Object-Oriented Design and Programming

C++ Basic Examples

Bounded Stack Example

Linked List Stack Example

UNIX File ADT Example

Specification for a String ADT

String Class ADT Example

Circular Queue Class Example

Matrix Class Example

Bounded Stack Example

- The following program implements a bounded stack abstraction
  - This is the solution to the first programming assignment

- *e.g.*,

```
/* The stack consists of ELEMENT_TYPEs. */
typedef int ELEMENT_TYPE;
class Stack {
private:
    /* Bottom and maximum size of the stack. */
    enum {BOTTOM = 0, SIZE = 100};
    /* Keeps track of the current top of stack. */
    int stack_top;
    /* Holds the stack's contents. */
    ELEMENT_TYPE stack[SIZE];
```

Bounded Stack Example (cont'd)

- /* The public section. */

```
public:
    /* Initialize a new stack so that it is empty. */
    Stack (void);
    /* Copy constructor */
    Stack (const Stack &s);
    /* Assignment operator */
    Stack &operator= (const Stack &s);
    /* Perform actions when stack is destroyed. */
    ~Stack (void);
    /* Place a new item on top of the stack
    Does not check if the stack is full. */
    void push (ELEMENT_TYPE new_item);
    /* Remove and return the top stack item
    Does not check if stack is empty. */
    ELEMENT_TYPE pop (void);
    /* Return top stack item without removing it
    Does not check if stack is empty. */
    ELEMENT_TYPE top (void);
    /* Returns 1 if the stack is empty,
    otherwise returns 0. */
    int is_empty (void);
    /* Returns 1 if stack full, else returns 0. */
    int is_full (void);
};
```

Bounded Stack Example (cont'd)

- Implementation of a bounded stack abstraction in C++

```
#include "stack.h"
Stack::Stack (void)
    : stack_top (Stack::BOTTOM) {}
Stack::Stack (const Stack &s)
    : stack_top (s.stack_top)
{
    for (int i = 0; i < s.stack_top; i++)
        this->stack[i] = s.stack[i];
}
Stack &
Stack::operator= (const Stack &s)
{
    if (this != &s) {
        this->stack_top = s.stack_top;
        for (int i = 0; i < s.stack_top; i++)
            this->stack[i] = s.stack[i];
    }
    return *this;
}
Stack::Stack (void)
{
    this->stack_top = Stack::BOTTOM;
}
Stack::~Stack (void)
{
    /* Not strictly necessary... */
    this->stack_top = Stack::BOTTOM;
}
```

Bounded Stack Example (cont'd)

- Implementation (cont'd)

```
void Stack::push (ELEMENT_TYPE new_item)
{
    this->stack[this->stack_top++] = new_item;
}
ELEMENT_TYPE Stack::pop (void)
{
    return this->stack[--this->stack_top];
}
ELEMENT_TYPE Stack::top (void)
{
    return this->stack[this->stack_top - 1];
}
int Stack::is_empty (void)
{
    return this->stack_top == Stack::BOTTOM;
}
int Stack::is_full (void)
{
    return this->stack_top >= Stack::SIZE;
}
```

Bounded Stack Example (cont'd)

- Use of a bounded stack to reverse a name

```
#include <stream.h>
#include "stack.h"

int main (int argc, char *argv[]) {
     const int MAX_NAME_LEN = 80;
     char name[MAX_NAME_LEN];
     Stack stack;

     cout << "please enter your name..: ";
     cin.getline (name, MAX_NAME_LEN);

     for (int i = 0; name[i] != '\n' && !stack.is_full (); i++)
          stack.push (ELEMENT_TYPE (name[i]));

     cout << "your name backwards is..: ";

     while (!stack.is_empty ())
          cout << char (stack.pop ());
     cout << '\n';
}
```

Linked List Stack Example

- This is a reimplement the stack ADT using dynamic memory allocation to remove limitations on the stack size
  - Note that extra operators have been added to support efficient memory management

