

Ch 3

Container Classes

- The Bag Class
- Programming Project: The Sequence Class
- Interactive Test Programs

The Bag Class

- Specification
- Documentation
- Client Program
- Design and Implementation
- Final Coding
- Testing
- Analysis

The Bag Class Specification

- A bag is a relaxed set: duplicates are allowed.
- a `typedef` statement will be used for genericity:
`typedef int value_type;`
- Non-negative numbers are represented by
`typedef std::size_t size_type;`

Public Member Functions

```
size_type size( ) const;  
void insert(const value_type& entry);  
size_type count(const value_type& target) const;  
bool erase_one(const value_type& target);  
size_type erase(const value_type& target);  
void operator +=(const bag& addend);
```

External Function

```
bag operator +(const bag& b1, const bag& b2);
```

Static Member Constants

A static member constant will be shared by all objects of the class. Only one copy of the data will be kept. Within the class definition:

```
static const size_type CAPACITY = 30;
```

This must also be declared in the implementation file as:

```
const bag::size_type bag::CAPACITY;
```

Class Documentation...

```
// FILE: bag1.h
// CLASS PROVIDED: bag (part of the namespace main_savitch_3)
//
// TYPEDEF and MEMBER CONSTANTS for the bag class:
// typedef ____ value_type
// bag::value_type is the data type of the items in the bag.
// It may be any of the C++ built-in types (int, char, etc.),
// or a class with a default constructor, an assignment
// operator, and operators to test for equality (x == y) and
// non-equality (x != y).
//
// typedef ____ size_type
// bag::size_type is the data type of any variable that keeps
// track of how many items are in a bag.
//
// static const size_type CAPACITY = _____
// bag::CAPACITY is the maximum number of items that a bag
// can hold.
```

...Class Documentation

```
// (MEMBER FUNCTION DOCUMENTATION)
//
// NONMEMBER FUNCTIONS for the bag class:
//   bag operator +(const bag& b1, const bag& b2)
//     Precondition: b1.size( ) + b2.size( ) <= bag::CAPACITY.
//     Postcondition: The bag returned is the union of b1 and b2.
//
// VALUE SEMANTICS for the bag class:
//   Assignments and the copy constructor may be used with
//   bag objects.
```

Bag Class Definition

```
class bag {  
    public:  
        typedef int value_type;  
        typedef std::size_t size_type;  
        static const size_type CAPACITY = 30;  
  
    bag( ) {  
        used = 0; }  
  
    size_type erase(const value_type& target);  
    bool erase_one(const value_type& target);  
    void insert(const value_type& entry);  
    void operator +=(const bag& addend);  
  
    size_type size( ) const {  
        return used; }  
    size_type count(const value_type& target) const;  
    private:  
        value_type data[CAPACITY];  
        size_type used;  
};  
  
bag operator +(const bag& b1, const bag& b2);
```

Client Program...

```
#include <iostream>
#include <cstdlib>
#include "bag1.h"

using namespace std;
using namespace main_savitch_3;

void get_ages(bag& ages);
void check_ages(bag& ages);

int main( ) {
    bag ages;
    get_ages(ages);
    check_ages(ages);
    cout << "May your family live long and prosper." << endl;
    return EXIT_SUCCESS;
}
```

...Client Program...

```
void get_ages(bag& ages) {  
    int user_input;  
  
    cout << "Type the ages in your family." << endl;  
    cout << "Type a negative number when you are done:" << endl;  
    cin >> user_input;  
  
    while (user_input >= 0) {  
        if (ages.size( ) < ages.CAPACITY)  
            ages.insert(user_input);  
        else  
            cout << "I have run out of room"  
                << "and can't add that age." << endl;  
        cin >> user_input; }  
}
```

...Client Program

```
void check_ages(bag& ages) {  
    int user_input;  
  
    cout << "Type those ages again. Press return after"  
        << "each age:" << endl;  
  
    while (ages.size( ) > 0) {  
        cin >> user_input;  
        if (ages.erase_one(user_input))  
            cout << "Yes, I've got that age and will"  
                << "remove it." << endl;  
        else  
            cout << "No, that age does not occur!" << endl; }  
}
```

Design and Implementation

- Data will be stored in a statically-allocated, partially-filled array.
- The base type of the array must have a default constructor.
- Document the class invariant in the implementation file:
 - the number of items in the bag is kept in the member variable `used`.
 - Only the logical size of the array `data[]` is relevant.

