

Ch 4

Pointers and Dynamic Arrays

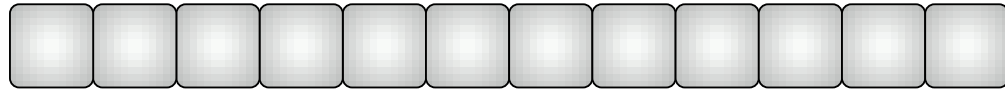
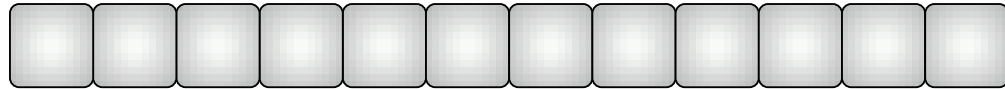
- Pointers and Dynamic Memory
- Pointers and Arrays as Parameters
- The Bag Class with a Dynamic Array
- Prescription for a Dynamic Class
- Programming Project: The String Class
- Programming Project: The Polynomial Class

Pointers and Dynamic Memory

The Legacy from Assembly language and C

```
int i = 6;
```

Memory in C

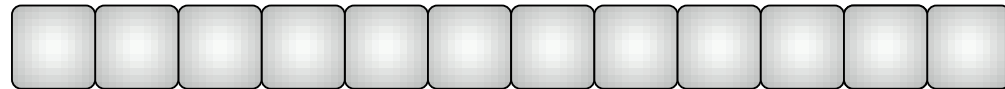
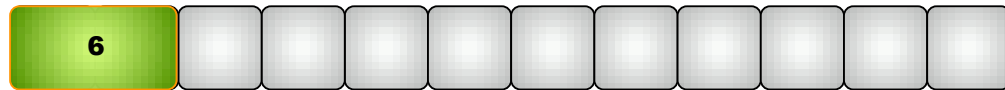


```
int i = 6;
```

Memory in C cont'd



i

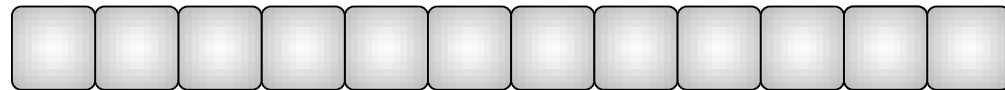
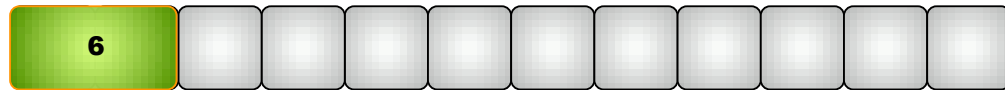


```
int i = 6;  
char c = 'A';
```

Memory in C cont'd

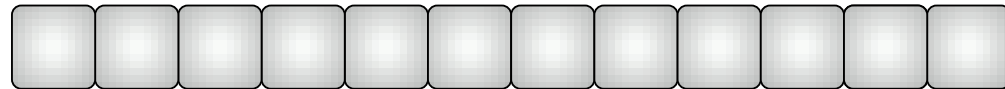
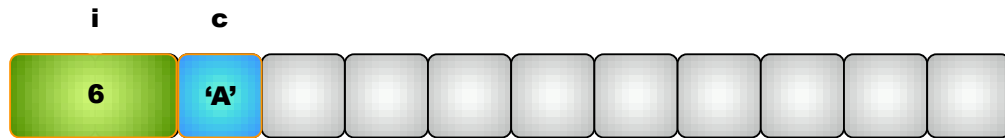


i



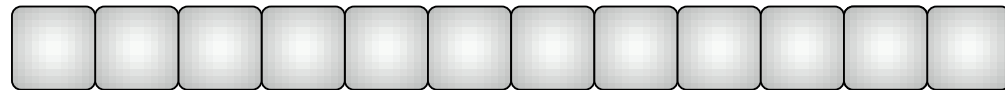
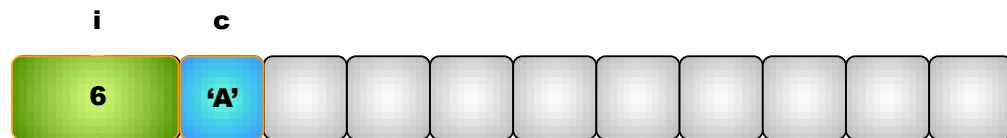
```
int i = 6;  
char c = 'A';
```