- // File stack.h

```
typedef int ELEMENT_TYPE;
class Stack {
public:
    /* First 9 member functions are the same...*/
    Stack (void): head (0) {}
    Stack (const Stack &s);
    Stack &operator= (const Stack &s);
    ~Stack (void);
    void push (ELEMENT_TYPE new_item);
    ELEMENT_TYPE pop (void);
    ELEMENT_TYPE top (void);
    int is_empty (void) { return this->head == 0; }
    int is_full (void) { return 0; }
    /* New function, for memory management...*/
    static void release_free_list (void);
```

Linked List Stack Example (cont'd)

- *e.g.*,

```
private:
    struct Node { /* Should be a class? */
        /* Memory management facilities. */
        static Node *free_list;
        static void free_all_nodes (void);
        void *operator new (size_t bytes);
        void operator delete (void *ptr);

        /* Stack Node declarations. */
        Node *next;
        ELEMENT_TYPE item;

        Node (ELEMENT_TYPE i, Node *n):
            item (i), next (n) {}
    };
    Node *head;
};
```

Linked List Stack Example (cont'd)

- // File stack.C

```
#include "stack.h"
void *Node::operator new (size_t bytes) {
    Node *temp = Node::free_list;
    if (temp != 0)
        Node::free_list = Node::free_list->next;
    else
        temp = (Node *) new char[bytes];
        /* Or temp = ::new Node; */
    return temp;
}

void Node::operator delete (void *ptr) {
    /* Note, a cast must be used... */
    ((Node *) ptr)->next = Node::free_list;
    Node::free_list = ptr;
}

void Node::free_all_nodes (void) {
    while (Node::free_list != 0) {
        Node *temp = Node::free_list;
        Node::free_list = Node::free_list->next;
        ::delete temp;
        /* Or delete (void *) temp; */
    }
}

void Stack::release_free_list (void) {
    Node::free_all_nodes ();
}
```

Linked List Stack Example (cont'd)

- // File stack.C

```
Stack::~Stack (void) {
     while (this->head != 0) {
          Node *temp = this->head;
          this->head = this->head->next;
          delete temp;
          /* Calls Node::delete; */
     }
}

void Stack::push (ELEMENT_TYPE new_item) {
     /* Calls Node::new; followed by Node::Node; */
     this->head = new Node (new_item, this->head);
}

ELEMENT_TYPE Stack::pop (void) {
     ELEMENT_TYPE return_value = this->head->item;
     Node *temp = this->head;
     this->head = this->head->next;
     delete temp;
     /* Calls Node::delete; */
     return return_value;
}

ELEMENT_TYPE Stack::top (void) {
     return this->head->item;
}
```

Linked List Stack Example (cont'd)

- // File stack.C

```
Stack::Stack (const Stack &s): head (0) {
     for (Node *n = s.head; n != 0; n = n->next)
          this->push (n->item);
}

Stack &Stack::operator= (const Stack &s) {
     /* Note, we could optimize this! */
     for (Node *n = this->head; n != 0; n = n->next)
          n->pop ();
     for (n = s.head; n != 0; n = n->next)
          this->push (n->item);
}
```

UNIX File ADT Example

- The following is an example of applying a C++ wrapper around existing C code
  - Once we have the C++ interface, we can use any implementation that fulfills the specified semantics

- Goals:
  1. Improve type-security
  2. Improve consistency and ease-of-use

- *Publicly Available Operations*:
  - *Open* and *close* a file for reading and/or writing
  - Read/write a single *character* or a single *line* from/to the file
  - Control functions (*e.g.*, **`seek`**, **`rewind`**, **`fcntl`**, etc.)

- *Implementation Details*:
  - Note: we'll cheat and use the Standard ANSI C library routines for our implementation

UNIX File ADT Example (cont'd)

- // File file.h

```
#ifndef _FILE_H
#define _FILE_H
#include <stdio.h>

class File
{
public:
     File (char *filename, char *mode);
     File (void);
     ~File (void);
     int open (char *filename, char *mode);
     int close (void);
     int get_char (void);
     int get_line (char *buf, int max_length);
     int put_char (char c);
     int put_line (char *str);
     int seek (long offset, int ptrname);
     int rewind (void);
     int fcntl (int cmd, int arg);
private:
     FILE *fp;
};
#endif
```

- Note, for maximum efficiency, we should include **inline** member functions...

