Data Structures and Problem Solving with C++
Second Edition
Instructors Resource Manual

Mark Allen Weiss
Florida International University

Addison-Wesley Publishing Company, Inc.
Menlo Park, California • Reading, Massachusetts • New York • Don Mills, Ontario
Wokingham, U.K. • Amsterdam • Bonn • Paris • Milan • Madrid • Sydney
Singapore • Tokyo • Seoul • Tapei • Mexico City • San Juan, Puerto Rico

Preface

This *Instructor's Resource Manual* provides additional material for instructors to use in conjunction with *Data Structures, and Problem Solving with C++, second edition.*

Each chapter in the text has a corresponding chapter in this manual that consists of:

- A section on the important concepts and how to teach them
- Solutions to many of the *In Short* and *In Theory* questions, as well as some comments for some of the *In Practice* questions
- Multiple choice questions

I have attempted to avoid redundancy. As a result, common errors, which are already listed in the text for each chapter, are not repeated. You should be sure to review these errors with your students. Also, I have not repeated material already stated in the book's preface.

A minimal set of multiple choice questions is provided. It is easy to generate additional questions from both these multiple choice questions (for example, replace *inorder* with *postorder*, and you have a different question in Chapter 18) and from some of the in chapter exercises. It is also a simple matter to design true/false questions or fill in questions based on what is provided. My own preference is to give three types of questions on an exam: long answer (write a code fragment...), simulation (show the data structure after the following operations...), and questions. Of course if you have very large sections, grading this might be too time consuming.

As mentioned in the textbook, the source code is available online. I have not included any additional directly compilable code in this supplement.

Appendix A provides two syllabii: One for the separation approach, and one for the traditional approach. Eight assignments are described in Appendix B. Many many more are sugested as *Programming Projects* throughout the text.

E-mail

Please send comments and error reports to `weiss@fiu.edu`. Eventually, my home page `http://www.fiu.edu/~weiss` will maintain an updated error list and additional notes.

Chapter 1

Pointers, Arrays, and Structures

1.1 Key Concepts and How To Teach Them

This chapter introduces several concepts:

- basic arrays (first-class arrays, using `vector`)
- basic strings (using `string`)
- pointers
- dynamic allocation
- reference variables and parameter passing mechanisms
- structures

Depending on the students' background, some or even all of this chapter could be skipped, but I recommend at least a quick review of the chapter in all circumstances. Students who have not had C or C++ should go through the entire chapter slowly. It is easy for students to think they understand the material in this chapter based on an earlier course, and my experience is that they do not.

1.1.1 Arrays

Modern C++ arrays use the standard `vector` class and avoids the C-style array. I like to explain the idea of using a library class, so as to lead in to Chapter 2. Remind students that array indexing starts at `0` and goes to `size()-1`. Off-by-one errors are very common; make sure the students are aware of these possibilities, and that the standard vector is not bounds checked (we write a better one in Chapter 3). Explain parameter passing. Finally discuss the `push_back` idea. I have found that `push_back` is easy for students to understand.

1.1.2 Strings

There is not much to do here; avoid C-style strings. You may prefer to do strings before arrays.

1.1.3 Pointers

Draw the usual memory pictures and emphasize that a pointer object is an object that stores a memory address. Go through as many examples as you can to distinguish between accessing a pointer and dereferencing it. *Note*: The `NULL` constant is defined in several header files, including `stdlib.h`.

1.1.4 Dynamic Allocation

This is here to avoid forward references in the text. You may prefer to delay this material until linked lists are discussed. Or you can preview the idea of `new` and `delete`, and explain the problems of stale pointers and double-deletion. Explain the term *memory leak*.

1.1.5 Reference Variables and Parameter Passing Mechanisms

The key topic here is the distinction between call-by-value, call-by-reference, and call-by-constant reference. Emphasize over and over again, that there are really three forms of parameter passing and that the `const` is not merely window dressing. Here are my rules:

- Call by value: used for IN parameters for built-in types
- Call by constant reference: used for IN parameters for class types
- Call by reference: used for IN OUT parameters

Many students insists on passing `int` objects by constant reference; this is wrong! It induces the overhead of pointer indirection when the reference is accessed inside the function.

