CMPE 180-92

# Data Structures and Algorithms in C++

October 19 Class Meeting

---

Department of Computer Engineering
San Jose State University


Fall 2017
Instructor: Ron Mak
www.cs.sjsu.edu/~mak

# Midterm Stats

| | |
|---|---|
| median | 167 (84%) |
| average | 161 (81%) |
| std.dev. | 29 |

# Midterm Solutions: Section 2

1. Write the definition of the default constructor that initializes the real and imaginary parts to zero.

```
Complex::Complex() : re(0), im(0)
{
}
```

2. Write the definition of the constructor that initializes the real and imaginary parts to the respective values of its two parameters.

```
Complex::Complex(double re, double im) : re(re), im(im)
{
}
```

# Midterm Solutions: Section 2, *cont'd*

3. Write the declaration of the overloaded + operator that operates on a complex number by adding another complex number to it and returning a new complex sum.

```
Complex operator +(const Complex& other);
```

Since it operates on an existing **Complex** object, the overloaded + operator is best a member function. The existing object is the implied first parameter.

4. Write the declaration of the overloaded + operator that operates on a complex number by adding a real number to it and returning a new complex sum.

```
Complex operator +(const double r);
```

An acceptable solution is to make both friend functions. Then they would each have two parameters.

# Midterm Solutions: Section 2, *cont'd*

5. Write the declaration of the overloaded **+** operator that adds a real number and a complex number and returns a new complex sum.

```
friend Complex operator +(const double r, const Complex& c);
```

6. Write the declaration of the overloaded **<<** operator that writes a complex number to an output stream.

```
friend ostream& operator << (ostream &outs, const Complex& c);
```

Both must be friend functions because the first parameter cannot be the existing **`Complex`** object.

# Midterm Solutions: Section 2, *cont'd*

7. Write the definition of the overloaded + operator of question 2-3.

```
Complex Complex::operator +(const Complex& other)
{
    Complex c(re + other.re, im + other.im);
    return c;
}
```

8. Write the definition of the overloaded + operator of question 2-4.

```
Complex Complex::operator +(const double r)
{
    Complex c(re + r, im);
    return c;
}
```

# Midterm Solutions: Section 2, *cont'd*

9. Write the definition of the overloaded + operator of question 2-5.

```
Complex operator +(const double r, const Complex& c)
{
    Complex z(c.re + r, c.im);
    return z;
}
```

# Midterm Solutions: Section 2, *cont'd*

10. Write the definition of the overloaded << operator of question 2-6.

```
ostream& operator << (ostream &outs, const Complex& c)
{
    outs << c.re;
    if (c.im >= 0) outs << "+";
    outs << c.im << "i";

    return outs;
}
```

# Midterm Solutions: Section 3

1. What are the main advantages of distributing the library as source files rather than as precompiled binary files?

   To maintain portability. You can compile the source files for different target computers.

2. When you were installing MPIR, what did the **`configure`** script accomplish?

   The script checked your system to ensure that you had all the software necessary to do the build. It generated custom makefiles based on your machine environment and the options you specified.

# Midterm Solutions: Section 3, *cont'd*

3. When you were installing MPIR, what did the **`make install`** command accomplish?

> The command installed the MPIR library on your system.

4. When you compile a program on the command line, what option tells the **`g++`** command to use the MPIR library? What additional option is needed if the library was not installed in a standard location?

> **`-lmpir`** to link to the MPIR library
> **`-L`***directory* to tell the linker that the library is in the specified directory

# Assignment #7: Sample Solution

```
class BookNode
{
public:
    BookNode(Book book);
    Book get_book() const;

    BookNode *get_next() const;
    void set_next(BookNode *next_node);

private:
    Book book;        // this node's book
    BookNode *next;   // link to the next node in the list
};
```

# Assignment #7: Sample Solution, *cont'd*

```
BookNode::BookNode(Book book) : book(book), next(nullptr) {}

Book BookNode::get_book() const { return book; }

BookNode *BookNode::get_next() const { return next; }

void BookNode::set_next(BookNode *next_node) { next = next_node; }
```