UNIX File ADT Example (cont'd)

- Gory details hidden in our implementation...
  - A FILE is allocated from a global structure called `_iob`
  - A buffer of size BUFSIZ is allocated
  - The file exists on a disk partitioned into blocks
  - The blocks are not necessarily contiguous
  - A structure called an *inode* is used to keep track of the blocks

- Here's an fragment from /usr/include/stdio.h

```
#define BUFSIZ 1024
extern struct _iobuf {
        int _cnt;
        unsigned char *_ptr;
        unsigned char *_base;
        int _bufsiz;
        short _flag;
        char _file;
} _iob[];
#define FILE struct _iobuf
```

UNIX File ADT Example (cont'd)

- // File file.h

```
#include <fcntl.h>
File::File (char *filename, char *mode) {
     this->open (filename, mode);
}

File::File (void): fp (0) {}

File::~File (void) { this->close (); }

int File::open (char *filename, char *mode) {
     this->fp = ::fopen (filename, mode);
}

int File::close (void) { ::fclose (this->fp); }

int File::get_char (void) { return getc (this->fp); }
```

UNIX File ADT Example (cont'd)

- // File file.h

```
int File::get_line (char *buf, int max_length) {
     return ::fgets (buf, max_length, this->fp)
          ? strlen (buf) : 0;
}

int File::put_char (char c) { return putc (c, this->fp); }

int File::put_line (char *s) { return ::fputs (s, this->fp); }

int File::seek (long offset, int ptrname) {
     return ::fseek (this->fp, offset, ptrname);
}

int File::rewind (void) { return this->seek (0, 0); }

int fcntl (int cmd, int arg) {
     return ::fcntl (fileno (this->fp), cmd, arg);
}
```

UNIX File ADT Example (cont'd)

- // File main.c

```
#include <stdio.h>
int copy_files (char *read_file, char *write_file) {
     FILE *fin = fopen (read_file, "r");
     FILE *fout = fopen (write_file, "w");
     int c;

     while ((c = getc (fin)) != EOF)
          putc (c, fout);
     fclose (fin);
     fclose (fout);
}
```

- // File main.C

```
#include "file.h"
int copy_files (char *read_file, char *write_file) {
     File in (read_file, "r");
     File out (write_file, "w");
     int c;

     while ((c = in.get_char ()) != EOF)
          out.put_char (c);
     // Calls destructor to close files
}
```

String Class ADT Example

- *Motivation*: built-in support for strings in C/C++ is rather weak

  1. It is easy to make mistakes since strings do not always work the way you might expect. For example, the following causes both str1 and str2 to point at "bar":

     char *str1 = "foo", *str2 = "bar";
     str1 = str2;

  2. Strings are NUL-terminated

     – Certain standard string operations do not work efficiently for long strings, *e.g.*, **`strlen`** and **`strcpy`**

     – Cannot store a NUL character in a string

  3. Certain operations silently allow hard-to-detect errors, *e.g.*,

     char s[] = "hello";
     char t[] = "world";
     strcat (s, t); // ERROR!!!

String Class ADT Example (cont'd)

