CS446
SOFTWARE ENGINEERING

Richard Bartels

University of Waterloo
Faculty of Mathematics
Department of Computer Science

May 28, 1996

C++

CS446
C++ OVERVIEW

Richard Bartels

University of Waterloo
Faculty of Mathematics
Department of Computer Science

May 28, 1996

Outline

- C++ class and C struct
- Stack/Node Example
- Inheritance
- Polymorphism
- Templates

C++

- Resources
  - Course WWW pages
  - *The C++ Primer, 2nd Edition*, Stanley B. Lippman
  - Warning: `g++` is not really `C++`
- C++: "C with Classes"
  - ANSI C programs compile and run in `C++` (~few changes)
- Class:
  - Expanded `struct`
  - Encapsulates data and functions
  - Access control through `public`, `protected`, `private`
  - Extensible/modifiable through inheritance
  - Constructors and destructor for initialization and cleanup
  - Dynamic binding (polymorphism)
- Also among some other features:
  - Overloading
  - Operator definitions
  - Templates and exceptions

Struct vs. Class: Superficial Overview

- Struct

```
typedef struct complexDef Complex;
struct complexDef
{
  double re, im;
};
extern Complex cAdd( Complex z, Complex w );
```

- Class

```
class Complex
{
  public:
    double re( void );
    double im( void );
    Complex operator+( Complex opnd );
  protected:
  private:
    double theRe, theIm;
};
```

- Remarks
  - many `struct` styles; `class` syntax cleaner than any
  - `cAdd` functionality taken over by `operator+`
  - `re`,`im` access functions hide implementation

C/Struct: Stack.h

```
#ifndef Stack_H
#define Stack_H

typedef struct stackDef Stack;
typedef struct nodeDef Node;

struct nodeDef
{
  Node* next;
  int content;
};

struct stackDef
{
  Node* top;
  int count;
};

/* Procedures go here */

#endif
```

- Remarks
  - `.h` defines interface; `ifndef` wrapper important

Stack Procedures

```
extern Node* nodeCreate( int i );
extern int nodeDestroy( Node* n );
extern void nodePrint( Node* n );

extern void stackInit( Stack* s );
extern int stackIsEmpty( Stack* s );
extern void stackPush( Node* n, Stack* s );
extern Node* stackPop( Stack* s );
extern void stackFlush( Stack* s );
extern void stackPrint( Stack* s );
```

- Remarks

– Minimal functionality for `Node`

Create with contents, destroy, print contents

– Minimal functionality for `Stack`

Initialize as empty, test if empty, push, pop, destroy all nodes, print all nodes

– All access through pointers

design decision: dynamic storage and uniformity

Examples of Use

```
#include <stdio.h>
#include "Stack.h"
#define MAX 5

int main( void )
{
  int i; Stack s; Node* n=(Node*)0;
  stackInit( &s );

  printf( "\n*******\nPUSHING\n*******\n" );
  for( i=0; i<MAX; ++i )
  {
    stackPush( nodeCreate(i), &s );
    printf( "\nFor i = %d\n=========\n", i );
    stackPrint( &s );
  }

  printf( "\n\n*******\nPOPPING\n*******\n\n" );
  while( !stackIsEmpty(&s) )
  { n=stackPop(&s); nodePrint(n); nodeDestroy(n); }

  stackPush( nodeCreate(99), &s ); stackFlush( &s );

  return( 0 );
}
```

Remarks

- At least one use of every procedure
- Declare `Stack` and initialize
- Declare `Node` pointer for future use
- Loop, for successive integers
  - Create `Node` with integer
  - Push onto `Stack`
  - Print current contents of `Stack`
- Loop, until `Stack` empty
  - Pop, print, destroy `Node`
- Push one `Node` and flush `Stack`

Stack.c, Node Implementation

```
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include "Stack.h"

Node* nodeCreate( int i )
{
  Node* temp;
  temp = (Node*)malloc( (unsigned)sizeof(Node) );
  assert( temp );
  temp->next = (Node*)0;
  temp->content = i;
  return( temp );
}

int nodeDestroy( Node* n )
{
  int temp;
  assert( n );
  temp = n->content;
  free( (char*)n );
  return( temp );
}
```

Remarks

- Print implementation straghtforward, omitted
- `assert` protects against null pointers
- Inclusion of `Stack.h` encourages consistency
- `Node` content returned upon distruction
- `malloc` and `free` syntax ugly
- Must always remember to use `nodeDestroy`

