypes*
- Allows several different commenting styles
- Provides type-safe linkage
- Provides **inline** function expansion
- Built-in dynamic memory management via **new** and **delete** operators
- Default values for function parameters
- Operator and function overloading

Other Enhancements (cont'd)

- References provide "call-by-reference" parameter passing
- Declare constants with the **const** type qualifier
- New **mutable** type qualifier
- New **bool** boolean type
- New type-secure extensible I/O interface called *streams* and *iostreams*

Other Enhancements (cont'd)

- A new set of "function call"-style cast notations
- Variable declarations may occur anywhere statements may appear within a block
- The name of a **struct**, **class**, **enum**, or **union** is a type name
- Allows user-defined conversion operators
- Static data initializers may be arbitrary expressions
- C++ provides a namespace control mechanism for restricting the scope of classes, functions, and global objects

Language Features Not Part of C++

1. *Concurrency*
   - See "Concurrent C" by Nehrain Gehani
2. *Persistence*
   - See Exodus system and E programming language
3. *Garbage Collection*
   - See papers in USENIX C++ 1994

Stack Example

- The following slides will examine several alteratives methods of implementing a Stack abstraction
  - Begin with C and evolve up to C++
- First, consider the "bare-bones" implementation

```
typedef int T;
const int MAX_STACK = 100;
T stack[MAX_STACK];
int top = 0;

T item;

stack[top++] = item; // push
...
item = stack[--top]; // pop
```

- Obviously not very abstract... ;-)

Stack Example in C

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

```
/* file stack.h */
typedef int T;

bool create (int size);
bool dispose (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 in C (cont'd)

- /* file stack.c */

```
#include <stdlib.h>
#include "stack.h"
static int top_, size_;
static T *stack_;
bool create (int sz) {
      top_ = 0; size_ = sz;
      return (stack_ =
            malloc (size_ * sizeof *stack_)) != 0;
}
bool dispose (void) { free ((void *) stack_); return true; }
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_; }
```

Stack Example in C (cont'd)

- /* Main program */

```
#include "stack.h"
int main (void) {
      /* Oops, forgot to call create! */
      T i;
      push (10);
      push (20);
      push (30);
      pop (&i);
      dispose ();
}
```

- Main problems:

1. The programmer must remember to call `create` before any other operations and `dispose` afterwards!
2. There is only *one* stack (*i.e.*, not generalizable)
3. Name space pollution...
4. Lack of synchronization (*i.e.*, non-reentrant)

Stack Example in C (cont'd)

- An ADT Stack in C:

```
/* file stack.h */
typedef int T;
typedef struct Stack {
      int top_, size_;
      T *stack_;
} STACK;
int stack_create (STACK *s, int sz);
void push (STACK *s, T item);
/* ...*/

/* File stack.c */
#include <stdlib.h>
/* Note, no need for static/global variables! */
int stack_create (STACK *s, int len) {
      s->top_ = 0; s->size_ = len;
      return (s->stack_ =
            malloc (s->size_ * sizeof *s->stack_)) != 0;
}
void push (STACK *s, T item) {
      s->stack_[s->top_++] = item;
}
```


Stack Example in C (cont'd)

- Main problems with the C approach

1. Only one type supported
2. The compiler does not enforce information hiding
3. Too much overhead due to function calls
4. No guaranteed initialization/termination, *e.g.*,

```
STACK s1, s2, s3; /* Multiple stacks! */
s1.top_ = s2.stack_[0]; /* violate abstraction */
s2.size_ = s3.top_; /* violate abstraction */
push (&s3, 10); /* forgot to call stack_create! */
stack_destroy (&s1); /* yikes! */
```

- Question: how to we get both abstraction *and* more than one stack?!?!

Stack Example in C++

- A Stack class interface using C++

```
// File Stack.h
typedef int T;
class Stack {
public:
      Stack (size_t len = 100);
      ~Stack (void)
      void push (T item);
      void pop (T &item);
      // ...
private:
      size_t top_, size_;
      T *stack_;
};
```

Stack Example in C++ (cont'd)

- A Stack class implementation using C++

```
Stack::Stack (size_t len)
      : top_ (0), size_ (len) {
      this->stack_ = new T[len];
}
Stack::~Stack (void) {
      delete [] this->stack_;
}
void Stack::push (T item) {
      this->stack_[this->top_++] = item;
}
void Stack::pop (T &item) {
      item = this->stack_[--this->top_];
}
// ...
#include "Stack.h"
int main (void) {
      Stack s1, s2 (10);
      s1.push (−291);
      s2.top_ = 10; /* Access violation caught! */
      /* Termination is handled automatically */
}
```


Stack Example in C++ (cont'd)

- The C++ version of a stack provides:

1. Data hiding and data abstraction
2. The ability to declare multiple stack objects
3. Automatic initialization and termination

- However, there are some limitations

1. 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
2. Function call overhead
   - We can use C++ inline functions to remove this overhead

Stack Example in C++ (cont'd)

- A parameterized type Stack class interface using C++