# Assignment #7: Sample Solution, *cont'd*

```
class BookList
{
public:
    BookList(const string name);
    BookList(const string name, vector<BookList>& categories);
    BookList(const string name, const BookList& other, bool test(Book& book));

    void print();
    void delete_books();

private:
    string name;       // name of this book list
    BookNode *head;    // head of the list of book nodes
    BookNode *tail;    // tail of the list of book nodes

    void create();
    void append(const Book book);
    void insert(const Book book);
    void merge(const BookList& other);
    void merge(vector<BookList> categories);
    void copy(const BookList &other);
    void copy_if_pass_test(const BookList& other, bool test(Book& book));
};
```

# Assignment #7: Sample Solution, *cont'd*

```
BookList::BookList(const string name)
    : name(name), head(nullptr), tail(nullptr)
{
    create();
}

BookList::BookList(const string name, vector<BookList>& categories)
    : name(name), head(nullptr), tail(nullptr)
{
    merge(categories);
}

BookList::BookList(const string name, const BookList& other,
                   bool test(Book& book))
    : name(name), head(nullptr), tail(nullptr)
{
    copy_if_pass_test(other, test);
}
```

# Assignment #7: Sample Solution, *cont'd*

```
void BookList::create()
{
    string book_file_name = name + ".txt";
    ifstream book_file;

    book_file.open(book_file_name);
    if (book_file.fail())
    {
        cout << "Failed to open " << book_file_name << endl;
        return;
    }

    Book book;
    book_file >> book;  // read the first book

    while (!book_file.fail())
    {
        insert(book);       // insert the book into the sorted list
        book_file >> book;  // read the next book
    }

    book_file.close();
}
```

# Assignment #7: Sample Solution, *cont'd*

```
void BookList::print()
{
    int count = 0;
    cout << endl << "Book list: " << name << endl << endl;

    for (BookNode *ptr = head; ptr != nullptr; ptr = ptr->get_next())
    {
        cout << "   " << ptr->get_book() << endl;
        count++;
    }

    cout << "     (" << count << " books)" << endl;
}
```

# Assignment #7: Sample Solution, *cont'd*

```
void BookList::append(const Book book)
{
    BookNode *new_node = new BookNode(book);

    // First node in the list.
    if (head == nullptr)
    {
        head = tail = new_node;
    }

    // Subsequent node.
    else
    {
        tail->set_next(new_node);
        tail = new_node;
    }
}
```

# Assignment #7: Sample Solution, *cont'd*

```
void BookList::insert(const Book book)
{
    BookNode *new_node = new BookNode(book);
    string isbn = book.get_isbn();

    // First node in the list, or insertion before the first node.
    if ((head == nullptr) || (isbn < head->get_book().get_isbn()))
    {
        new_node = new BookNode(book);
        new_node->set_next(head);
        head = new_node;

        if (head->get_next() == nullptr) tail = head;
    }
```

# Assignment #7: Sample Solution, *cont'd*

```
    // Insertion somewhere after the first node.
    else
    {
        BookNode *ptr = head;
        BookNode *prev;

        // Look for where to insert into the book list.
        while ((ptr != nullptr) && (isbn > ptr->get_book().get_isbn()))
        {
            prev = ptr;
            ptr = ptr->get_next();
        }

        // Insert only if not a duplicate ISBN.
        if ((ptr == nullptr) || (isbn < ptr->get_book().get_isbn()))
        {
            prev->set_next(new_node);
            new_node->set_next(ptr);

            if (ptr == nullptr) tail = prev;
        }
    }
}
```

# Assignment #7: Sample Solution, *cont'd*

```
void BookList::merge(const BookList& other)
{
    // Insert each node of the other book list.
    for (BookNode *ptr = other.head; ptr != nullptr; ptr = ptr->get_next())
    {
        insert(ptr->get_book());
    }
}

void BookList::merge(vector<BookList> categories)
{
    // Merge category book lists.
    if (categories.size() > 0)
    {
        // Copy the first book list.
        copy(categories[0]);

        // Merge in the remaining book lists.
        for (int i = 1; i < categories.size(); i++) merge(categories[i]);
    }
}
```