The Standard Copy Function

The standard library's copy function takes three iterators, copies from the first through the second to the third.

```
OutIter copy(InIter start, InIter end, OutIter dest);
```

To copy ten elements from array `b[]` to array `c[]`:

```
copy(b, b + 10, c);
```

Bag Class Implementation...

```
const bag::size_type bag::CAPACITY;

bag::size_type bag::erase(const value_type& target) {
    size_type index = 0;
    size_type many_removed = 0;

    while (index < used) {
        if (data[index] == target) {
            --used;
            data[index] = data[used];
            ++many_removed; }
        else
            ++index; }

    return many_removed;
}
```

...Bag Class Implementation...

```
bool bag::erase_one(const value_type& target) {  
    size_type index;  
    index = 0;  
  
    while ((index < used) && (data[index] != target))  
        ++index;  
  
    if (index == used)  
        return false;  
    else {  
        --used;  
        data[index] = data[used];  
        return true; }  
}
```

...Bag Class Implementation...

```
void bag::insert(const value_type& entry) {  
    assert(size( ) < CAPACITY);  
    data[used] = entry;  
    ++used;  
}  
  
void bag::operator +=(const bag& addend) {  
    assert(size( ) + addend.size( ) <= CAPACITY);  
    copy(addend.data, addend.data + addend.used, data + used);  
    used += addend.used;  
}
```

...Bag Class Implementation

```
bag::size_type bag::count(const value_type& target) const {
    size_type answer;

    for (size_type i = 0, answer = 0; i < used; ++i)
        if (target == data[i])
            ++answer;

    return answer;
}

bag operator +(const bag& b1, const bag& b2) {
    assert(b1.size( ) + b2.size( ) <= bag::CAPACITY);

    bag answer;

    answer += b1;
    answer += b2;
    return answer;
}
```

Testing

An object may be an argument to its own method.

This condition should be tested:

```
bag b;  
b.insert(5);  
b.insert(2);  
b += b;
```

What's wrong with this implementation?

```
void bag::operator +=(const bag& addend) {  
    size_type i;  
    for (i = 0; i < addend.used; i++) {  
        data[used] = addend.data[i];  
        ++used; }  
}
```

Analysis

Operation	Time
Constructor	$O(1)$
count()	$O(n)$
erase_one()	$O(n)$
erase()	$O(n)$
+=	$O(n)$
+	$O(n_1 + n_2)$
insert()	$O(1)$
size()	$O(1)$

The Sequence Class

- A sequence is like a bag except that its contents are ordered.
- An internal iterator will be included to provide sequential, unidirectional access. Functions include:
 - **start()** to reset the sequence,
 - **current()** to return the current item,
 - **advance()** to move to the next item,
 - **is_item()** to determine if the current item is valid.

Using the Iterator

Two private members are employed to implement the iterator: `used` and `current_index`.

```
sequence nums;  
...  
for (nums.start(); nums.is_item(); nums.advance())  
    cout << nums.current() << endl;
```

Sequence Interface

```
class sequence {  
    public:  
        typedef double value_type;  
        typedef std::size_t size_type;  
        static const size_type CAPACITY = 30;  
  
    sequence( );  
  
    void start( );  
    void advance( );  
    void insert(const value_type& entry);  
    void attach(const value_type& entry);  
    void remove_current( );  
  
    size_type size( ) const;  
    bool is_item( ) const;  
    value_type current( ) const;  
    private:  
        value_type data[CAPACITY];  
        size_type used;  
        size_type current_index;  
};
```

Sequence Class Invariants

- The number of items in the sequence is stored in the member variable `used`.
- Items are stored in their sequence order from `data[0]` to `data[used-1]`. The remainder of the array's contents are ignored.
- If there is a current item, then it lies in `data[current_index]`; if there is no current item, then `current_index` equals `used`.

Interactive Testing

- The test program begins with the instantiation of one or more objects.
- The remainder of the program consists of a loop that continues as long as the tester wishes.
 - A menu is presented. Each choice has a brief description along with a unique character to represent it.
 - The user's selection from the menu is read from standard input.
 - The appropriate action is taken on the object(s) and the results are displayed.