Memory in C cont'd



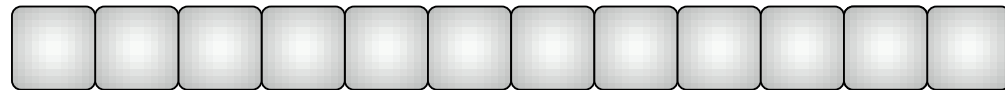
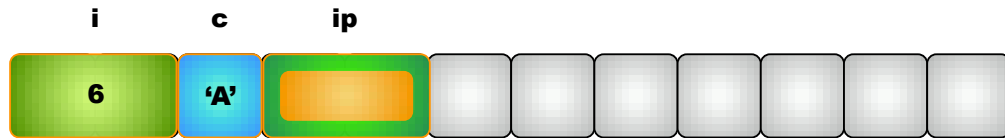
```
int i = 6;  
char c = 'A';  
int* ip;
```

Memory in C cont'd



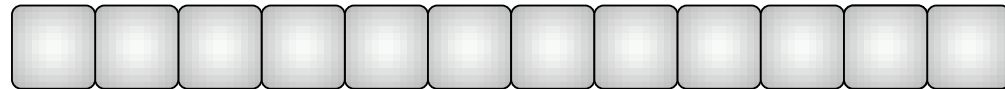
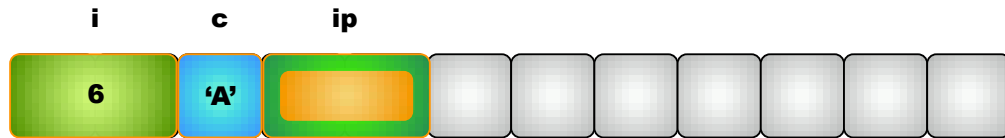
```
int i = 6;  
char c = 'A';  
int* ip;
```

Memory in C cont'd



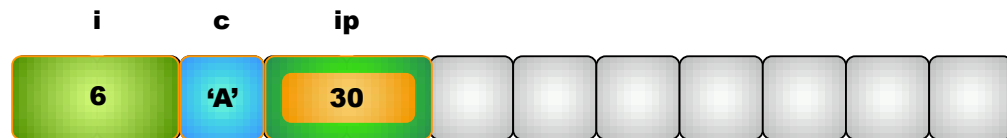
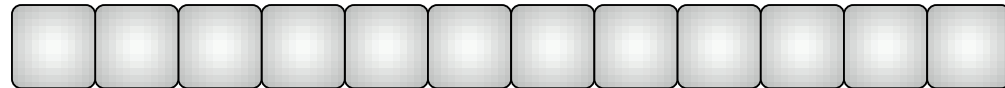
Memory in C cont'd

```
int i = 6;  
char c = 'A';  
int* ip;  
ip = &i;
```



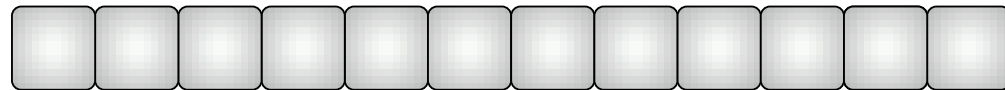
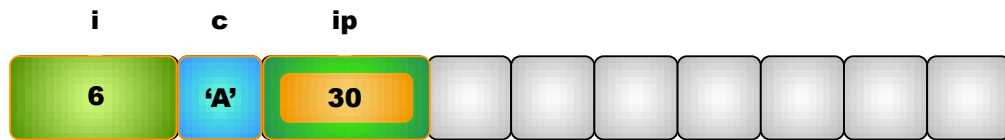
Memory in C cont'd

```
int i = 6;  
char c = 'A';  
int* ip;  
ip = &i;
```