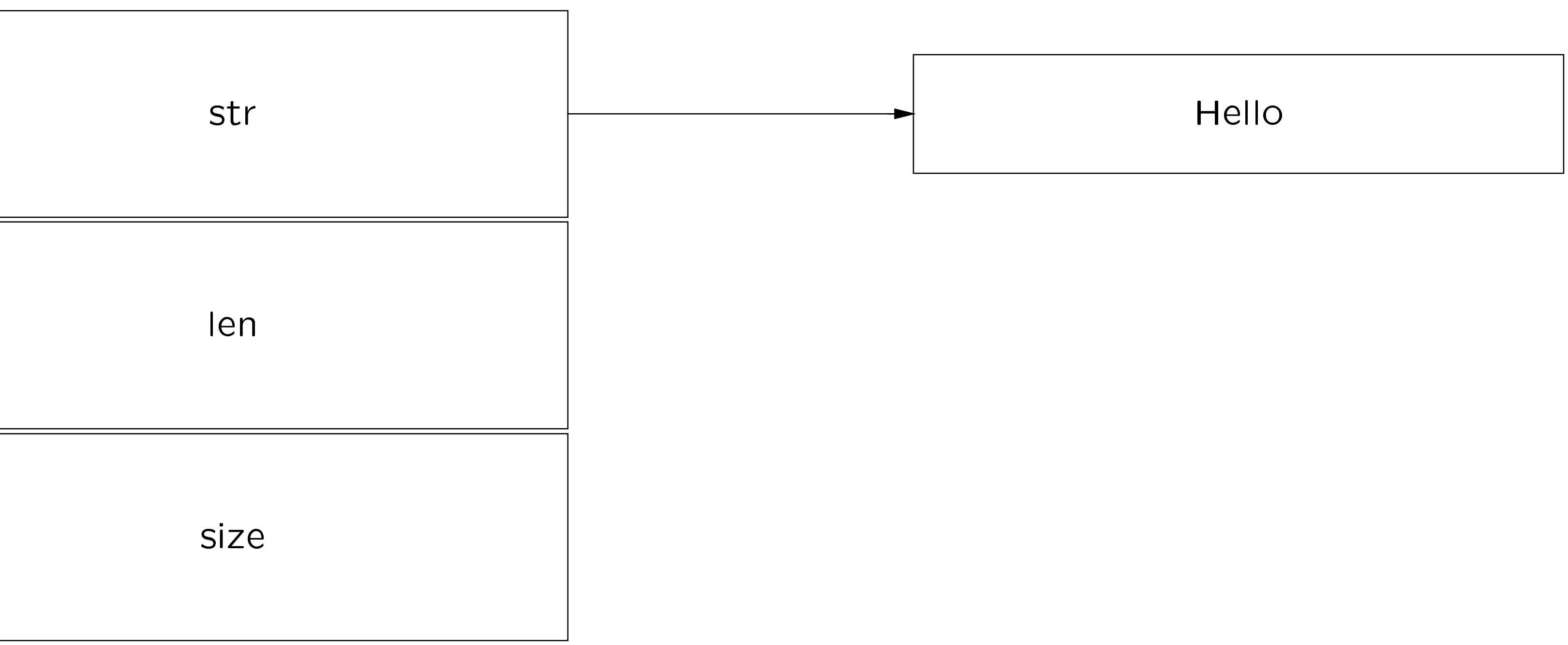
- Therefore, we need flexible, safe, and efficient support for operations like:
  - Automatically creating and destroying Strings
  - Assigning/initializing Strings from other Strings
  - Concatenating two Strings
    - destructive and/or non-destructive
  - Comparing Strings for equality/inequality
  - Printing Strings
  - Returning the length of a String
  - Accessing individual characters in a string
  - Inserting/deleting/extracting/finding substrings
  - Converting built-in C/C++ strings to String
  - Regular expression matching

String Class ADT Example (cont'd)

- *e.g.*,

```
class String {
private:
     /* pointer to dynamically allocated data */
     char *str;
     /* current length of the string */
     int len;
     /* total size of allocated buffer */
     int size;
     /* Returns best size for underlying allocator,
          smallest power of 2 >= n */
     int best_new_size (int);
     /* Increase string to size N. */
     void grow_string (int);
     /* make an uninitialized string of given size. */
     String (int);
     /* Return the min of two integers. */
     int min (int a, int b);
     /* Move memory up or down by skip_amount */
     void copy_up (int index_pos, int skip_amount);
     void copy_down (int index_pos, int skip_amount);
```

String Class ADT Example (cont'd)



- Layout for a String object, *e.g.*,

  String s ("Hello");