# Assignment #7: Sample Solution, *cont'd*

```
void BookList::copy(const BookList &other)
{
    // Append a copy of each node of the other book list.
    for (BookNode *ptr = other.head; ptr != nullptr; ptr = ptr->get_next())
    {
        append(ptr->get_book());
    }
}

void BookList::copy_if_pass_test(const BookList& other, bool test(Book& book))
{
    // Append a copy of each node of the other book list that passes the test.
    for (BookNode *ptr = other.head; ptr != nullptr; ptr = ptr->get_next())
    {
        Book book = ptr->get_book();
        if (test(book)) append(book);
    }
}
```

# Assignment #7: Sample Solution, *cont'd*

```
void BookList::delete_books()
{
    BookNode *ptr = head;

    // Loop over the book nodes and delete each one.
    while (ptr != nullptr)
    {
        BookNode *next = ptr->get_next();
        delete ptr;
        ptr = next;
    }
}
```

# Assignment #7: Sample Solution, *cont'd*

```
const string CATEGORIES_FILE_NAME = "categories.txt";

/**
 * Make the category book lists.
 * @param category_names the vector of category names.
 * @return a vector of category book lists.
 */
vector<BookList> make_category_lists(vector<string>& category_names);

/**
 * Test an author's last name.
 * @return true if the name starts with A-M.
 */
bool test_author_a_m(Book& book);

/**
 * Test an author's last name.
 * @return true if the name starts with N-Z.
 */
bool test_author_n_z(Book& book);
```

# Assignment #7: Sample Solution, *cont'd*

```
int main()
{
    // Open the categories file.
    ifstream categories_file;
    categories_file.open(CATEGORIES_FILE_NAME);
    if (categories_file.fail())
    {
        cout << "Failed to open " << CATEGORIES_FILE_NAME << endl;
        return -1;
    }

    vector<string> category_names;
    string name;  // category name

    categories_file >> name;  // read the first category name

    // Loop to read the remaining category names.
    while (!categories_file.fail())
    {
        category_names.push_back(name);  // append a name to the vector
        categories_file >> name;         // read the next name
    }
```

# Assignment #7: Sample Solution, *cont'd*

```
    // Create and print the category book lists.
    vector<BookList> category_lists = make_category_lists(category_names);
    for (BookList book_list : category_lists) book_list.print();

    // Create and print the merged book list.
    BookList merged_list("MERGED", category_lists);
    merged_list.print();

    // Create and print the list of authors with last names A-M.
    BookList author_list_a_m("AUTHORS A-M", merged_list, test_author_a_m);
    author_list_a_m.print();

    // Create and print the list of authors with last names N-Z.
    BookList author_list_n_z("AUTHORS N_Z", merged_list, test_author_n_z);
    author_list_n_z.print();

    // Delete all the book lists.
    for (BookList book_list : category_lists) book_list.delete_books();
    merged_list.delete_books();
    author_list_a_m.delete_books();
    author_list_n_z.delete_books();

    return 0;
}
```

# Assignment #7: Sample Solution, *cont'd*

```
vector<BookList> make_category_lists(vector<string>& category_names)
{
    vector<BookList> category_lists;   // vector of category book lists
    string name;                       // category name

    // Loop to fill the vector of category book lists.
    for (string name : category_names)
    {
        category_lists.push_back(BookList(name));
    }

    return category_lists;
}
```

# Assignment #7: Sample Solution, *cont'd*

```
bool test_author_a_m(Book& book)
{
    return book.get_last() < "N";
}

bool test_author_n_z(Book& book)
{
    return book.get_last() >= "N";
}
```

# Exception Handling

- Exception handling is an elegant way to handle "exceptional" error situations at run time.
  - Meant to be used sparingly.
- Code (such as a function) that encounters an error situation can "throw" an exception.
- "Catch" the exception by code elsewhere (possibly in another function) that handles the exception.

# Exception Handling Example

exception1.cpp

```
int main()
{
    int value;
    cout << "Enter positive integers, 0 to quit." << endl;

    do
    {
        cout << "Value? ";
        try
        {
            cin >> value;
            if (value < 0) throw value;

            if (value > 0) cout << "You entered " << value << endl;
        }
        catch (int v)
        {
            cout << "*** Error: You entered the negative value " << v << endl;
        }

    } while (value != 0);

    cout << "Done!" << endl;
    return 0;
}
```

Try-catch block

Throw the exception **value**.

