Workbook on C++ Programming Version 1.0

Author
Richard Conn
University of Cincinnati
Department of Electrical and Computer Engineering

February, 1992

Table of Contents

1. Types and Functions 4
    1.1. Modeling the World with Types 4
    1.2. Types = Structs + Functions 4
    1.3. Data Protection....6
    1.4. Classes 6
    1.5. Function Overloading 7
    1.6. Default Function Arguments....7
2. More C++ Basics 8
    2.1. Type-Safe Linkage 8
    2.2. Constructors and Destructors 8
    2.3. Const....10
    2.4. Inline....10
    2.5. Objects May Be Defined Anywhere....11
    2.6. References....11
    2.7. this 13
3. Even More C++ Basics 14
    3.1. Static Class Members 14
    3.2. Dynamic Object Creation 15
    3.3. Container Classes 16
4. Classes and Inheritance 17
    4.1. Designing Header Files....17
    4.2. Inheritance 18
    4.3. More Inheritance 19
    4.4. Base Class Constructors....20
    4.5. Base Class Destructors 21
5. Families of Types and More Features of C++....23
    5.1. Composition....23
    5.2. Creating Families of Types....24
    5.3. Virtual Functions....24
    5.4. Operator Overloading 25
6. Closing 26

Solutions....27
Solution 1.1....27
Solution 1.2....28
Solution 1.3....29
Solution 1.4....30
Solution 1.5....31
Solution 1.6....32
Solution 2.1....33
Solution 2.2....36
Solution 2.3....37
Solution 2.4....38
Solution 2.5....39
Solution 2.6....40
Solution 2.7....41
Solution 3.1....42
Solution 3.2....43
Solution 3.3....45
Solution 4.1....47
Solution 4.2....49
Solution 4.3....50
Solution 4.4....52
Solution 4.5....54
Solution 5.1....56
Solution 5.2....58
Solution 5.3....60
Solution 5.4....62

The purpose of this workbook is to teach you about C++ in an overview fashion. This workbook follows the video course entitled ***The World of C++*** *by Borland.*

1. Types and Functions

1.1. Modeling the World with Types

Object-oriented programming includes the following basic concepts:

✓ We organize our world using *types*. We use the concept of type to organize our world in a meaningful way. Virtually every object in our world is classified as some "kind of" or "type of" object.

✓ A type has *characteristics* and *behaviors*. Characteristics are represented by data (and the state of that data) and behaviors are represented by functions.

✓ *Object-oriented programming* (sometimes abbreviated as *OOP*) allows real-world types to be represented as abstract data types in the computer. Our computer programs become models of the real world, where the real world is realized as a number of interacting objects and so is our program.

✓ *Inheritance* establishes a relationship between types. It shows which characteristics and behaviors are common and which are different. It also lets you reuse an existing type by adding code on top of the existing code, rather than modifying (or even accessing) the original source code.

✓ A type hierarchy establishes a common interface in the base type, and different implementations in the derived types. This is called *polymorphism*. Polymorphism allows you to create clear programs which can be easily and inexpensively extended.

✓ *Dynamic object creation* lets you decide the quantity, type, and lifetime of variables at run-time rather than when you are writing the program.

Problem 1.1: Draw a hierarchy diagram to represent types of electronic equipment. Put electronic equipment at the root of the hierarchy, with business and entertainment branching from it. Business includes such things as cash registers, copiers, and so on; entertainment includes such things as television, video games, and music systems. How do you classify things that could belong to both groups, like telephones and computers?

1.2. Types = Structs + Functions

C++ allows you to create new types by using structs and functions. Some notes:

✓ C++ has a comment syntax of its own, in addition to the regular C-style comment syntax. The new comment specifier, **//**, starts a comment that continues to the end of the line.

✓ A *type* is a struct with functions. In C++, the concept of a struct is extended so it can contain functions as well as data. An example from the video:

```
struct elevator {
  int floor_selected;
  int floor_number;
  void initialize();  // function declaration in the scope
                      // of the struct
  void select_floor (int floor);
  void go();
};
```

✓ When member functions are defined outside the body of the struct, their associated struct is specified using the scope resolution operator. This is a double colon (::) placed between the type name and the member function name, like this:

```
void typename::member_name() { /* function body */ };
```

In the elevator structure, the initialize function would look like:

```
void elevator::initialize() {
  floor_selected = 0;
  floor_number = 0;
}
```

You use the scope resolution operator any time when the compiler would not normally choose the desired name. For example, if you have a global variable **X** that is hidden by a local variable **X**, you can access the global variable using **::X**. For example,

```
int my_var;       // global variable
void main (void) {
  int my_var;     // local variable
  my_var = 1;     // this sets the local variable above
  ::my_var = 2;   // this sets the global variable
}
```

✓ A member function can access any other member in the same struct, including both data and function members. In the **initialize()** function, it accessed the **floor_selected** and **floor_number** variables.

✓ The C++ compiler automatically generates the equivalent of a **typedef** for a struct name. The name becomes the equivalent of a new keyword. For example, in the elevator example:

```
struct elevator my_elevator;  // create an elevator object
elevator his_elevator;  // "struct" not needed
```

✓ Given the creation of an object via a struct, you invoke a member function by following the object name with a dot and the name of the function. For example, to invoke the initialize() function on the two elevator objects above:

```
my_elevator.initialize();
his_elevator.initialize();
```

✓ You can use existing C code in your programs. All the ANSI C and Borland C libraries are automatically available.