String Class ADT Example (cont'd)

- *e.g.*,

```
public:
    /* constructors and destructors */
    String (const char * = "");
    String (const String &);
    String &operator= (const String&);
    ~String (void);

    /* destructive concat */
    String &operator+= (const String&);
    /* non-destructive concat */
    friend String
        operator+ (const String&, const String&);

    /* equality tests */
    friend int operator== (const String&, const String&);
    friend int operator!= (const String&, const String&);
    friend int operator< (const String&, const String&);
    friend int operator> (const String&, const String&);
    friend int operator<= (const String&, const String&);
    friend int operator>= (const String&, const String&);
    /* Same return value as strcmp...*/
    friend int cmp (const String&, const String&);
```

String Class ADT Example (cont'd)

- *e.g.*, /* public continued. */

```
public:
    /* output operator */
    ostream &operator << (ostream &, const String&);

    /* Indexing operator */
    char &operator[] (int);
    int length (void) const;

    /* Dynamically allocates a C/C++ string copy */
    operator char *() const;
```

String Class ADT Example (cont'd)

- *e.g.*,

```
public:
     /* insert String s starting at index_pos, return −1
          if index_pos is out of range, else 0. */
     int insert (int index_pos, const String &s);

     /* remove length characters starting at index_pos,
          returns −1 if index_pos is out of range, else 0. */
     int remove (int index_pos, int length);

     /* return the starting index position if S is a substring
          return −1 if S is not a substring, else 0. */
     int find (const String &s) const;

     /* copy count characters starting at index_pos and retu
          new string containing these characters. Returns an
          empty String if index_pos or count are out of rang
     String substr (int index_pos, int count) const;

     /* Replace substring S with substring T in
     this String. Returns the index where S
     begins (if found), else return −1. */
     int String::replace (const String &s, const String &t);
};
```

String Class ADT Example

- /* Optimization code */

```
#if defined (__OPTIMIZE__)
inline int String::min (int a, int b) { a > b ? b : a; }

inline char &String::operator[] (int index) const {
#if defined (EXCEPTIONS)
     if (index < 0 || index >= this->length ())
          throw (RANGE_ERROR);
#else
     assert (index >= 0 && index < this->length ());
#endif
     return this->str[index];
}

inline int String::length (void) const {
     return this->len;
}
#endif /* defined (__OPTIMIZE__) */
```

String Class ADT Example

- /* Implementation of String ADT in C++. Note that we try to minimize the number of new allocations whenever possible. */

```
#include <string.h>
#include <stdlib.h>
#include "String.h"
```

- /* Overload operator new to use malloc. */

```
static void *operator new (size_t size) {
      return malloc (size);
}
```

- /* Overload operator new to act like realloc. */

```
static void *
operator new (size_t size, void *ptr, int new_len) {
      return realloc (ptr, size * new_len);
}
```

String Class ADT Example

- /* Implementation of String ADT in C++. Returns the smallest power of two greater than or equal to n! */

```
int String::best_new_size (int n) {
     if (n <= 0)
          return 1;
     else
#if defined (USE_POWER_OF_TWO_HACK)
     return n -= 1, n |= n >> 1, n |= n >> 2,
          n |= n >> 4, n |= n >> 8, n |= n >> 16,
          n + 1;
#else
     {
          for (int i = 1; i < n; i += i)
               ;
          return i;
     }
#endif
}
```

String Class ADT Example

- /* Enlarge the String to **new_size**. */

```
void String::grow_string (int new_size) {
     this->size = best_new_size (new_size);
#if defined (__GNUG__)
     this->str = new {this->str, this->size} char;
#else
     this->str = new (this->str, this->size) char;
#endif
}
```

- /* Make an uninitialized string of size **sz**. */

```
String::String (int sz):
     len (sz), size (best_new_size (sz)) {
     this->str = new char[this->size];
}
```