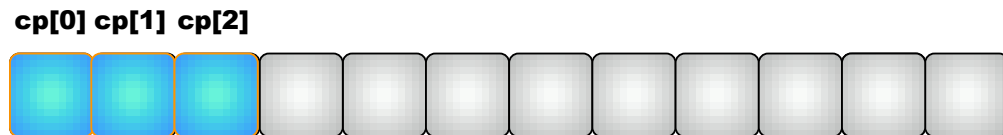
Memory in C cont'd

```
int i = 6;  
char c = 'A';  
int* ip;  
ip = &i;  
char* cp = new char[3];
```



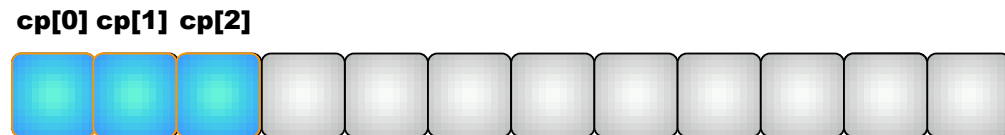
Memory in C cont'd

```
int i = 6;  
char c = 'A';  
int* ip;  
ip = &i;  
char* cp = new char[3];
```



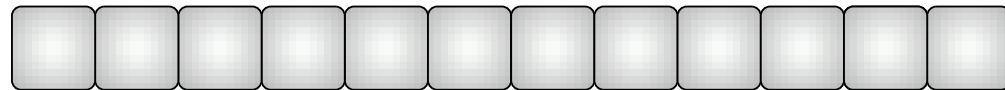
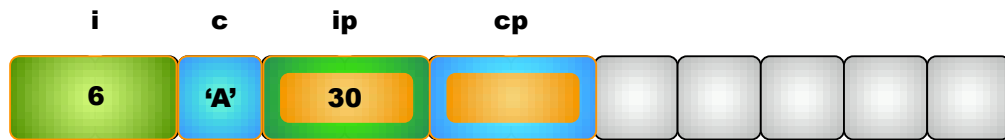
Memory in C cont'd

```
int i = 6;  
char c = 'A';  
int* ip;  
ip = &i;  
char* cp = new char[3];  
delete[] cp;
```



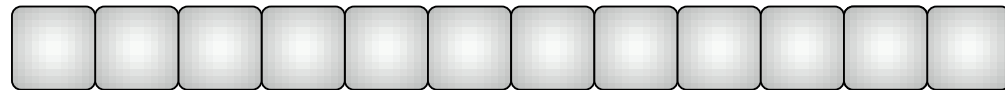
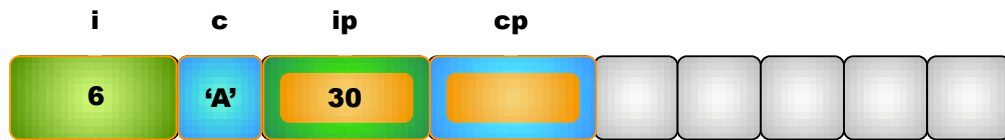
Memory in C cont'd

```
int i = 6;  
char c = 'A';  
int* ip;  
ip = &i;  
char* cp = new char[3];  
delete[] cp;
```



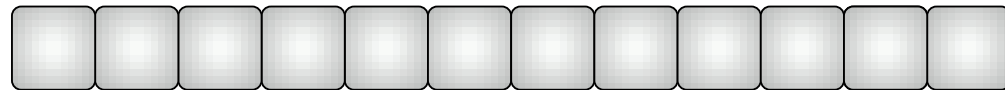
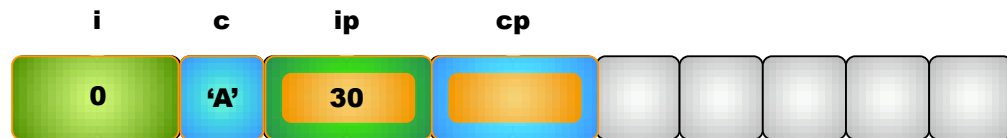
Memory in C cont'd

```
int i = 6;  
char c = 'A';  
int* ip;  
ip = &i;  
char* cp = new char[3];  
delete[] cp;  
*ip = 0;
```