Many students get confused about passing pointers. When a pointer object is passed, the value of the object is an address. Passing a pointer by reference means that *where* the pointer points at could change. This is useful for resizing dynamically allocated C-style arrays and also in binary tree updates.

1.1.6 Structures

This is a quickie opener to the class discussion. A C-style structure achieves the grouping of data, but does not provide for information hiding or encapsulation of functionality. Even so, some issues become evident and are worth discussing:

1. Structures should be passed either by reference or constant reference.
2. Deep vs. shallow copy.
3. Quick introduction to the linked list, C-style.

1.2 Solutions To Exercises

In Short

1.1 Pointers can be declared and initialized, they can be assigned to point at an object, they can be dereferenced, they can be involved in arithmetic. The address-of operator can be applied to a pointer in the same manner as any other object.

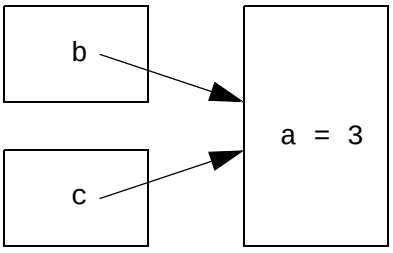
1.2 (a) Yes; (b) Both have the same value as `A`; (c) `*ptrPtr=&b;` (d) No; these objects have different types.

1.3 (a) Yes as long as `x` is an object. (b) No, because `x` might not be a pointer.

1.4 (a) the address where `a` is stored; (b) the value of `a` (5); (c) 1; (d) this is a type mismatch, and if accepted by the compiler is most likely 0; (e) the address where `ptr` is stored; (f) illegal, because `a` is not a pointer; (g) the value of `a` (5); (h) the value of `a` (5).

1.5 (a) `a` is a member of type `int` in struct `S` and `b` is a member of type pointer to `S` in struct `S`; (b) `z` is of type `S`; (c) `x` is of type pointer to `S`; (d) `y` is of type array of 10 `S`; (e) `u` is of type array of 10 pointer to `S`; (f) `x->a` is of type `int`; (g) `x->b` is of type pointer to `S`; (h) `z.b` is of type pointer to `S`; (i) `z.a` is of type `int`; (j) `*z.a` is illegal because `z.a` is not a pointer type; (k) `(*z).a` is illegal because `z` is not a pointer type; (l) (this question should not be here) `x->b-z.b` is a subtraction of pointers and is thus of type `int`; (m) `y->a` is illegal; (n) `y[1]` is of type `S`; (o) `y[1].a` is of type `int`; (p) `y[1].b` is of type pointer to `S`; (q) `u[2]` is of type pointer to `S`; (r) `*u[2]` is of type `S`; (s) `u[2]->a` is of type `int`; (t) `u[2]->b` is of type pointer to `S`; (u) `u[10]` is of type pointer to `S` but is past the declared bounds of `u`; (v) `&z` is of type pointer to `S`; (w) `&x` is of type pointer to pointer to `S`; (x) `u` is of type array of pointer to `S`; (y) `y` is of type array of `S`.

1.6 The picture below reflects `a`, `b`, and `c` after the declarations. The statement `b=5` changes `a` to 5 and then `c=2` changes `a` to 2.



1.7 This is perfectly legal. However if the `const` is moved from the second declaration to the first, then the declaration and initialization of `b` would be illegal.

1.8 `/*` begins a comment.

1.3 Exam Questions

1.1. For the declarations below, which statement is illegal?

```
int *ptr1;
int *ptr2;
int a = 5;
```

a. `ptr1 = ptr2;`
b. `ptr1 = a;`
c. `ptr1 = &a;`
d. `*ptr1 = *ptr2;`
e. `*ptr1 = a;`

1.2. For the declarations below, which expression is true if `ptr1` and `ptr2` point at the same object?

```
int *ptr1;
int *ptr2;
```

a. `ptr1 == ptr2`
b. `*ptr1 == *ptr2`
c. `*(ptr1 == *ptr2)`
d. `&ptr1 == &ptr2`
e. None of the above

1.3. For the declaration below, what is the type of `*a`?

```
const int *a;
```

a. `int`
b. `const int`
c. `int *`
d. `const int *`
e. none of the above

1.4. A *memory leak* occurs when:

a. A local array is deleted.
b. A dynamically allocated object is deleted.
c. A dynamically allocated object is no longer referenced.
d. Two pointers point at the same object.
e. A dynamically allocated object is deleted twice.