Problem 1.2: Write a C++ program which creates a complex object struct. The struct should contain the member data real_part and imag_part as type double. The struct should also contain the member functions set(), add_one_to(), and print() which sets the value of a complex number, adds 1 to a complex number, and prints a complex number. Define the required member functions by placing their code in the body of the struct (you don't have to use the scope resolution operator). In your mainline, create a complex number, set it to 20i - 30j, print it, add one to it, and print it again.

1.3. Data Protection

Protection of data from accidental modification enforces the proper use of a data type and easily isolates errors. Access is controlled with the three access specifiers:

- ✓ **private:**, which prevents access from the outside world -- only member functions (and friend functions) may change private data or access private member functions
- ✓ **protected:**, which is like private except inherited classes also have access (inheritance is covered later)
- ✓ **public:**, which permits everyone to access the members

For example,

```
struct protection_example {
  int public_can_change_this;  // public by default
private:
  float public_can_not_change_this;
protected:
  float public_can_not_change_this_also;  // unless you
                                          // inherit
};
```

Special access to **private** and **protected** members could be granted to non-member functions -- either ordinary functions, member functions of **structs**, or entire **structs** -- by using the **friend** keyword when declaring the function inside a struct.

Problem 1.3: Write a C++ program which creates a struct named P containing an integer. Define two member functions: set() which sets the value of this integer and print() which prints its value. Also define a friend function called printit(). In your mainline, create an object of type P and set its value to 12. Print it using the member function. Change its value to 14. Print it using the friend function. To get you started, the declaration of struct P should look like this:

```
struct P {
private:
  int value;
public:
  void set (int);
  void print (void);
  friend void printit (P);
};
```

1.4. Classes

The C++ keyword *class* is like *struct* in functionality except that *class* defaults to *private* members while *struct* defaults to *public* members. *class* is the preferred keyword for defining new types, particularly since it adds extended functionality over *struct* (which we will discuss later).

The video showed that *class*, *struct*, *union*, and *enum* are treated similarly by the compiler in that their tag names create reserved words within their scope (similar to doing a *typedef* in C), and the forms of the declarations and definitions are very similar.

Problem 1.4: Write a C++ program which creates a class called counter. Keep the integer which keeps the count for the counter private. Provide public member functions which set, increment, and display the count. Create two counter objects, set them to different values, display them, increment them, and display them again.

1.5. Function Overloading

Function overloading allows you to create more than one function with the same name as long as all the functions have distinct argument lists. Function overloading has many advantages, such as preventing name clashes when you are using multiple libraries.

Some examples of overloaded functions are:

```
void f(int);
void f(int, char);
float f(double);
```

Overloading is resolved by the argument lists only. *The following pair of declarations is invalid and not an example of proper overloading since only the types returned differ:*

```
void f(int);
double f(int);
```

Problem 1.5: Write a C++ program which contains two functions called print(). One function prints an int and the other a double. Output an int and a double using these functions from the mainline.

1.6. Default Function Arguments

Default arguments are used with a function when you want some of the arguments to be automatically inserted by the compiler instead of writing them all out yourself every time you call the function. Here is an example of a function with default arguments and several calls to it:

```
void g(float f, float f2 = 1.1, char x = 'i');

g(12.2);  // f=12.2, f2=1.1, x='i'
g(20.0, 4.0);  // f=20.0, f2=4.0, x='i'
g(100.0, 200.0, 'a');  // f=100.0, f2=200.0, x='a'
```

You can declare a function more than once, but you may only give default arguments once. Only trailing arguments may be given default values, and once you start giving default values, all the rest of the arguments in the list must have defaults.

Problem 1.6: Write a C++ program which contains a function called print() with a default argument of 1. Call print with no arguments and with an argument of 20. As a function, print() is to display the value of its argument to the console.

2. More C++ Basics

2.1. Type-Safe Linkage

In C, you can call functions without declaring them, which means that the C compiler may make incorrect assumptions about that function, such as the type of its return value or the number and types of its parameters. C++ includes several innovations to help reduce errors:

✓ **C++ requires full function prototyping**. C++ forces you to declare all functions and to use full prototypes in those declarations. Even if you declare functions in C, you may declare them incorrectly without generating any error messages. C++ checks every function call during compilation to determine if the number and types of the arguments to the function and the type returned from the function match its prototype.

✓ **C++ has type-safe linkage**. In C++, type-safe linkage occurs because all function names are mangled in the object files. This mangling of function names embeds information about the arguments into the function names.

✓ **C++ supports an alternate linkage specification to provide compatibility with C libraries**. In some situations, such as linking to libraries created with an ANSI C compiler, you may not want C++ to mangle function names. C++ lets you tell it not to mangle a function name through the use of an alternate linkage specification, which looks like this:

```
extern "C" { float round(float); }
```

Problem 2.1: Create two C++ files, one with a function definition (code) and one that declares (contains a function prototype) and uses the function. In the second file (the one with the prototype), make the declaration incorrect by putting the wrong argument type in the prototype. Compile and link the files, noting that the linker catches the error since the files are compiled separately. Create a third file which declares and uses the function, but this time make the declaration incorrect by putting the wrong return type in the prototype. Compile and link the files. Note that the error is not caught -- why do you think the return types are not encoded in function names?