Upshifting User Input

To simplify menu choice processing, use the `toupper()` function declared in `<cctype>`.

```
choice = toupper(get_user_command());
```

Sequence Test Program...

```
void print_menu( ) {
    cout << endl;
    cout << "The following choices are available: " << endl;
    cout << " !   Activate the start( ) function" << endl;
    cout << " +   Activate the advance( ) function" << endl;
    cout << " ?   Print the result from the is_item( ) function" << endl;
    cout << " C   Print the result from the current( ) function" << endl;
    cout << " P   Print a copy of the entire sequence" << endl;
    cout << " S   Print the result from the size( ) function" << endl;
    cout << " I   Insert a new number with the insert(...) function" << endl;
    cout << " A   Attach a new number with the attach(...) function" << endl;
    cout << " R   Activate the remove_current( ) function" << endl;
    cout << " Q   Quit this test program" << endl;
}
```

...Sequence Test Program...

```
char get_user_command( ) {  
    char command;  
    cout << "Enter choice: ";  
    cin >> command;  
    return command;  
}  
  
void show_sequence(sequence display) {  
    for (display.start( ); display.is_item( ); display.advance( ))  
        cout << display.current( ) << endl;  
}  
  
double get_number( ) {  
    double result;  
    cout << "Please enter a real number for the sequence: ";  
    cin >> result;  
    cout << result << " has been read." << endl;  
    return result;  
}
```

...Sequence Test Program...

```
int main( ) {  
    sequence test;  
    char choice;  
  
    cout << "I have initialized an empty "  
          << "sequence of real numbers." << endl;  
  
    do {  
        print_menu( );  
        choice = toupper(get_user_command( ));  
        switch (choice) {  
            case '!':  
                test.start( );  
                break;  
            case '+':  
                test.advance( );  
                break;  
            case '?':  
                if (test.is_item( ))  
                    cout << "There is an item." << endl;  
                else  
                    cout << "There is no current item." << endl;  
                break;  
        }  
    } while (choice != 'q');
```

...Sequence Test Program...

```
case 'C':
    if (test.is_item( ))
        cout << "Current item is: " << test.current( ) << endl;
    else
        cout << "There is no current item." << endl;
    break;
case 'P':
    show_sequence(test);
    break;
case 'S':
    cout << "Size is " << test.size( ) << '.' << endl;
    break;
case 'I':
    test.insert(get_number( ));
    break;
```

...Sequence Test Program

```
case 'A':
    test.attach(get_number( ));
    break;
case 'R':
    test.remove_current( );
    cout << "The current item has been removed." << endl;
    break;
case 'Q':
    cout << "Ridicule is the best test of truth." << endl;
    break;
default:
    cout << choice << " is invalid." << endl; } }

while ((choice != 'Q'));

return EXIT_SUCCESS;
}
```

Complete Project 8 from Chapter 3 of the text. Directions are on page 142.

- Name your **class** `keyed_bag`. You will submit its definition in `keyed_bag.h` and the implementation in `keyed_bag.cpp`.
- Set the data type of the bag as

```
typedef double value_type;
```

- Include member functions whose prototypes are:

```
keyed_bag();  
void insert(const value_type& entry, int key);  
bool erase(int key);  
value_type get(int key) const;  
size_type size() const;  
size_type count(const value_type& target) const;  
bool has_key(int key) const;
```

- If an attempt is made to `insert()` an entry with a duplicate key, the existing entry should be replaced by one with the new value.
- Create a **namespace** for your class: `cis11`.
- Be sure to include appropriate documentation including
 - class invariant,
 - value semantics,
 - description of functionality,
 - precondition(s) and
 - postcondition(s).
- For some design suggestions follow this [link](#).
- Test your class thoroughly with your own interactive test program similar to the one presented on pp. 133-135 of the text. Name the program `keyed_bag_test.cpp`.

- Hold the keys and the data in two parallel arrays or linked lists, or
- Place the keys and data in a single two-dimensional array, or
- Create an array or linked list of record objects. Here is a sample **class** illustrating the technique (notice that there is an anonymous nested **class**):

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

class Test {
public:
    typedef double value_type;
    void setKey(int r, int k) {
        record[r].key = k; }
    int getKey(int r) const {
        return record[r].key; }
    void setData(int r, value_type d) {
        record[r].data = d; }
    value_type getData(int r) const {
        return record[r].data; }
private:
    class {
    public:
        int key;
        value_type data; }
    record[10];
};

int main() {
    Test t;
    t.setKey(0, 2000);
    t.setData(0, 2);
    cout << "Key: " << t.getKey(0) << " Data: " << t.getData(0) << endl;
    cout << "Key: " << t.getKey(9) << " Data: " << t.getData(9) << endl;
    //cout << t.record[0].key; Can't do that!
}
```