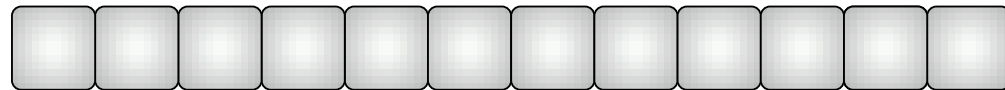
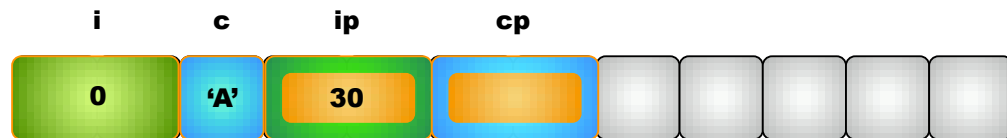
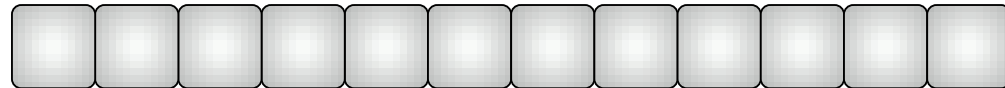
Memory in C cont'd

```
int i = 6;  
char c = 'A';  
int* ip;  
ip = &i;  
char* cp = new char[3];  
delete[] cp;  
*ip = 0;
```



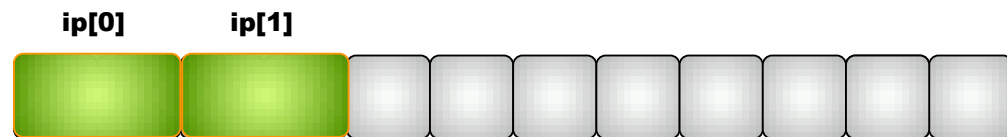
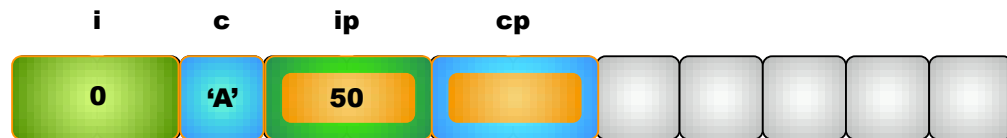
Memory in C cont'd

```
int i = 6;  
char c = 'A';  
int* ip;  
ip = &i;  
char* cp = new char[3];  
delete[] cp;  
*ip = 0;  
ip = new int[2];
```



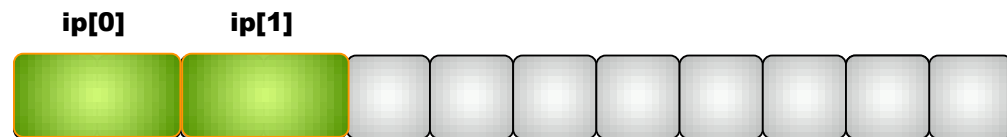
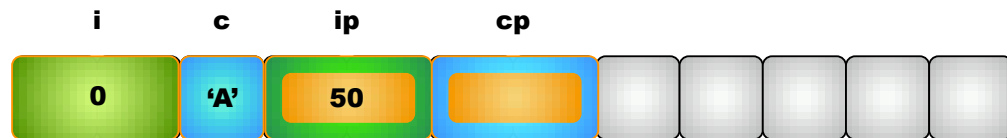
Memory in C cont'd

```
int i = 6;  
char c = 'A';  
int* ip;  
ip = &i;  
char* cp = new char[3];  
delete[] cp;  
*ip = 0;  
ip = new int[2];
```



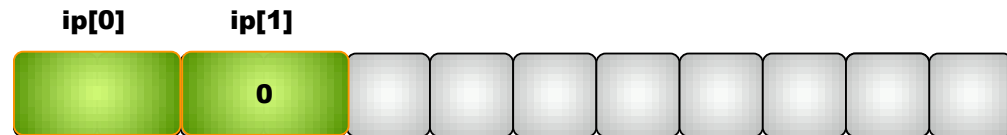
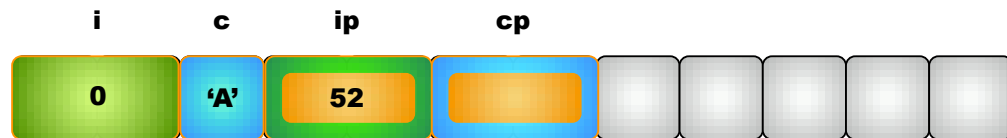
Memory in C cont'd

```
int i = 6;  
char c = 'A';  
int* ip;  
ip = &i;  
char* cp = new char[3];  
delete[] cp;  
*ip = 0;  
ip = new int[2];  
++*ip = 0;
```