2.2. Constructors and Destructors

When you define a class in C++, a special kind of member function which is automatically called whenever an instance of the class (e.g., an object) is created. This member function is called a *constructor*, and it is designated by having the same name as the class itself. Here is an example of a class with a constructor:

```
class complex {
  float real_part;
  float imag_part;
public:
  complex();  // object is initialized to zero
  void set (float rp, float ip);
  void print(void);
};
```

As mentioned above, constructor calls occur automatically at the point the variable is defined. The user cannot access the variable before the constructor has been called. Although constructors are optional, you will often want to use one. The following shows an example of the declaration of a **complex** object:

```
complex value;  // space is allocated and value is set to
                // 0,0
```

You can create as many overloaded constructors as you want to perform as many different kinds of initialization as you want. For example, extending the definition of the **complex** class above:

```
class complex {
  float real_part;
  float imag_part;
public:
  complex();  // object is initialized to zero
  complex(float rp);  // init only real part, imag is 0
  complex(float rp, float ip);  // init both parts
  void set (float rp, float ip);
  void print(void);
};
```

With this class definition, there are three ways to create **complex** objects:

```
complex val1;            // val1 = 0.0i + 0.0j
complex val2(2.0);       // val2 = 2.0i + 0.0j
complex val3(4.4, 5.5);  // val3 = 4.4i + 5.5j
```

You can also use default arguments with constructors, so long as you don't generate ambiguities. The requirement to not generate ambiguities is true for all functions that use overloading and default arguments. Changing the **complex** class again:

```
class complex {
  float real_part;
  float imag_part;
public:
  complex(float rp = 0.0, float ip = 0.0);  // init 3 ways
  void set (float rp, float ip);
  void print(void);
};
```

This single constructor function supports all three types of object declaration (shown with the declarations of val1, val2, and val3 above).

C++ allows you to ensure proper cleanup with *destructor* functions. A *destructor* is a member function with the same name as the class preceded by a tilde (such as **~complex**). Here is our **complex** class with a destructor:

```
class complex {
  float real_part;
  float imag_part;
public:
  complex(float rp = 0.0, float ip = 0.0);  // init 3 ways
  ~complex();  // destructor
  void set (float rp, float ip);
  void print(void);
};
```

Destructor calls are also invoked automatically, and they occur when a variable goes out of scope. Destructors are optional, but you often need one. You can only create one destructor function for each class, and it cannot have any arguments.

Problem 2.2: Write a C++ program which contains a class that has only a constructor and a destructor as its member functions. Determine the order of constructor and destructor calls for variables by putting printf() statements inside the constructor and destructor and creating several variables inside main(). To generate a unique identifier for each variable, use the keyword **this** inside the printf() statement. **this** is the address of the current variable, and **this** is a pointer. You can print pointers inside printf() by using %p as a format specifier.

2.3. Const

You can use **const** in front of any variable definition to indicate that the value cannot be changed and that the C++ compiler should try not to allocate storage for it, keeping the information about it in the symbol table only. Examples of constants in C++ are:

```
const a = 1;  // int is assumed
const float pi = 3.14159;
const char exit_command = 'x';
```

If you are familiar with ANSI C, the behavior of **const** in C++ is distinctly different from the behavior of **const** in ANSI C. In ANSI C, **const** defaults to *external* linkage (it is global), and **const** always allocates storage for the value, so you cannot use it in constant expressions like array definitions. In C++, **const** defaults to *internal* linkage (as if you had said **static const**). Also, the C++ compiler stores the value of **const**s in the symbol table, so they can be used in constant expressions. Most C++ compilers, however, must allocate storage for user-defined types, so you should only expect to be able to use built-in types in constant expressions.

Because of these differences, you cannot use **const** in a header file or to otherwise replace the use of **#define** in ANSI C, while in C++, use in header files and replacing the use of **#define** is exactly what **const** is for.

Problem 2.3: Write a C++ program which creates an array of integers, where the size of the array is dictated by a **const** variable. Print out the size of the array (using the **sizeof** operator) and the value of the **const** variable.

2.4. Inline

In C++, the preprocessor is seen as a trouble spot not only because it can create expressions with unusual behavior and side effects, but also because it has no concept of type. Type is a fundamental idea in C++, and type checking is a very important way to discover programmer errors during compile time. The preprocessor's ignorance about types means that it can hide errors, making them difficult to find.

C++ provides an improvement to preprocessor macros with the **inline** keyword. **Inline** functions behave exactly like conventional functions, but they do not generate code until the point at which they are called, at which time the code is placed in with the code which calls the function rather than placing a subroutine call in the calling code. *Functions that are defined within a class declaration are automatically* ***inline***, but global functions must use the **inline** keyword.

The function prototype and function body of an **inline** function are stored in the C++ compiler's *symbol table*. When you call an **inline** function, the C++ compiler checks to see that the arguments and return values are correct, and then it substitutes the function body directly into the code.

Problem 2.4: Write a C++ program which contains an inline function that takes a single integer argument, adds 5 to it, and prints the result using printf(). Have the mainline call this inline function 10 times with different arguments. What do you think the object code looks like at each one of these calls?

2.5. Objects May Be Defined Anywhere

In a C++ program, unlike a C program, *objects may be defined anywhere*. There are cases where some variables cannot be initialized until some code has executed, so C++ allows you to define variables at any point in a scope. *These variables exist after their definition to the end of the scope*. For example:

```
complex a;  // create complex number a
a.print();  // print it
complex b;  // create complex number b
b.print();  // print it
```

C++ supports a very sophisticated mechanism for the initialization of aggregates. This means that you can ensure that aggregates are initialized at their point of definition, allowing you to avoid the tedious and error-prone code that may otherwise be required to initialize the aggregate manually.

Variables of built-in types are also regions of storage, so we sometimes call them *objects*. Conversely, we sometimes call *class objects* by the word *variables*. Even function definitions require storage, but these are rarely called objects except where the linker is concerned. In a *pure* object-oriented language like Smalltalk, *everything* is an object, so this distinction is not required. C++ is called a *hybrid* language since it is a hybrid of C and other languages, most notably Simula-67.