Stack.c, Stack Implementation

```
void stackInit( Stack* s )
{
  assert( s );
  s->top = (Node*)0;
  s->count = 0;
}

void stackFlush( Stack* s )
{
  assert( s );
  while( (s->count)>0 )
  {
    nodeDestroy( stackPop(s) );
  }
}

int stackIsEmpty( Stack* s )
{
  assert( s );
  return( (s->count)==0 );
}
```

Stack Implementation Continued

```
void stackPush( Node* n, Stack* s )
{
  assert( n ); assert( s );
  n->next = s->top;
  (s->top) = n;
  ++(s->count);
}

Node* stackPop( Stack* s )
{
  Node* n;
  assert( s );
  assert( (s->count)>0 );
  n = s->top;
  s->top = (s->top)->next;
  --(s->count);
  return( n );
}
```

Stack Implementation Final

```
void stackPrint( Stack* s )
{
  int i;
  Node* n;
  assert( s );

  printf( "Stack\n-----\n" );

  n = s->top;
  for( i=0; i<(s->count); ++i )
  {
    nodePrint( n );
    n = n->next;
  }
}
```

- Remarks
  - `Stack` implementation straghtforward, for future reference

C++/Class: Stack.h, Node

```
class Node
{
  friend class Stack;
  public:
    Node( int i ); // Node( const Node& n );
    ~Node( void );
    void print( void );
  protected:
  private:
    Node* theNext;
    int theContent;
};
```

- Remarks

  - Same `ifndef` wrapper as for C
  - Ordinary users can't access private data
  - `Stack` is `friend` and can access everything
  - `Node(int i)` is constructor

    Takes resposibility for creation and initialization
  - `~Node(void)` is destructor

    Takes resposibility for clean up and deletion

C++/Class: Stack.h, Stack

```
class Stack
{
  public:
    Stack( void );
    ~Stack( void ); // Stack( const Stack& n );
    int isEmpty( void );
    void push( Node* n );
    Node* pop( void );
    void flush( void );
    void print( void );
  private:
    Node* theTop;
    int theCount;
};
```

- Remarks
  - No free-standing routines needed

    `push`, `pop`, `isEmpty`, `flush`, `print` inside class
  - `Stack(void)` is constructor
  - `~Stack(void)` is destructor

Examples of Use

```
#include <iostream.h>
#include "Stack.h"
#define MAX 5

int main( void )
{
  Stack s;

  cout << "\n*******\nPUSHING\n*******\n";
  for( int i=0; i<MAX; ++i )
  {
    s.push( new Node(i) );
    cout << "\nFor i =" << i << "\n=========\n";
    s.print();
  }

  cout << "\n\n*******\nPOPPING\n*******\n\n";
  while( !s.isEmpty() )
  { Node* n; n = s.pop(); n->print(); delete n; }

  s.push( new Node(99) ); s.flush();
  return( 0 );
}
```

Remarks

- Syntax for access to member procedures reflects `struct`
- `iostream` replaces `stdio`

  `cout` prettier, easier, more flexible than `printf`
- Identifiers can be declared at point of first usage
- `new` and `delete` more convenient than `malloc` and `free`

  Included in C++; `stdlib` not needed

Stack.cc, Node Implementation

```
Node::Node( int i )
:  theNext((Node*)0), theContent(i)
{}

Node::~Node( void )
{}

void Node::print( void )
{ cout << "Node = " << theContent << "\n"; }
```

- Remarks
  - `Node::` specifies which class owns procedure

    Scope resolution specifier
  - `:theNext((Node*)0)` is an initializer

    Initialize in order listed in class
  - Constructor and destructor could include code

    Example: if `Node` contained a dynamic data structure

Stack.cc, Stack Implementation

```
Stack::Stack( void )
:  theTop((Node*)0), theCount(0)
{}

Stack::~Stack( void )
{ flush(); }

int Stack::isEmpty( void )
{ return( theCount==0 ); }
```

- Remarks
  - `.cc` file extension rather than `.c`

    Other possibilities: `.cp`, `.c++`, `.cpp`, ...
  - Cleanup code in destructor
  - Scope resolution equivalences

    `flush()`, `Stack::flush()`, and `this->flush()`

    `theCount`, `Stack::theCount`, and `this->theCount`

Stack Implementation Continued

```
void Stack::push( Node* n )
{
  assert( n );
  n->theNext = theTop;
  theTop = n;
  ++theCount;
}

Node* Stack::pop( void )
{
  Node* n;
  assert( theCount>0 );
  n = theTop;
  theTop = theTop->theNext;
  --theCount;
  return( n );
}
```