Memory in C cont'd

```
int i = 6;  
char c = 'A';  
int* ip;  
ip = &i;  
char* cp = new char[3];  
delete[] cp;  
*ip = 0;  
ip = new int[2];  
++*ip = 0;
```



Dynamic Allocation Notes

- The `new` operator throws a `bad_alloc` exception if it fails. The value of the `new` expression is then undefined.
- Pointers used in a `delete` expression are undefined after the expression is evaluated.
- It is the programmer's responsibility to avoid dereferencing undefined (wild) pointers.

Pointers and Arrays as Parameters

- When a parameter is a pointer, the function may change the value pointed to, but not the address.
- Array parameters are declared with brackets after the parameter name. All but the first dimension must be declared, as in

```
void foo(int twoDim[][4]);
```

- The `const` modifier may be used

```
bool foo(const int* ip);
```

```
double bar(const double data[]);
```

Pointer Reference Parameters

Sometimes a function will need to change the value of a pointer passed to it. This may be accomplished by passing a reference to the pointer

```
void allocate(double*& p, size_t& n);
```

Programming Example

Using a Dynamic Array

`dynademo.cxx`

dynademo.cxx...

```
void allocate_doubles(double*& p, size_t& n);
void fill_array(double data[ ], size_t n);
double average(const double data[ ], size_t n);
void compare(const double data[ ], size_t n, double value);

int main( ) {
    double *numbers;
    size_t array_size;
    double mean_value;

    cout << "This program will compute the average of some numbers. The\n";
    cout << "numbers will be stored in an array of doubles that I allocate.\n";

    allocate_doubles(numbers, array_size);
    fill_array(numbers, array_size);

    mean_value = average(numbers, array_size);
    cout << "The average is: " << mean_value << endl;
    compare(numbers, array_size, mean_value);
    cout << "This was a mean program.";
    return EXIT_SUCCESS;
}
```

...dynademo.cxx...

```
void allocate_doubles(double*& p, size_t& n) {  
    cout << "How many doubles should I allocate?" << endl;  
    cout << "Please type a positive integer answer: ";  
    cin >> n;  
    p = new double[n];  
}  
  
void fill_array(double data[ ], size_t n) {  
    size_t i;  
    cout << "Please type " << n << " double numbers: " << endl;  
    for (i = 0; i < n; ++i)  
        cin >> data[i];  
}
```

...dynademo.cxx

```
void compare(const double data[ ], size_t n, double value) {
    size_t i;
    for (i = 0; i < n; ++i) {
        cout << data[i];
        if (data[i] < value)
            cout << " is less than ";
        else if (data[i] > value)
            cout << " is more than ";
        else
            cout << " is equal to ";
        cout << value << endl; }
}

double average(const double data[ ], size_t n) {
    size_t i;
    double sum;
    assert(n > 0);
    sum = 0;
    for (i = 0; i < n; ++i)
        sum += data[i];
    return (sum / n);
}
```

The Bag Class with a Dynamic Array

Run-time vs. Compile-time Allocation

Class Invariants

- The number of items in the bag is in the member variable **used**.
- The actual items of the bag are stored in a partially-filled array. It is a dynamic array, pointed to by the member variable **data**.
- The total size of the dynamic array is in the member variable **capacity**.

Programming Considerations

- Member functions must allocate dynamic memory as needed.
- Documentation about possible memory failure should be provided.
- To preserve the value semantics the copy constructor and assignment operator(s) should be overloaded.
- A destructor method must be written.

The Destructor

- The destructor's name is a tilde (~) followed by the name of the class.
- Destructors have no return type and take no arguments.
- To permit inheritance they must be declared **virtual**.
- The destructor is automatically invoked when an object created on the stack goes out of scope and when one created on the heap is deleted.
- It's the destructor's job to release resources captured by the object.