1.5. Which of the following parameter passing mechanisms can be used to alter a parameter?

a. Call by value
b. Call by reference
c. Call by constant reference
d. All of the above
e. None of the above

1.6. A *shallow copy* refers to

a. the copying of small objects
b. the copying of pointers
c. the copying of objects that are being pointed at
d. the copying of basic types, such as integers
e. call by value

1.7. *Exogenous data* is

a. a small member of a structure
b. a large member of a structure
c. an object that is not part of the structure, but is pointed at by the structure
d. a global variable
e. the entire structure

1.8. If `f` is a member of structure `S`, and `p` is of type pointer to `S`, then which expression must be legal?

a. `p.f`
b. `p->f`
c. `*p.f`
d. `s.f`
e. More than one of the above

1.9. What is the result of the following?

```
int x = 5;
int & ref = x;
ref++;
```

a. It increments `x`
b. It increments `ref`
c. It increments `*ref`
d. It increments `&ref`
e. It is illegal

1.10. What is the result of the following?

```
int x = 5;
int *ptr = &x;
int * & ref = ptr;
*ref++;
```

a. It increments `ptr`
b. It increments `ref`
c. It increments `x`
d. It increments `&ref`
e. It is illegal

Answers to Exam Questions

1. B
2. A
3. B
4. C
5. B
6. B
7. C
8. B
9. A
10. A

Chapter 2

Objects and Classes

2.1 Key Concepts and How To Teach Them

Students who have not had C++, with a description of class design, will need to go through this chapter in its entirety. Students who have may want to quickly review the chapter.

This chapter introduces the general concept of encapsulation and information hiding, but is geared towards practical use of C++ with lots of emphasis on designing classes and syntax. You may want to have the students bring a copy of the code to class so you can avoid rewriting the classes. You also may want to devote an entire class to reviewing the common error section at the end of the chapter. The basic concepts are:

- the class construct
- public and private sections
- interface vs. implementation
- constructors, destructors, copy constructor, and copy assignment operator
- `const` member functions
- `this` pointer
- operator overloading
- type conversions
- I/O
- static class members
- enum, and constant class members
- `string` class implementation

2.1.1 The `class` Construct

Describe how the structure achieves the grouping of data members, and that C++ extends it to allow information hiding and encapsulation of functionality. The basic mechanism for the latter is to allow functions to be data members. You can

illustrate this with `IntCell`. Add the private and public after discussing it.

2.1.2 Public and Private Sections

This seems to be a relatively easy topic. Explain that everything in the private section is inaccessible to non-class routines. Continue with the `IntCell` example.

2.1.3 Interface vs. Implementation

Explain that the implementation of member functions cannot be practical for large classes, so we have to separate. Continue with `IntCell` class. At this point you can describe the entire layout for separate compilation (if appropriate), and remark about the `#ifndef/#endif` trick.

2.1.4 Constructors, Destructors, Copy Constructor, and Copy Assignment `operator=`

Explain that in addition to particular member functions, every class has implicit characteristics such as creation, destruction, and copying. The mechanics in C++ are complex because of default scenarios and syntax.

Describe the constructor first. Show how a declaration is matched by a constructor. Warn about several potential errors (see common errors). Next show the copy constructor. Explain over and over again the difference between the copy constructor and `operator=`: The copy constructor is called when a new object is created, while `operator=` copies into an already existing object. Emphasize the signature of the copy constructor (it takes a constant reference). Finally, describe the destructor. It is enough to say at this time that it cleans up things.

Remark about the defaults: The rules are uniform, and boils down to member-by-member application. If no constructor is provided, a default zero-parameter constructor is generated. The copy constructor is not counted in deciding if a default zero-parameter constructor is generated.

Remark about initializer lists. These are very important for more complex cases, and must be used for constant members or reference members.

Explain that copy constructors are needed for call by value and return by value. Both of these mechanisms generate a hidden temporary by using the copy constructor (and then eventually call a destructor for the temporary). Later on, return types are discussed (when operator overloading is examined).