Some notes about C++ and its use of storage for objects:

✓ **Storage is reserved at the beginning of a scope**. Like C, the C++ compiler allocates storage on the stack when a scope is entered. To do this, the C++ compiler must scan forward and determine all the variables which are defined in that scope.

✓ **Initialization of objects occurs at the point of definition**. Although the space is reserved upon entering the scope, initialization does not occur until the point at which the object is defined.

✓ **An object is unavailable until it is defined**. The object is not available until after the point of definition. In a class with a constructor and destructor, if you leave a scope before the constructor is called, then the destructor is not called. The compiler will not allow a goto that skips object initialization.

Problem 2.5: To prove that C does not let you create variables anywhere in a scope and that C++ does, create a small C program that has variable definitions after a puts() statement. Compile it first with C++ and then with C.

2.6. References

C++ has the traditional pointer facility of C, and pointers act just the way you would expect, even when calling member functions. C++ also has a new feature called a *reference*, which is like a pointer except the compiler automatically takes the address and dereferences it for you. A *reference* looks just like an object except at the point of creation. *References* are almost exclusively used as function arguments and return values. For example:

```
// Exchange function in C++
void exchange (int &left, int &right)
{
  int temp;
  temp = left;
  left = right;
  right = temp;
}

// Code which calls this exchange () function
int a = 5, b = 4;
exchange (a, b);  // pointers to a and b are passed
```

The above example is equivalent to its C counterpart:

```
/* Exchange function in C */
void exchange (int *left, int *right)
{
  int temp;
  temp = *left;
  *left = *right;
  *right = temp;
}

/* Code which calls this exchange () function */
int a = 5, b = 4;
exchange (&a, &b);  /* pointers must be created by user */
```

Problem 2.6: Write a C++ program which creates two objects which are of the type of this book struct:

```
struct book {
  char title[40];
  char author[20];
};
```

Initialize these objects in an aggregate using dummy data of your choice. Include in the program a function which receives a **book** object by reference and prints out the values of the fields. Call this function twice, one time with each of the books you created.

2.7. this

The keyword **this** is really important in C++. It provides an object which is created as an instance of a class with a mechanism to determine its own address. **this** is a hidden pointer which is created with each instance of a class, generating a little additional overhead in the process.

Problem 2.7: Write a C++ program which contains a class based on the following class declaration:

```
class person {
  char *name;
public:
  person (char *my_name);  // create a person with a given
                           // name
  void print_me(void);  // print the name of a person and
                        // his address using this
};
```

Implement this class. Create five instances of this class with different names. Have each object invoke its **print_me()** member function.

3. Even More C++ Basics

3.1. Static Class Members

The *class members*, both functions and data, discussed so far work with each instance of the class. Each time a new object is created, a new copy of the member data is produced.

A special kind of *class member*, declared with the keyword **static**, is a *class member* which works with the class as a whole, not with the individual members of a class. A *static member data element*, for instance, is created only once for the entire class, regardless of how many instances of that class are created. Static class members can be accessed by all members of a class, and the name of a static member data element is hidden within the scope of the class. Here is an example of a class with a static member data and a static member function:

```
class demo_static {
  static int i;  // only one of these is created
  int j;         // one is created local to each object
public:
  static void f();
};

demo_static x, y, z;  // one int i is created and 3 int j's
```

Defining and initializing *static member data* is performed by a global definition that reserves storage and initializes the data. The **int i** *static member data element* above is initialized as follows:

```
int demo_static::i = 0;
```

The *member function* **f()** can access the **int i** like any other *member data element*.

Static member functions also work with the entire class rather than a particular object. The address of the object, referred to with the keyword **this**, is not secretly passed into a *static member function*. Thus, a *static member function can only access static data members or call other static member functions* unless it gains access to the public members of an object by having the object passed as an argument to it.

A *static member function* is called either with an object or by specifying the class name and the scope resolution operator. Two ways to call the *static member function* **f()** in the example above are:

```
object.f();        // access f() through an object
demo_static::f();  // access f() without an object
```

Problem 3.1: Write a C++ program which defines the following class:

```
class counter {
  static int object_count;
public:
  counter();
  static int get_count(void);
};
```

Each time an object is created, the constructor is to increment the **object_count**. **get_count()** returns the current value of the **object_count**. Your program is to create 5 instances of this class, printing out the current value of **object_count** each time a new object is created.

3.2. Dynamic Object Creation

All the examples shown so far have used static or automatic objects with their memory allocated by the C++ compiler at compile-time. To write a program which uses only static or automatic objects, you must know the quantity, type, and lifetime of all the objects you will ever need in advance. This is a severe limitation when solving more general types of problems when the number and details of the objects in the system are not known until run-time.

Dynamic object creation in C++ lets you choose, at run-time, the type and the lifetime of an object. You can also decide at run-time how many objects you will need. Arrays of objects can even be created and destroyed as needed.

In conventional C, the C standard library functions **malloc()** and **free()** were always used to dynamically create objects and then later destroy them. These C standard library functions are also available in C++ if you wish to use them, but C++ has two new operators which eliminate the need for **malloc()** and **free()** while adding automatic invocation of *constructors* and *destructors*. The C++ operator **new** allocates memory and calls the *constructor* associated with the object to guarantee that the object is properly initialized. The C++ operator **delete** calls the *destructor* associated with the object and then releases the memory associated with the object. These operators let you create objects at run-time as easily and safely as you do at compile-time.