Testing the Destructor

```
class deletable {  
  public:  
    deletable(string str) : s(str) {  
      cout << s << "I'm here!" << endl; }  
    ~deletable() {  
      cout << s << "I'm outa here!" << endl; }  
  private:  
    string s;  
};
```

```
int main() {  
  cout << "{" << endl;  
  deletable d1("  On the stack, d1: ");  
  deletable* d2 = new deletable("  On the heap,  d2: ");  
  {  
    cout << "  {" << endl;  
    deletable* d3 = new deletable("    On the heap, d3: ");  
    delete d3;  
    cout << "  }" << endl;  
  }  
  cout << "}" << endl;  
  return EXIT_SUCCESS;  
}
```

```
{  
  On the stack, d1: I'm here!  
  On the heap,  d2: I'm here!  
  {  
    On the heap, d3: I'm here!  
    On the heap, d3: I'm outa here!  
  }  
}  
On the stack, d1: I'm outa here!
```

The Revised Bag Class

Class Definition

bag2.h

bag2.h

```
class bag {
public:
    typedef int value_type;
    typedef std::size_t size_type;

    static const size_type DEFAULT_CAPACITY = 30;

    bag(size_type initial_capacity = DEFAULT_CAPACITY);

    bag(const bag& source);
    void operator =(const bag& source);
    ~bag( );

    void reserve(size_type new_capacity);
    bool erase_one(const value_type& target);
    size_type erase(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;
    size_type used;
    size_type capacity;
};

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

Implementation: The Assignment Operator

The assignment operator is overloaded. It is similar to the copy constructor, but differs in some ways:

- The copy constructor starts from scratch.
- If the current array isn't big enough to receive the data from the source, a new one must be allocated and the old one destroyed after copying.
- A check for self-assignment should be made.

Implementation: Memory Allocation

- The documentation should indicate which functions may throw a `bad_alloc` exception.
- Check the class invariants before using the `new` operator.
- Never apply `delete` before using `new`.
- Constructors are usually exempt from these considerations.

The Revised Bag Class

Class Implementation

bag2.cxx

bag2.cxx...

```
const bag::size_type bag::DEFAULT_CAPACITY;

bag::bag(size_type initial_capacity) {
    data = new value_type[initial_capacity];
    capacity = initial_capacity;
    used = 0;
}

bag::bag(const bag& source) {
    data = new value_type[source.capacity];
    capacity = source.capacity;
    used = source.used;
    copy(source.data, source.data + used, data);
}

void bag::operator =(const bag& source) {
    value_type* new_data;
    if (this == &source)
        return ;
    if (capacity != source.capacity) {
        new_data = new value_type[source.capacity];
        delete [ ] data;
        data = new_data;
        capacity = source.capacity; }
    used = source.used;
    copy(source.data, source.data + used, data);
}

bag::~bag( ) {
    delete [ ] data;
}
```

...bag2.cxx...

```
void bag::reserve(size_type new_capacity) {
    value_type *larger_array;
    if (new_capacity == capacity)
        return ;
    if (new_capacity < used)
        new_capacity = used;
    larger_array = new value_type[new_capacity];
    copy(data, data + used, larger_array);
    delete [ ] data;
    data = larger_array;
    capacity = new_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;
}
```

...bag2.cxx...

```
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;
    --used;
    data[index] = data[used];
    return true;
}

void bag::insert(const value_type& entry) {
    if (used == capacity)
        reserve(used + 1);
    data[used] = entry;
    ++used;
}
```

...bag2.cxx

```
void bag::operator +=(const bag& addend) {  
    if (used + addend.used > capacity)  
        reserve(used + addend.used);  
    copy(addend.data, addend.data + addend.used, data + used);  
    used += addend.used;  
}  
  
bag::size_type bag::count(const value_type& target) const {  
    size_type answer = 0, i;  
    for (i = 0; i < used; ++i)  
        if (target == data[i])  
            ++answer;  
    return answer;  
}  
  
bag operator +(const bag& b1, const bag& b2) {  
    bag answer(b1.size( ) + b2.size( ));  
    answer += b1;  
    answer += b2;  
    return answer;  
}
```

Programming Project

The String Class

ASCIIIZ or C-strings

- They are stored as an array of **char**.
- A terminating **NULL** or '**\0**' is used to signal the logical end of the array.
- Functions that process ASCIIIZ strings rely on the **NULL** termination convention.
- C-strings' maximum logical size is always one less than their physical size.

C-string Declarations

There are special initialization provisions:

```
char s[11];  
char s[LOGICAL_MAX + 1];  
char s[] = "Peace";  
char s[20] = "Happiness";  
char s[] = {'a', 'b', 'c', '\0'};  
//! char s[] = {'a', 'b', 'c'};
```

C-string Functions

- Insertion and extraction operators are overloaded. Extraction skips leading white space and stops after the last non-whitespace character.
- Functions substitute for other operators. Their prototypes are in `<cstring>`.