String Class ADT Example

- /* Move the contents of **this** String up **skip_amount** characters starting at **index_pos** (automatically expand String length if necessary). */

```
inline void String::copy_up (int index_pos,
                             int skip_amount) {
     int new_len = this->len + skip_amount;
     if (new_len >= this->size)
          this->grow_string (new_len);
     for (int i = this->len; i >= index_pos; i--)
          this->str[i + skip_amount] =
               this->str[i];
}
```

- /* Starting at **index_pos + skip_amount**, copy all the remaining characters in the string down **skip_amount** number of characters. */

```
inline void String::copy_down (int index_pos,
                               int skip_amount) {
     int number_to_move =
          this->len - (index_pos + skip_amount);
     for (int i = 0; i <= number_to_move; i++)
          this->str[index_pos + i] =
               this->str[index_pos + i + skip_amount];
          // memmove (this->str + index_pos,
          // this->str + (index_pos + skip_amount),
          // number_to_move);
}
```

String Class ADT Example

- /* Create a new String from an existing C string. */

```
String::String (const char *s):
     len ((s = s == 0 ? "" : s), strlen (s)),
     size (best_new_size (len)) {
     this->str =
          memcpy (new char[this->size], s, this->len);
}
```

- /* Create a new String from an existing String. */

```
String::String (const String &s): len (s.len), size (s.size) {
     this->str =
          memcpy (new char[this->size], s.str, s.len);
}
```

- /* Free dynamically allocated memory when String goes out of scope. */

```
String::~String (void) {
     delete this->str;
}
```

String Class ADT Example

- /* Performs "'destructive"' concatenation */

```
String &String::operator+= (const String &s) {
     int new_len = this->len + s.len;
     if (this->size < new_len)
          this->grow_string (new_len);
     memcpy (this->str + this->len, s.str, s.len);
     this->len += s.len;
     return *this;
}
```

- /* Performs "'non-destructive"' concatenation (note, not a member function) */

```
String operator+ (const String &s,
                  const String &t) {
     String tmp (s.len + t.len);
     memcpy (tmp.str, s.str, s.len);
     memcpy (tmp.str + s.len, t.str, t.len);
     return tmp;
}
```

- /* Assign String S to **this** String. */

```
String &String::operator= (const String &s) {
     if (this != &s) {
          if (this->size <= s.len)
               this->grow_string (s.len);
          memcpy (this->str, s.str, s.len);
          this->len = s.len;
     }
     return *this;
}
```

String Class ADT Example

- /* The following 6 overloaded operators handle equality and relational operations. */

```
int operator== (const String &s, const String &t) {
    return s.len == t.len &&
        memcmp (s.str, t.str, t.len) == 0;
}
int operator!= (const String &s, const String &t) {
    return s.len != t.len ||
        memcmp (s.str, t.str, t.len) != 0;
}
int operator< (const String &s, const String &t) {
    int result = memcmp (s.str, t.str, min (s.len, t.len));
    return result < 0 ? 1 : (result == 0 ? s.len < t.len : 0);
}
int operator<= (const String &s, const String &t) {
    return memcmp (s.str, t.str, min (s.len, t.len)) <= 0;
}
int operator> (const String &s, const String &t) {
    int result = memcmp (s.str, t.str, min (s.len, t.len));
    return result > 0 ? 1 : (result == 0 ? s.len > t.len : 0);
}
int operator>= (const String &s, const String &t) {
    return memcmp (s.str, t.str, min (s.len, t.len)) >= 0;
}
```

String Class ADT Example

- /* Performs a 3-way comparison, like **ansi** C's **strcmp** library function. */

```
int cmp (const String &s, const String &t) {
     int result = memcmp (s.str, t.str, min (s.len, t.len));
     return result != 0 ? result : s.len − t.len;
}
```

- /* Output operator */

```
ostream &operator << (ostream &stream,
                const String &str) {
     for (int i = 0; i < str.len; i++)
          stream << str[i]; // actually str.operator[] (i);
     return stream;
}
```

- /* Converts C++ String to C string. */

```
String::operator char *() const {
     char *s = memcpy (new char[this->len + 1],
                    this->str, this->len);
     s[this->len] = '\0';
     return s;
}
```