This example uses **new** and **delete** to create and destroy an instance of the class **string**:

```
string *s = new string ("hello");
delete s;
```

Problem 3.2: Write a C++ program which defines a **string** class declared as follows:

```
const max_string_length = 100;
class string {
  char data[max_string_length];
  static int number_of_strings;
public:
  string(char *);  // place arg string into buffer and
                   // increment number_of_strings
  static int count(void);  // return number_of_strings
  void print(void);        // print current string
};
```

Create 3 **string** objects, initializing them to different values. Print out the count of the strings. Create 2 more **string** objects, also initializing them to different values. Print out the count of the strings. Have each **string** object print out its value.

3.3. Container Classes

Container classes, which are sometimes called collections, are classes whose member data elements include instances of other classes. For example, a car class may contain member data elements which are wheels, where a wheel is defined as a class in its own right.

Problem 3.3: Write a C++ program which defines two classes declared as follows:

```
class note {
  char text[40];
public:
  note (char * = " ");  // create a note
  void print (void);  // print the note
};

class note_book {
  note *narray[10];
  int number_of_notes;
public:
  note_book();  // init number_of_notes
  void add (note *);  // add a note to the note_book
  void print(void);  // print out the note_book
};
```

Have your program create 7 **note** objects (containing different values) and add them to the **note_book**. Print out the **note_book**.

4. Classes and Inheritance

4.1. Designing Header Files

In C++, it is often important to organize your code effectively for large projects, libraries, and situations where you are using classes and separate compilation. Correctly organized code greatly facilitates reuse. The organization of the code is most apparent in its *header files*.

A *header file* includes the class *declarations* (but *not* the definitions unless you absolutely have to), function prototypes, **const** values, and anything else that is a part of the public interface to a class or library. Again, header files contain only *declarations*, not *definitions*. If an entity is realized as code or data which occupies space in the memory of the running process, it does not belong in the header file. An example of a C++ *header file* follows:

```
// Header file for a stack class

// This preface ensures that the header file will never
// be included twice
#ifndef STACK_H_
#define STACK_H_

const stack_size = 100;

class stack {
  int stack[stack_size];
  int top_of_stack;  // index of the next available element
                     // in the stack array
public:
  stack();           // init top_of_stack to zero
  void push (int);   // place an element on the stack
  int pop (void);    // extract an element from the stack
  int is_full (void);   // return 1 if stack is full,
                        //0 otherwise
  int is_empty (void);  // return 1 if stack is empty,
                        //0 otherwise
};

#endif // STACK_H_
```

The declarations for classes and functions that belong together should be placed in a single location: the *header file*. Use this header where ever the classes and functions are defined or used to ensure consistency and reduce bugs.

Since C++ does not allow you to re-declare classes, you must insulate header files so the C++ compiler sees their contents only once when compiling a file. The situation often comes up where one header file needs another, so it **#include**s it. If you already have a **#include** directive for this second header file, C++ would see the same header file twice, confusing matters greatly. The example above shows how a header file may be insulated to eliminate this problem. The general rule is to include the following sequence of preprocessor statements around the body of the header file:

```
#ifndef FILE_H_
#define FILE_H_
// code for the body
#endif  // FILE_H_
```

Any defined symbol may be used, but it is common to use a form like FILE_H_ so that it is clear that a header file is involved. In effect, this practice says "if this header file has been included before

(FILE_H_ has already been defined), then ignore the rest of this file." Key to making this work is coming up with an identifier whose name is unique that can be used by the preprocessor so it can determine if the header file was included before. It is easiest to use a modification of the header file name (the header file was named in this case, so the preprocessor symbol FILE_H_ was chosen).

Problem 4.1: Write a header file for a C++ class of complex numbers. Include as many pertinent operations you can think of. Look at the solution in the back, and you may be surprised because the idea of operator overloading is introduced here. We will discuss operator overloading later in this workbook.

4.2. Inheritance

Inheritance in C++ lets you easily create new classes from existing ones. This feature of the language has several benefits:

✓ New classes can be created quickly without introducing bugs to code which was previously debugged.

✓ Code can be reused and augmented without having to rewrite it.

✓ When you extend or modify a system, you don't end up with a lot of copies of similar code to maintain.

✓ You don't need access to the source code of the member function definitions, so you can use another library even if it's just a header file and compiled code. This goes a long way to creating a software business area of designing and selling reusable components libraries in C++.

✓ The design of programs is easier because you can further partition a program into logical pieces.

✓ This technique helps to isolate bugs.

The syntax for inheritance is simple. In the class declaration, place a colon after the class name. Put the name of the class you are inheriting after the colon which follows the name of the class you are declaring. The name of the class you are inheriting is now followed by the opening brace of the body of the class declaration. Inheriting a class called **base** looks like this:

```
class derived : base {
  // declaration of class derived goes here
};
```

Problem 4.2: Write a C++ program which contains the following class definition for a **number** class and the definition of a derived class called **pnumber** which inherits the **number** class definition:

```
class number {
protected:
  int value;
public:
  number(int new_value = 0);
  void set (int new_value = 0);  // change existing number
};
```

Implement the **number** class definition by filling in the member functions. Create the derived **pnumber** class declaration, adding a new member function called **print()** which displays the value of the **pnumber** object. Note that this problem introduces the concept of a **protected** class member, where a **protected** class member is **private** from the point of view of the outside world but **public** from the point of view of

a derived class. Complete your program by creating three **pnumber** objects, setting them to different values upon creation. Print these values, then change them and print them again.