- Remarks
  - Fewer uses of `assert` needed
  - Porting from C straightforward

Stack Implementation Final

```
void Stack::flush( void )
{
  while( theCount>0 )
  { delete pop(); }
}

void Stack::print( void )
{
  Node* n;
  cout << "Stack\n-----\n";
  n = theTop;
  for( int i=0; i<theCount; ++i )
  { n->print(); n = n->theNext; }
}
```

- Remarks

– `delete pop()` should be clear

  `pop` returns a `Node*` and `delete` uses it

– Syntax of `struct` access again

  `n->print()` where `n` is a `Node*`

– Compiler allows `n->theNext` only because `Stack` is `friend`

Improvements

- `Node` can only have `int` contents
  - `Node` structure and contents should be separate
  - `Stack` only needs to deal with `Node` structure
- Make `Node` an abstraction
  - Embody only linkage structure
  - Define common properties of any kind of node
  - Extend to various kinds of nodes-with-contents
- Polymorphic (generic) code should be possible
  - Use only common node properties
  - Push and pop random `Node` types

Example

```
char c[MAX+1]="abcde";
Stack s;

for( int i=0; i<MAX; ++i )
{
  if( rand()&32 )
  {
    s.push( new IntNode(i) );
  } else {
    s.push( new CharNode(c[i]) );
  }
}

while( !s.isEmpty() )
{ Node* n; n = s.pop(); n->print(); delete n; }
```

- Remarks
  - Two kinds of nodes

    `IntNode` and `CharNode`
  - `Stack` handles both
  - `print` in the common interface
  - `while` body is polymorphic code; `print` is dynamically bound

C++/Derived Class: Stack.h, Node

```
class Node
{
  friend class Stack;
  public:
    Node( void );
    virtual ~Node( void );
    virtual void print( void )=0;
  protected:
  private:
    Node* theNext;
};

class IntNode :  public Node  // CharNode
{
  public:
    IntNode( int i );         // CharNode, char
    ~IntNode( void );         // CharNode
    void print( void );
  protected:
  private:
    int theContent;           // char
};
```

Remarks

- `IntNode` and `CharNode` derived from `Node`

  Member procedures/data may be added

  Member procedures may be redefined

  Likewise member data, although less common

- One-line comments start with `//`
- Comments indicate changes for `CharNode` structure
- `print` is the common interface defined by `Node`

  `print(...)=0` indicates no implementation

  No `Node` object may ever be declared

  Derived classes must implement

- `virtual` keyword enables dynamic binding

  Enabled for pointer (or reference) to derived class

  Destructor must also be `virtual`

Node Implementation

```
Node::Node( void )
:  theNext((Node*)0)
{}

Node::~Node( void )
{}

IntNode::IntNode( int i )
:  theContent(i)
{}

IntNode::~IntNode( void )
{}

void IntNode::print( void )
{ cout << "IntNode = " << theContent << "\n"; }
```

- Remarks
  - `Node` is simple

    Implementation only requires dividing up code and data
  - `Stack` requires no change at all

Templates, the Final Frontier

```
class Node
{
  friend class Stack;
  public:
    Node( void );
    virtual ~Node( void );
    virtual void print( void )=0;
  protected:
  private:
    Node* theNext;
};

template< class T >
class TypeNode :  public Node
{
  public:
    TypeNode( T i );
    ~TypeNode( void );
    void print( void );
  protected:
  private:
    T theContent;
};
```

Example of Use

```
char c[MAX+1]="abcde";
Stack s;

for( int i=0; i<MAX; ++i )
{
  if( rand()&32 )
  {
    s.push( new TypeNode<int>(i) );
  } else {
    s.push( new TypeNode<char>(c[i]) );
  }
}

while( !s.isEmpty() )
{ Node* n; n = s.pop(); n->print(); delete n; }
```

Implementation

```
template< class T >
TypeNode<T>::TypeNode( T i )
:  theContent(i)
{}

template< class T >
TypeNode<T>::~TypeNode( void )
{}

template< class T >
void TypeNode<T>::print( void )
{ cout << "TypeNode = " << theContent << "\n"; }
```

- Remarks
  - `cout operator<<` must be defined for any `T` used
  - No other code changes from non-template version
  - Templates are currently very poorly supported