Explain that if the destructor is non-trivial, then the copy constructor and copy assignment operator must almost certainly not use the default.

2.1.5 `const` Member Functions

This is another often overlooked task. Constant member functions promise not to change the state of an object. Consider carefully whether each member function should be a constant member function. This is not merely an afterthought. Provide an example where it matters.

2.1.6 `this` Pointer

The `this` pointer enters into place when assignment operators are overloaded. It is used in both the return value (`*this`) and in testing for aliasing (`*this==&rhs`). Emphasize the distinction between `this` and `*this`. Here are examples where aliasing can hurt:

1. Copying from one file to another. If file names are the same, the file is probably clobbered unless a check is performed first.
2. `strcat(a,a)` fails in primitive C; overlapping memory copies fail in most languages.
3. A simple implementation of `operator/=` for the `Rational` class if temporaries are not used will fail. This is a nice convincing example because `=` does work, and gives a false sense of security. Explain that the test should always be performed because it is too hard to try to decide whether or not it might be needed.

2.1.7 Operator Overloading

For assignment operators, explain the signature and the return type. The return type is a constant reference because it is just a synonym for the object being assigned to. Explain that the return value represents the value of the expression, `a op= b`, when used in a further expression. If the return type was `void`, that would still change `a`, and would mean only that the result of the assignment could not be used in a larger expression. Show an implementation that includes alias testing and a return of `*this`.

For binary operators, we always have constant member functions and constant reference parameters because the objects do not change. The return value is by copy because the result is a brand new object that did not exist. Explain how reference returns would give garbage. Then show how these operators can always be implemented in terms of the corresponding assignment operators. Comparison operators are straightforward.

Unary operators have no parameters. Explain why `operator+` can return a constant reference but `operator-` cannot (because in the first case, we have a synonym for an already existing object). The `++` and `--` operators are used later on, so discuss them briefly. Again, as the functions are described, make the students tell you if they should be constant members and what the return types and

parameters should be.

2.1.8 Type Conversions

Several things: first, a constructor defines an automatic type conversion whether you want it or not. Second, for a member function, the controlling object (and first parameter for operators) must be an exact match. Third, for reference parameters, the match must be exact (for constant reference, conversions are ok). Fourth, type conversion operators can be written, but avoid them because they will introduce ambiguities.

2.1.9 I/O (Friends), and More on Binary Operators

I/O is achieved by overloading `<<` and `>>` for `ostream` and `istream` objects. Since the first parameter is not of the class, it must be a global function. Since the global function accesses private members, it must be a friend. Remark that the same trick is used for mixed types of operators, such as `==`; furthermore, remark that although a friend function

```
bool operator!=( const Rational& lhs, const Rational& rhs );
```

will match `Rational` and `int` interchangeably, it involves the overhead of construction and destruction. As a result, a high quality class would have function for each case. If `Rational` is on the left side, `operator!=` is overloaded as a member function (with one less parameter); otherwise it is a friend function. I do not like using friends; Section 2.4.1 should be emphasized.

2.1.10 `static` Class Members

Not a big deal. Describe it briefly.

2.1.11 `enum`, and Constant Class Members

You may want to explain the `enum` trick at some point. I hardly use it. For constants, I use `static const` defined in implementation files. These are *static global variables*. This restricts its visibility to the implementation file, and avoids possible name clashes. An alternative is a static constant member.

2.1.12 `string` Class

This is not a high-quality replacement, but does illustrate the basics. There isn't really much new here, except for the two versions of `operator[]`. This is worth a short discussion, but not much more than that, since it is a technical detail and is fairly confusing.

2.2 Solutions To Exercises

In Short

2.1 Information hiding makes implementation details, including components of an object, inaccessible. Encapsulation is the grouping of data and the operations that apply to them to form an aggregate while hiding the implementation of the aggregate. Encapsulation and information hiding are achieved in C++ through the use of the class.

2.2 Members in the public section of a class are visible to non-class routines and can be accessed via the `.` member operator. Private members are not visible outside of the class.

2.3 The constructor is called when an object is created, either by declaration, a call to `new`, or as a member of an object which itself is being constructed. The destructor is called when the object exits scope, either because it is a local variable in a terminating function, it is subject to a `delete`, or it is a member of an object whose destructor is called.