```
template <class T>
class Stack {
public:
      Stack (size_t len = 100);
      ~Stack (void)
      void push (T item);
      void pop (T &item);
      // ...
private:
      size_t top_, size_;
      T *stack_;
};
```


Stack Example in C++ (cont'd)

- A parameterized type Stack class implementation using C++

```
template <class T> inline
Stack<T>::Stack (size_t len)
      : top_ (0), size_ (len) {
      this->stack_ = new T[len];
}

template <class T> inline
Stack<T>::~Stack (void) {
      delete [] this->stack_;
}

template <class T> inline
void Stack<T>::push (T item) {
      this->stack_[this->top_++] = item;
}

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

Stack Example in C++ (cont'd)

- Note the minor changes to the code to accommodate parameterized types

```
#include "Stack.h"

int main (void)
{
      Stack<int> s1 (1000);
      Stack<float> s3;
      Stack< Stack <Employee> > s3;
      s1.push (−291);
      s2.top_ = 3.1416; /* Access violation caught! */
      /* Termination is handled automatically */
}
```


Stack Example in C++ (cont'd)

- The previous example has the drawback that changes to the implementation will require recompilation and relinking of client applications
- By combining inheritance with dynamic binding we can *completely* decouple the interface of the Stack from the implementation and the binding time
- *e.g.*,

```
template <class T>
class Stack
{
public:
      virtual void push (T item) = 0;
      virtual void pop (T &item) = 0;
      virtual void top (T &item) = 0;
      virtual bool is_empty (void) = 0;
      virtual bool is_full (void) = 0;
};
```


Stack Example in C++ (cont'd)

- We can now use inheritance to create a specialized version of a stack:

```
#include "Stack.h"
#include "Vector.h"
template <class T>
class Bounded_Stack : public Stack<T>
{
public:
      Bounded_Stack (size_t len = 100)
            : top_ (0), stack_ (len) {}
      virtual void push (T item) {
            this->stack_.set (this->top_++, item);
      }
      virtual void pop (T &item) {
            this->stack_.get (--this->top_, item);
      }
      virtual bool is_full (void) {
            return this->top_ >= this->size ();
      }
      // ...
private:
      Vector<T> stack_; // user-defined
      size_t top_; // built-in
};
```


Stack Example in C++ (cont'd)

- Likewise, we can use inheritance to create a totally different implementation of a Stack:

```
template <class T>
class Unbounded_Stack : public Stack<T>
{
public:
      Unbounded_Stack (size_t len = 100) // ignored...
            : head_ (new Node) {}
      ~Unbounded_Stack (void); // delete all Nodes...
      virtual void push (T new_item) {
            this->head_->item_ = new_item;
            this->head_->next_ = new Node (this->head_->next_);
      }
      virtual void pop (T &top_item) {
            top_item = this->head_->item_;
            Node *t = this->head_;
            this->head_ = this->head_->next_;
            delete t;
      }
      virtual bool is_empty (void) {
            return this->head_->next_ == 0;
      }
      // ...
private:
      struct Node {
            Node (Node *n = 0, T i = 0): item_ (i), next_ (n) {}
            T item_;
            Node *next_;
      } *head_;
};
```

Stack Example in C++ (cont'd)

- Using our abstract base class, it is possible to write code that does not depend on the stack implementation
- Moreover, we can make changes at run-time without modifying, recompiling, or relinking existing code
  - *e.g.*,

```
// Does not require recompilation
int foo (Stack<int> *stack) {
      int i;
      stack->push (100);
      stack->pop (i);
}

// May require recompilation
int bar (bool use_bounded_stack) {
      Stack<int> *sp;
      if (use_bounded_stack)
            sp = new Bounded_Stack<int>;
      else
            sp = new Unbounded_Stack<int>;
      foo (sp);
}
```


Stack Example in C++ (cont'd)

- In addition, we can use "dynamic linking" to make the selection of the stack representation at run-time, *e.g.*,

```
char stack_symbol[maxnamlen];
char stack_file[maxnamlen];
cin >> stack_file >> stack_symbol;
void *handle = ::dlopen (stack_file);
Stack<int> *sp =
      (Stack<int> *) ::dlsym (handle, stack_symbol);
foo (sp);
```

- In this case, we don't even have to stop and restart an executing application in order to alter its functionality!
  - Naturally, this requires careful configuration management...

Summary

- A major contribution of C++ is its support for defining abstract data types (ADTs), *e.g.*,
  - Classes
  - Parameterized types
  - Exception handling
- For many systems, successfully utilizing C++'s ADT support is more important than using the OO features of the language, *e.g.*,
  - Inheritance
  - Dynamic binding