4.3. More Inheritance

The following header file contains the declaration for a class whose mission is to remember its creation date and time:

```
// TSTAMP.H: type of signal which remembers its creation
// time
#ifndef TSTAMP_H_
#define TSTAMP_H_

// Provide access to time(), ctime(), and printf()
#include <time.h>
#include <stdio.h>

class time_stamp {
  time_t stamp
public:
  time_stamp();          // set time stamp
  void showtime(void);  // display time stamp
};

#endif  // TSTAMP.H
```

This header file may be implemented with the following code definition:

```
// TSTAMP.CPP: implementation of TSTAMP
#include "tstamp.h"

time_stamp::time_stamp() {
  time (&stamp);  // get time from system
}

void time_stamp::showtime(void) {
  printf("Time Stamp: %s\n", ctime (&stamp));
}
```

Note that for any class derived from **time_stamp**, if you want to add functions or data to **time_stamp** or modify the existing member functions, these changes are immediately propagated through to all the derived classes. This means that maintaining code becomes much easier because it isn't duplicated when you make changes -- there's a single definition for a function.

Just like members of a class may be **public**, **private**, or (a hybrid) **protected**, base classes may be **public** or **private** (they default to **public**). A ***private*** *base class* is one in which all its members are hidden within the derived class -- its members are **private** to the derived class. A ***public*** *base class* is one in which all of its **public** members are also **public** to the derived class -- its **public** members are available to users of the derived class. For example,

```
class message : private time_stamp {
  char *msg;
public:
  message(char *);  // init the msg ptr
  void print(void); // print the message with date and time
};
```

In the case of class **message**, the only member functions available to the outside world are the constructor and **print()**. However, the situation is different for this case:

```
class message2 : public time_stamp {  // keyword public
                                      // may be omitted
  char *msg;
public:
  message(char *);  // init the msg ptr
  void print(void); // print the message with date and time
};
```

In the case of class **message2**, the member functions available to the outside world are the constructor, **print()**, and **showtime()**, where **showtime()** is in the base class **time_stamp**.

You have just observed one disadvantage to inheritance in C++:

> ***In order to determine all of the member functions available to a derived class, the user must examine each of the base classes. If the base classes are themselves derived, the user must also examine each of the sub-base classes associated with the base classes.***

Breaking your problem into classes has the effect of *partitioning* the problem. This establishes principal dividing lines that are enforced by the C++ compiler, thereby establishing an organization that prevents the kind of entropy that causes spaghetti code. Inheritance partitions your solution even further, so you can try out new ideas without damaging code that works. If a bug appears, it is immediately isolated to the additional code you added during the inheritance process.

Problem 4.3: Write a C++ program which implements **time_stamp**, **message**, and **message2**. You may place all the code in one file to simplify the problem. Create an instance of **message** and an instance of **message2**. Exercise *all* the member functions of each object.

4.4. Base Class Constructors

Inheritance as described so far is a wonderful idea, but one thing is missing -- the invocation of the *constructors* of the base classes, particularly when these constructors require arguments. The constructors of the base classes are *explicitly* called (this is the only time you may explicitly call a constructor in C++) in the *constructor initializer list* for the derived class. The *constructor initializer list* is placed after the constructor argument list of the derived class's constructor definition and before the opening brace of the constructor body. For example:

```
class base {
public:
  base(char *);  // requires a string
};

class derived : base {
public:
  derived (char *);  // requires a string to be passed to
                     // base
};
```

The class **derived** is derived from class **base**. The constructor for class **derived** would look like this:

```
derived::derived(char *data) : base (data) {
  // details of derived constructor
}
```

So, the *constructor initializer list* is used to explicitly pass arguments to the base class constructors. If the base class constructors do not require arguments, they need not be specified in the *constructor initializer list*.

When using inheritance, all the constructors in all the base classes are called, either explicitly using the constructor initializer list or implicitly by the C++ compiler (using the default constructors). ***Constructors are called starting at the base class and working their way up to the derived class.***

The way C++ calls base class constructors ensures that all derived class constructors can depend on the base class being properly initialized.

Problem 4.4: Create a new C++ program from the one you just did on the **time_stamp** base class. Create another derived class called **priority_message**, which requires a second string that indicates the urgency of the message (where urgency strings may be something like "routine", "flash", and "dire emergency"). The **priority_message** class should inherit the **message2** class, replacing **print()** to include the **urgency** string. Add to the constructors of all these classes a **printf()** call that outputs a note saying that the constructor was called. Also add destructors to all these classes which contain **printf()** calls that output notes saying that the destructors are called. Create two **priority_message** objects and note the order of the constructor and destructor calls. Output the **priority_message** values through **print()** member functions declared and defined in the **priority_message** class.

4.5. Base Class Destructors

Only one destructor may be defined for a class, and destructors (which cannot take any arguments) are automatically called by the C++ compiler. The automatic calling of destructors means that you don't have to specify which destructor to call. For derived classes, there is no destructor equivalent to the constructor initializer list.

All destructors are called for an instance of a derived class -- not just the destructor declared in the derived class itself. As with constructors, this is done to ensure that all parts of an object are properly cleaned up.

Destructors are called from the top down, which is the opposite order from which the corresponding constructors were called. This way, any activities the destructor performs can be sure that base class function calls operate properly.