2.4 The copy constructor creates and initializes a new object. It is used to implement call by value. The copy assignment operator copies into an already existing object.

2.5 The default constructor is a member-by-member application of a default constructor. The default destructor is a member-by-member application of a destructor.

2.6 A destructor is not needed if no resources are allocated for the object or if allocated resources are deallocated automatically by the default destructor. `operator=` and a copy constructor should be provided whenever a default would be wrong. The most common example involves data members that are pointers. The default would be a shallow copy.

2.7 (This material is not explicitly discussed in the second edition and thus this question should have been removed.) The benefit of an inline function is that it avoids the overhead of a function call. One disadvantage is that it increases code size (which may offset the gain incurred by avoiding a function call). A second disadvantage is that the definition of an inline function must be visible to all calling routines and the calling routines must be recompiled when the definition of the inline function changes.

2.8 Four operators cannot be overloaded: `.`, `.*`, `?:`, and `sizeof`. Precedence cannot be changed, arity cannot be changed, and only existing operators can be overloaded.

2.9 A friend function of class `C` is a function that is not a member of `C` but nonetheless has access to `C`'s private members.

2.10 `<<` and `>>` need to be overloaded as friend functions. The first parameter and return type are `ostream&` for output and `istream&` for input. The second parameter is a constant reference to a `ClassName` object for output and a reference to a `ClassName` object for input.

2.11 (`i` and `k` may be beyond the scope of this edition of the text.) `a`, `b`, and `d`

are initialized using the constructor at line 25. The initial values are 0, 3, and 0, respectively. `c` and `e` are initialized using the constructor at line 27. `f` is a function that returns a `Rational` object. `g` and `h` point at objects created by a call to the constructors at line 27 and 25, respectively. `i` points at an array of 5 objects each constructed by the constructor at line 25 (with default initial value 0). `j` represents a vector of size 10, with each object constructed as was done for `i`. `k` represents an array of 10 zero-sized vectors.

2.12 (`i` may be beyond the scope of this edition of the text.) We need three calls:

```
delete g;
delete h;
delete[] i;
```

2.13 The `sizeof` operator returns the size of the object, including all private members.

2.14 In that situation, no operations can be performed on the class.

In Theory

2.15 If `rhs` is passed by copy, the copy constructor will be called ad-infinitum.

In Practice

2.16 (d) Although `^` can be overloaded, its precedence is low, so `1+2^3` (with `Rational` objects) would evaluate to 27.

2.18 (a) The return type should be a `string` (that is, return by value), unless the `string` class itself is drastically rewritten. (c) The difference is the usual difference between initialization versus assignment. In alternative 2, an empty string is constructed, and then a copy is performed. This could be more expensive that the initialization at construction.

2.19 (a) Yes, because the `char` can be implicitly converted using a `string` constructor. (b) A temporary is created, and `string::operator+=` is used.

Programming Projects

2.23 First, move `totalDays` into the private section. The default constructors and destructor are acceptable. `operator+=` should return a constant reference. The parameters to `operator-` and `operator<` should be references. The implementation is left to the reader.

2.24 The defaults are acceptable.

2.3 Exam Questions

2.1. Which of the following is the most direct example of how *encapsulation* is supported in C++?

a. constructors
b. destructors
c. member functions
d. public and private specifiers
e. the class declaration

2.2. Which of the following is the most direct example of how *information hiding* is supported in C++?

a. constructors
b. destructors
c. member functions
d. `public` and `private` specifiers
e. the class declaration

2.3. What is the difference between a `struct` and a `class`?

a. constructors and destructors are allowed for the `class`
b. member functions are allowed for the `class`
c. members are private by default for the `class`
d. copying is not allowed for the `struct`
e. none of the above

2.4. Which of the following should be placed in a header file?

a. class interface
b. class implementation
c. inline function bodies
d. (a) and (c)
e. (a), (b), and (c)

2.5. What happens if a non-class function attempts to access a private member?

a. compile time error
b. compile time warning, but program compiles
c. the program compiles but the results are undefined
d. the program is certain to crash
e. some of the above, but the result varies from system to system

2.6. When a parameter is passed call by value, what functions are guaranteed to be called?

a. zero-parameter constructor
b. copy constructor
c. destructor
d. `operator=`
e. two or more of the above