```
char* strcpy(char d[], const char s[]);  
char* strcat(char d[], const char s[]);  
  
char* strncpy(char d[], const char s[], int n);  
char* strncat(char d[], const char s[], int n);  
  
int strlen(const char s[]);  
  
int strcmp(const char s1[], const char s2[]);
```

mystring.h

```
class string {
public:
    string(const char str[ ] = "");
    string(const string& source);
    void operator =(const string& source);
    ~string( );

    void operator +=(const string& addend);
    void operator +=(const char addend[ ]);
    void operator +=(char addend);
    void reserve(size_t n);

    size_t length( ) const {
        return current_length; }
    char operator [ ](size_t position) const;

    friend ostream& operator <<(ostream& outs, const string& source);
    friend bool operator ==(const string& s1, const string& s2);
    friend bool operator !=(const string& s1, const string& s2);
    friend bool operator >=(const string& s1, const string& s2);
    friend bool operator <=(const string& s1, const string& s2);
    friend bool operator > (const string& s1, const string& s2);
    friend bool operator < (const string& s1, const string& s2);
private:
    char *sequence;
    size_t allocated;
    size_t current_length;
};

string operator +(const string& s1, const string& s2);
istream& operator >>(istream& ins, string& target);
void getline(istream& ins, string& target, char delimiter);
```

Invariant for the String Class

- The string is stored as a null-terminated string in the dynamic array that `characters` points to.
- The total length of the dynamic array is stored in the member `allocated`.
- The total number of characters prior to the null character is stored in `current_length`, which is always less than `allocated`.

Implementation Details

The extraction operator needs to skip over white space to provide standard behavior.

```
while (inStream && isspace(inStream.peek()))  
    inStream.ignore();
```

Any function that expects a string object will work with other types as long as an appropriate constructor is provided.

```
string(const char s[]);
```

```
match("first name", meFirst, youFirst);
```

str_demo.cxx

```
int main( ) {
    const string BLANK(" ");
    string me_first("Demo"), me_last("Program");
    string you_first, you_last, you;

    cout << "What is your first name? ";
    cin >> you_first;
    match("first name", me_first, you_first);

    cout << "What is your last name? ";
    cin >> you_last;
    match("last name", me_last, you_last);

    you = you_first + BLANK + you_last;
    cout << "I am happy to meet you, " << you << "." << endl;
    return EXIT_SUCCESS;
}

void match(const string& variety,
           const string& mine,
           const string& yours) {
    if (mine == yours)
        cout << "That is the same as my " << variety << "!" << endl;
    else
        cout << "My " << variety << " is " << mine << "." << endl;
}
```

Project C

Complete Project 2e from Chapter 4 of the text. Directions are on page 209.

- Name your **class** `keyed_bag2`. You will submit its definition in `keyed_bag2.h` and the implementation in `keyed_bag2.cpp`.
- Set the data type of the bag as

```
typedef double value_type;
```

- Include member functions whose prototypes are:

```
keyed_bag2();  
keyed_bag2(const keyed_bag2& source);  
~keyed_bag2();  
keyed_bag2& operator =(const keyed_bag2& source);  
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.
- Do not create a **namespace** for your **class**.
- 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. You will not submit the test program.

```

class Test {
    public:
        typedef double value_type;
        typedef int key_type;
        Test(size_t n) {
            record = new record_type[n]; }

        void setKey(size_t r, key_type k) {
            record[r].key = k; }
        void setData(size_t r, value_type d) {
            record[r].data = d; }

        key_type getKey(size_t r) const {
            return record[r].key; }
        value_type getData(size_t r) const {
            return record[r].data; }

    private:
        class record_type {
            public:
                key_type key;
                value_type data; }
        * record;
};

```