The rest of the try block is skipped whenever an exception is thrown.

Catch and handle the exception.

# Exception Classes

- You can throw a value of any type.
- You can define your own exception classes.
- A try-catch block can throw and catch multiple exceptions.

# Exception Classes Example

exception2.cpp

```
class SomeNumber
{
public:
    SomeNumber(int n) : value(n) {}
    int get_value() const { return value; }
private:
    int value;
};

class NegativeNumber : public SomeNumber
{
public:
    NegativeNumber(int n) : SomeNumber(n) {};
};

class NumberTooBig : public SomeNumber
{
public:
    NumberTooBig(int n) : SomeNumber(n) {};
};
```

Invoke the base class constructor.

# Exception Classes Example, *cont'd*

exception2.cpp

```
int main()
{
    ...
        try
        {
            cin >> value;
            if (value < 0)    throw NegativeNumber(value);
            if (value >= 10)  throw NumberTooBig(value);

            if (value > 0) cout << "You entered " << value << endl;
        }
        catch (NegativeNumber& v)
        {
            cout << "*** Error: Negative value: " << v.get_value() << endl;
        }
        catch (NumberTooBig& v)
        {
            cout << "*** Error: Value too big: " << v.get_value() << endl;
        }

    ...
}
```

# Throwing Exceptions in a Function

- A function can throw exceptions.
- The caller of the function must call the function inside a try-catch block to catch any exceptions thrown by the function.

# Throwing Exceptions in a Function, *cont'd*

exception3.cpp

```
void read_numbers() throw(NegativeNumber, NumberTooBig);

int main()
{
    try
    {
        read_numbers();
    }
    catch (NegativeNumber& v)
    {
        cout << "*** Error: Negative value: " << v.get_value() << endl;
        return -1;
    }
    catch (NumberTooBig& v)
    {
        cout << "*** Error: Value too big: " << v.get_value() << endl;
        return -2;
    }

    cout << "Done!" << endl;
    return 0;
}
```

# Throwing Exceptions in a Function, *cont'd*

exception3.cpp

```
void read_numbers() throw(NegativeNumber, NumberTooBig)
{
    int value;
    cout << "Enter positive integers < 10, 0 to quit." << endl;

    do
    {
        cout << "Value? ";
        cin >> value;
        if (value < 0)   throw NegativeNumber(value);
        if (value >= 10) throw NumberTooBig(value);

        if (value > 0) cout << "You entered " << value << endl;
    } while (value != 0);
}
```

# Quiz

- Until 7:15
- Then break until 7:30

# Random Numbers

- To generate (pseudo-) random numbers using the predefined functions, first include two library header files:

```
#include <cstdlib>
#include <ctime>
```

- "Seed" the random number generator:

```
srand(time(0));
```

  - If you don't seed, you'll always get the same "random" sequence.

# Random Numbers, *cont'd*

- Each subsequent call

```
rand();
```

returns a "random" number ≥ 0
and < **RAND_MAX**.

- Use **+** and **%** to scale to a desired number range.
  - Example: Each execution of the expression

```
rand()%6 + 1
```

returns a random number
with the value 1, 2, 3, 4, 5, or 6.

# chrono

TimeVector.cpp

```
#include <iostream>
#include <vector>
#include <chrono>

using namespace std;
using namespace std::chrono;

void initialize_vector(vector<int> v)
{
    for (int i = 0; i < 10000000; i++) v.push_back(i);
}
```

# chrono, *cont'd*

TimeVector.cpp

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <chrono>

using namespace std;
using namespace std::chrono;

long time_vector_initialization(vector<int> v, int n);

int main()
{
    vector<int> v;

    for (long n = 10000; n <= 100000000; n *= 10)
    {
        long elapsed_time = time_vector_initialization(v, n);

        cout << "Elapsed_time for " << setw(9) << n << " : "
             << setw(4) << elapsed_time << " ms" << endl;
    }
}
```

# chrono, *cont'd*

```
long time_vector_initialization(vector<int> v, int n)
{
    steady_clock::time_point start_time = steady_clock::now();
    v.clear();
    for (int i = 0; i < n; i++) v.push_back(i);
    steady_clock::time_point end_time = steady_clock::now();