2.7. Suppose the `string` class defines `operator==` as a class member with a single `string` parameter. Assume that no other `operator==` are defined, that a constructor for `string` from `const char *` is defined, but `operator const char* ()` is not defined. `s` is of type `string`. Which of the following is illegal?

a. `s==s`
b. `s=="junk"`
c. `"junk"==s`
d. all are illegal
e. all are legal

2.8. If a class contains a pointer to dynamically allocated memory, which of the following defaults is likely to be unacceptable?

a. copy constructor
b. destructor
c. `operator=`
d. all of the above
e. none of the above

2.9. Which of the following does not default to member-by-member application?

a. zero-parameter constructor
b. copy constructor
c. destructor
d. `operator=`
e. all of the above default to member-by-member application.

2.10. When is a destructor not called?

a. A local automatic variable goes out of scope
b. A pointer variable goes out of scope
c. `delete`
d. `delete []`
e. Destructors are called in all instances

2.11. A copy constructor is not called for

a. call by value
b. return by value
c. `string s = t;  // t is also a string`
d. `string s( t );  // t is also a string`
e. `s = t;  // s and t are both string`

2.12. What is the prototype for the copy constructor?

a. `ClassType( );`
b. `ClassType( ClassType );`
c. `ClassType( ClassType & );`
d. `ClassType( const ClassType & );`
e. None of the above

2.13. A constant member function means that:

a. All of the parameters are constant
b. The object being acted upon is a constant
c. The parameters are not passed using call by value
d. The result of the function call cannot be assigned to
e. none of the above

2.14. Which condition occurs when the same object appears as both an input and output parameter?

a. aliasing
b. copy construction
c. operator overloading
d. type conversion
e. none of the above

2.15. The current object is given by

a. `this`
b. `*this`
c. `&this`
d. `**this`
e. none of the above

2.16. Which of the following statements is false?

a. some operators cannot be overloaded
b. the precedence of an operator cannot be changed
c. the arity of an operator cannot be changed
d. only existing operators can be overloaded
e. binary operators can be overloaded only if both operands have the same type.

2.17. If `operator==` is written as member function, how many parameters should it have?

a. none
b. 1, passed by value
c. 1, passed by constant reference
d. 2, both passed by value
e. 2, both passed by constant reference

2.18. Which of the following `string` operators should not return by constant reference?

a. `operator=`
b. `operator+=`
c. unary `operator+`
d. binary `operator+`
e. constant reference is acceptable for all of these

2.19. Which of (a) to (d) is false about the pointer `this`?

a. it is used to test for aliasing
b. it is used for the return in an assignment operator
c. `this` is not modifiable
d. `*this` is not modifiable
e. all of the above are true

2.20. *Initializer lists* are used in

a. assignment operators
b. all member functions
c. constructors
d. destructors
e. friend functions

2.21. Which of (a) to (d) is false about overloading << for output?

a. the output function cannot be a class member
b. the output function must be a friend
c. the output function takes a stream passed by reference
d. the output function takes an object passed by reference
e. more than one of the above statements is false

2.22. Which of (a) to (c) is false about a static class member?

a. a defining declaration must be placed in the implementation file
b. one member is allocated for each declared class object
c. the static class member is guaranteed to be private to the class
d. two of the above are false
e. all of (a), (b), and (c) are false

2.23. Which of the following is not part of the function signature?

a. function name
b. constant member function declaration
c. parameter passing mechanisms
d. return type
e. all of the above are part of the signature

2.24. Which of the following parameter passing mechanisms requires an exact match?

a. call by value
b. call by reference
c. call by constant reference
d. all of the above require exact matches
e. none of the above require exact matches

2.25. In which of the following cases is a class member *M* invisible in a function *F*?

a. *F* is a member function and *M* is private
b. *F* is a friend function and *M* is private
c. *F* is a general function and *M* is public
d. *F* is a general function and *M* is private
e. none of the above

Answers to Exam Questions

1. C

2. D
3. C
4. D
5. A
6. E
7. C
8. D
9. E
10. B
11. E
12. D
13. B
14. A
15. B
16. E
17. C
18. D
19. D
20. C
21. E
22. D
23. D
24. B
25. D