Problem 4.5: Write a C++ program which contains a chain of derived classes like the following:

```
class base {
  char *msg;
public:
  base(char *);   // prints message only
  ~base();        // prints an exit message
  void print(void);  // print msg
};

class derived1 : base {
public:
  derived1(char *);   // prints a different message
  ~derived1();        // prints an exit message
};

class derived2 : derived1 {
public:
  derived2(char *);   // prints a 3rd message
  ~derived2();        // prints an exit message
};
```

Write implementations for all the member functions. Also type the following main() function into your code. Execute the program and note what happens.

```
void main(void)
{
  derived2 x("X");  // in scope of main()
  {  // establishing a new scope in scope of main()
    derived2 y("Y");
  }
  derived2 z("Z");  // also in scope of main()
}
```

When were the destructors for **x**, **y**, and **z** called? When were the constructors for **x**, **y**, and **z** called? When was space allocated for the objects **x**, **y**, and **z**?

5. Families of Types and More Features of C++

5.1. Composition

While inheritance is great in C++, you don't always want to use inheritance (which is a form of derivation) when making new classes out of old ones. *Inheritance* is sometimes called an ***is-a*** relationship between classes; for example, a car is a type of vehicle. Another kind of relationship is a ***has-a*** relationship; for example, a car has an engine and four wheels. *Inheritance* does not make sense in describing a ***has-a*** relationship, but *composition*, wherein a class is built that contains instances of other classes, does make sense. An example of composition:

```
class wheel {
  int wheel_diameter;
public:
  wheel(int diameter);  // set wheel diameter
  void print(void);
};

class vehicle {
  int horse_power;
  wheel lfront, rfront, lrear, rrear;
public:
  vehicle (int hp,
           int diameter_of_each_wheel);
  void print(void);
};

wheel::wheel(int diameter) { wheel_diameter = diameter; }

vehicle::vehicle (int hp,
                  int diameter_of_each_wheel) :
  horse_power(hp),  // init like an assignment
  lfront (diameter_of_each_wheel),  // init each object
  rfront (diameter_of_each_wheel),
  lrear (diameter_of_each_wheel),
  rrear (diameter_of_each_wheel)
{
  // nothing else to be done
}
```

Using composition is just like using built-in types: you create instances of a class inside another class. The only trick is that if the objects have constructors which take arguments, those objects must be explicitly initialized in the constructor initializer list. See the example above and its long constructor initializer list.

Built-in types can be initialized in the constructor initializer list, and **const**s must be initialized this way. Again, see the example above. Note that **wheel_count(number_of_wheels)** in the constructor initializer list is equivalent to **wheel_count = number_of_wheels;** in some line of code (assuming **wheel_count** was not a **const**).

Problem 5.1: Write a C++ program by typing in the above body of code. Add **print()** member functions to both the **wheel** and **vehicle** classes to print out the values of their internal data. Create a **vehicle** object and invoke the member **print()** function on it.

5.2. Creating Families of Types

Inheritance is used to create families of types:

✓ The *base class* provides the common interface for the family. The *base class* is the abstraction of the characteristics and behaviors that are common to all types in the family.

✓ The *derived classes* offer implementations which differ from the base class in significant ways. From the base class, you derive new types to express the differences between all the objects in your type family.

✓ A feature of C++ called *late binding* provides the proper manipulation of a common interface. To use these types effectively, you must be able to send a message to an object and let the object figure out what function to call at run-time. Determining a function call at run-time is called *late binding*, *run-time binding*, or *dynamic binding*. Determining a function call at compile-time is called *early-binding*, *compile-time binding*, or *static binding*.

Problem 5.2: Write a C++ program which demonstrates inheritance and late binding by creating a base class called **pet** and the derived classes of **dog** and **cat**. Define a **speak()** member function for each class. The **speak()** function for the base **pet** class outputs "silence". The **speak()** function for a **dog** outputs "woof" and the **speak()** function for a **cat** outputs "meow". The base class keeps internal data on the type of **pet** it represents using an enum and an internal variable of that enum type. A member function of the base **pet** class called **type()** returns the value of the type of the **pet**.

Write a **talk()** function which is not a member of a class. The **talk()** function is to accept the address of a **pet** as an argument (use references), and cause the correct **speak()** function to be called by determining the **type()** of the **pet** and invoking the associated **speak()** function.

Create a **dog** object, a **cat** object, and a **pet** object. Call the **talk()** function on each of them.

5.3. Virtual Functions

C++ implements late binding with the **virtual** keyword, eliminating the need of the somewhat awkward function selection mechanism illustrated in the solution to the last problem. Some details of how late binding works through *virtual functions* are:

✓ A special pointer, called **VPTR** (pronounced *vee pointer*), is secretly added to a class structure when the class contains virtual functions. The **VPTR** is assigned by the constructor to the address of the **VTABLE** (pronounced *vee table*), and the **VTABLE** contains the addresses of all related virtual functions.

✓ A *virtual function call* consists of code that indexes into the **VTABLE** through the **VPTR**. This way, the function call is resolved at run-time, based on the type of the object.

A *virtual function* is declared like any other function except that the **virtual** keyword prefixes the function declaration.

Problem 5.3: Rewrite the C++ program in problem 5.2 to employ *virtual functions* for **speak()**. Modify talk to use these *virtual functions*. Make the **speak()** function for the **pet** class into a ***pure virtual function***, which is defined as follows:

```
virtual void speak() = 0;
```

Note that instances of classes that contain *pure virtual functions* cannot be created in C++, which is fine since it makes no sense to create such an object. In the real world, there is no generic pet object. Instead, there are dog and cat objects.