String Class ADT Example

- /* If not optimizing */

```
#if !defined (__OPTIMIZE__)
```

- /* Min operator */

```
int String::min (int a, int b) { a > b ? b : a; }
```

- /* Index operator. */

```
char &String::operator[] (int index) const {
#if defined (EXCEPTIONS)
     if (index < 0 || index >= this->len)
          throw (RANGE_ERROR);
#else
     assert (index >= 0 && index < this->len);
#endif
     return this->str[index];
}
```

- /* Returns the length of the String. */

```
int String::length (void) const {
     return this->len;
}
#endif /* !defined (__OPTIMIZE__) */
```

String Class ADT Example

- /* Challenge code */

  ```
  #if defined (CHALLENGE)
  ```

- /* Insert String S into **this** String at offset **index_pos**. */

  ```
  int String::insert (int index_pos, const String &s) {
       if (index_pos < 0 || index_pos > this->len)
            return −1;
       else {
            this->copy_up (index_pos, s.len);
            memcpy (this->str + index_pos, s.str, s.len);
            this->len += s.len;
            return 1;
       }
  }
  ```

- /* Remove **len** characters from **string**, starting at offset **index_pos**. */

  ```
  int String::remove (int index_pos, int len) {
       if (index_pos < 0 || index_pos + len > this->len)
            return −1;
       else {
            this->copy_down (index_pos, len);
            this->len -= len;
            return 1;
       }
  }
  ```

String Class ADT Example

- /* Check whether String S is a substring in **this** String. Return −1 if it is not, otherwise return the index position where the substring begins. */

```
int String::find (const String &s) const {
     char firstc = s[0]; // s.operator[] (0);
     int end_index = this->len − s.len + 1;
     for (int i = 0;
          i < end_index
          && ((*this)[i] != firstc ||
          memcmp (&this->str[i] + 1, s.str + 1, s.len − 1) !
          i++)
          ;
     return i < end_index ? i : −1;
}
```

- /* Extract a substring from **this** String of size **count** starting at **index_pos**. */

```
String String::substr (int index_pos, int count) const {
     if (index_pos < 0 || index_pos + count > this->len)
          return "";
     else {
          String tmp (count);
          for (int i = 0; i < count; i++)
               tmp[i] = (*this)[index_pos++];
               /* tmp.operator[] (i) =
               this->operator[] (index_pos++); */
          return tmp;
     }
}
```

String Class ADT Example

- /* Replace substring S with substring T in **this** String. Returns the index where S begins (if found), else return −1. */

```
int String::replace (const String &s, const String &t) {
     int index = this->find (s);
     if (index < 0)
          return −1;
     else {
          int delta = s.len − t.len;
          if (delta == 0)
               memcpy (&this->str[index], t.str, t.len);
          else if (delta > 0) {
               memcpy (&this->str[index], t.str, t.len);
               this->copy_down (index + t.len, delta);
               this->len -= delta;
          }
          else {
               delta = -delta;
               this->copy_up (index + s.len, delta);
               memcpy (&this->str[index], t.str, t.len);
               this->len += delta;
          }
          return index;
     }
}
#endif /* CHALLENGE */
```

Circular Queue Class Example

- This file implements a bounded circular queue
  - It illustrates the use of reference parameter return
  - It also illustrates the importance of designing the interface for *change* (*i.e.*, using parameterized types)

- // File queue.h

```
template <class TYPE> class Queue {
private:
     int count, size;
     int front, rear;
     TYPE *array;
     int next (int index);
public:
     Queue (int sz = 100): size (sz),
          front (0), rear (0), count (0)
          array (new TYPE[sz]) {}
     ~Queue (void) { delete this->array; }
     int enqueue (TYPE item);
     int dequeue (TYPE &item);
     int is_empty (void);
     int is_full (void);
};
```

Circular Queue Class Example (cont'd)