    // Other options include: nanoseconds, microseconds
    long elapsed_time =
            duration_cast<milliseconds>(end_time - start_time).count();

    return elapsed_time;
}
```

# Review: Templates

- A template enables the C++ compiler to generate different versions of some code, each version of the code for a different type.
  - function templates
  - class templates
- The C++ compiler does not generate code from a template for a particular type unless the program uses the template with that type.

# The Standard Template Library (STL)

- The Standard Template Library (STL) is a collection of function and class templates for various data structures, including:
  - vector
  - stack
  - queue
  - list (doubly-linked list)
  - map (hash table)
  - set
- Example: `vector<int> v;`

# Iterators

- Iterators provide a uniform way to successively access values in a data structure.
  - Go through the values of a data structure one after another and perform some operation on each value.
- Iterators spare you from having to know how a data structure is implemented.
- Iterators are part of the STL.
- An iterator is similar to a pointer.

# A Vector Iterator

- Declare a vector iterator:

```
vector<int>::iterator it;
```

- Set the iterator to point to the first value:

```
it = v.begin();
```

- Test that the iterator hasn't gone off the end:

```
it != v.end()
```

- Access a value of the vector: `*it`
- Point to the next value: `it++`

# Vector Iterator Example

IteratorVector1.cpp

```
#include <iostream>
#include <vector>
#include <iterator>
using namespace std;

int main()
{
    vector<int> v;

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    vector<int>::iterator it;

    cout << "Test 1:";
    for (it = v.begin(); it != v.end(); it++)
    {
        cout << " " << *it;
    }
    cout << endl;
}
```

San José State University

# Kinds of Iterators

- Forward iterator
  - Use `++` to advance to the next value in the data structure.
- Bidirectional
  - Use `++` and `--` to move the iterator to the next and to the previous data values, respectively
- Random access iterator
  - `++`, `--`
  - Random access to the $n^{th}$ data value with `[n]`

# Kinds of Iterators, *cont'd*

- Constant iterator
  - Example: `vector<char>::const_iterator it;`
  - Not allowed to use the iterator to change a value.
  - Illegal use of a constant iterator: `*it = 'a';`
- Reverse iterator
  - Go through the values of a data structure in reverse order.
  - Example:

```
vector<int> v;
vector<int>::reverse_iterator it;
for (it = v.rbegin(); it != v.rend(); it++) ...
```

# Reverse Iterator Example

IteratorVector2.cpp

```
#include <iostream>
#include <vector>
#include <iterator>
using namespace std;

int main()
{
    vector<int> v;

    v.push_back(10);
    v.push_back(20);
    v.push_back(30);
    v.push_back(40);
    v.push_back(50);

    vector<int>::reverse_iterator it;

    cout << "Test 1:";
    for (it = v.rbegin(); it != v.rend(); it++)
    {
        cout << " " << *it;
    }
    cout << endl;
};
```

Note: ++

# Containers

- STL container classes are data structures that hold data.
  - Examples: lists, stacks, queues, vectors
- Each container class has its own iterator.
  - However, all the iterators have the same operators and the member functions **`begin`** and **`end`** have the same meanings.
- Sequential containers arrange their values such that there is a first value, a next value, etc. until the last value.

# The `list` Template Class

- The STL list is a doubly linked list.
  - Each element has two pointers.
  - One pointer points forward to the next element (as in a singly linked list).
  - One pointer points back to the previous element.
- You can traverse the list from either direction.
- Another pointer to manipulate when inserting or deleting an element.

# Linked List vs. Vector

- A vector has random access iterators.
- A linked list only has bidirectional iterators.
- Inserting an element into a linked list or deleting an element from a linked list are very efficient.
  - Just some pointer manipulation.
- Inserting an element into a vector or deleting an element from a vector are much less efficient.
  - Must move existing elements to make room for an insertion or to close the gap after a deletion.

# Assignment #9. Linked List vs. Vector

- Time and compare the performance of the following operations on an STL list and an STL vector:
  - inserting elements
  - searching for elements
  - accessing the *i* th element
  - deleting elements
- Use `std::chrono::steady_clock` to calculate elapsed time.