- // File queue.C

```
#include "int-queue.h"

template <class TYPE>
inline int Queue<TYPE>::next (int i) {
     return (i + 1) % this->size;
}

template <class TYPE>
int Queue<TYPE>::is_empty (void) {
     return this->count == 0;
}

template <class TYPE>
int Queue<TYPE>::is_full (void) {
     return this->count >= this->size;
}
```

Circular Queue Class Example (cont'd)

- // File queue.C

```
template <class TYPE>
int Queue<TYPE>::enqueue (int item) {
     this->array[this->rear] = item;
     this->rear = this->next (this->rear);
     this->count++;
     return 1;
}

template <class TYPE>
int Queue<TYPE>::dequeue (TYPE &item) {
     item = this->array[this->front];
     this->front = this->next (this->front);
     this->count--;
     return 1;
}
```

Circular Queue Class Example (cont'd)

- // File main.C

```
#include <stream.h>
#include "queue.h"

int main (void) {
     Queue<int> q; // Defaults to 100

     for (int i = 0; !q.is_full () && i < 10; i++) {
          cout << "enqueueing " << i << "\n";
          q.queue (i);
     }

     while (!q.is_empty ()) {
          int i;
          q.dequeue (i);
          cout << "dequeueing " << i << "\n";
     }
     // ~Queue () destructor called here!
}
```

Matrix Class Example

- Shows the use of overloading to allow C++ to use two-dimensional array indexing syntax similar to Pascal (don't try this at home!)

```
class Double_Index {
public:
    int i1, i2;
    Double_Index (int i, int j): i1 (i), i2 (j) {}
};

class Index {
public:
    Index (void): i (0) { }
    Index (int j): i (j) { }
    operator int &() { return i; }
    Double_Index operator, (Index j) {
        return Double_Index (this->i, j.i);
    }
    const Index &operator= (int j) {
        this->i = j; return *this;
    }
    const Index &operator++ (void) {
        ++this->i; return *this;
    }
private:
    int i;
};
```

Matrix Class Example (cont'd)

- *e.g.*,

```
class Two_D_Matrix { // Should support dynamic matrices.
private:
     enum { ROW_SIZE = 20, COL_SIZE = 10 };
     int m[ROW_SIZE][COL_SIZE];
public:
     int &operator[] (Double_Index I) {
          return m[I.i1][I.i2];
     }
     int row_size (void) { return ROW_SIZE; }
     int col_size (void) { return COL_SIZE; }
};

int main (void) {
     Two_D_Matrix m;

     for (Index i = 0; i < m.row_size (); ++i)
          for (Index j = 0; j < m.col_size (); ++j)
               m[i, j] = 10 * i + j;
               // m.operator[] (i.operator, (j)) ..
     for (i = 0; i < m.row_size (); ++i) {
          for (Index j = 0; j < m.col_size (); ++j)
               printf ("%4d ", m[i, j]);
          printf ("\n");
     }
     exit (0);
}
```

Matrix Class Example (cont'd)

- Output from Two_D_Matrix program:

```
  0   1   2   3   4   5   6   7   8   9
 10  11  12  13  14  15  16  17  18  19
 20  21  22  23  24  25  26  27  28  29
 30  31  32  33  34  35  36  37  38  39
 40  41  42  43  44  45  46  47  48  49
 50  51  52  53  54  55  56  57  58  59
 60  61  62  63  64  65  66  67  68  69
 70  71  72  73  74  75  76  77  78  79
 80  81  82  83  84  85  86  87  88  89
 90  91  92  93  94  95  96  97  98  99
100 101 102 103 104 105 106 107 108 109
110 111 112 113 114 115 116 117 118 119
120 121 122 123 124 125 126 127 128 129
130 131 132 133 134 135 136 137 138 139
140 141 142 143 144 145 146 147 148 149
150 151 152 153 154 155 156 157 158 159
160 161 162 163 164 165 166 167 168 169
170 171 172 173 174 175 176 177 178 179
180 181 182 183 184 185 186 187 188 189
190 191 192 193 194 195 196 197 198 199
```