5.4. Operator Overloading

In C++, you can change the meaning of almost any operator when that operator is used with a variable of a particular type. Notice that the meaning of the operator does not change everywhere -- just when the C++ compiler matches the proper use of the operator with the objects to which it applies. For instance, if you have two variables of the class **point** called **a** and **b**, the following expressions

```
a + b;
```

will only work if the class **point** has an overloaded **operator+()** function. When the C++ compiler sees a **point** followed by a **+** and another **point**, it will call the function **operator+()** to operate on the first **point** with the second **point** as an argument. C++ is context-sensitive in its selection of the appropriate functions.

Operator overloading is convenient, especially for mathematically-oriented classes where a natural syntax is desired as the instances of the class are being used. However, operator overloading can get a bit tricky until you really understand what is going on.

Problem 5.4: Write a C++ program which contains a class declaration and definition for the following class:

```
class complex {
  float real_part;
  float imag_part;
  char *name;
public:
  complex(char *, float rp = 0.0, float ip = 0.0);
  void set (float rp = 0.0, float ip = 0.0;  // change value
  complex &operator = (complex &);  // assign
  complex operator + (complex &);   // add two objects,
                                    // producing a third
  complex operator - (complex &);
  complex operator * (complex &);
  void print(void);  // print a x + yj
};
```

Write the definition for this class. In your mainline, create three **complex** objects (initializing them to three different values in the process). Add two of them, assigning the result to the third. Print them out. Subtract two of them, assigning the result to the third. Print them out.

6. Closing

We have gone through a lot in the workbook, and it is a good idea to review some of the basic ideas, making sure you don't loose sight of the forest for the trees:

- **We organize the world as *types* when we do object-oriented programming**. People view the world in terms of *families of related types*. We have been taught to think this way since birth. Only in traditional procedural programming must we try to fit a problem in the world into the framework of the computer.

- **C++ organizes a program as types**. C++ lets you organize your code into types which reflect the types we use when organizing a problem in the real world. The code you write becomes an image or model of the problem you are trying to solve.

- **A program has a single essential purpose**. The program has a single job it must do, no matter how complex the program seems or how much peripheral support there is for the fundamental purpose. If you can discover the essential job of the program, it will be easy to read, modify, and extend it. This is because a good C++ program will map the types in the real world onto types in the computer.

- ***Pure abstract base classes* allow us to capture real-world abstractions**. Base classes generally represent the primary concept of an object-oriented program. Because base classes represent concepts, which are abstractions and not specific things, it does not make sense to create objects of an abstract base class. To support this idea, C++ allows you to create *pure virtual functions* by assigning the function body to zero in the class declaration (as we discussed previously). Any class which contains a *pure virtual function* is a ***pure abstract class***. No objects of a *pure abstract class* can be created -- you must use classes derived from the *pure abstract class*. Those classes must have definitions for the *pure virtual functions*.

- **Programs in C++ can be readily created to be *extensible***. To extend a C++ program, you must do two things:

  1. Derive a new class from the *abstract base class*. This new class embodies the extensions you wish to make by redefining the virtual functions in the base class.

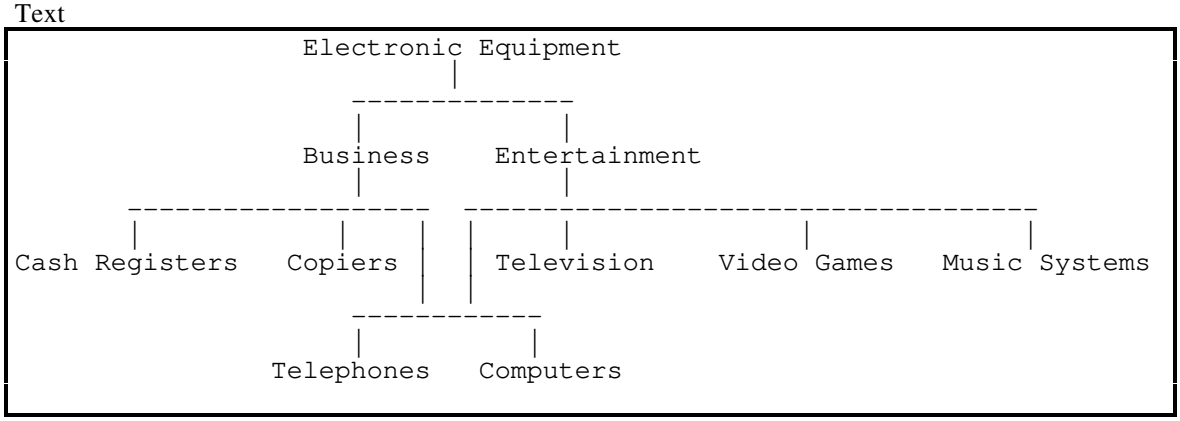
  2. You must add code at the point where you create new objects so the constructor for your new class is called.

Extensible programs are one of the major goals of object-oriented programming because they drastically reduce the cost of creating and maintaining software.

Solutions

Solution 1.1

Text



Solution 1.2

Code

```
#define HEADER "C++ Problem 1.2 by Rick Conn using Borland C++"

#include <stdio.h>

struct complex {
  double real_part;
  double imag_part;
public:
  void set(double rp, double ip)
  {
    real_part = rp;
    imag_part = ip;
  }

  void add_one_to (void)
  {
    real_part += 1.0;
  }

  void print (void)
  {
    printf("(%5.1lfi + %5.1lfj)\n", real_part, imag_part);
  }
};

void main(void)
{
  printf("%s\n", HEADER);

  complex value;
  value.set (20.0, -30.0);
  value.print();
  value.add_one_to();
  value.print();
}
```

Output

```
C++ Problem 1.2 by Rick Conn using Borland C++
( 20.0i + -30.0j)
( 21.0i + -30.0j)
```

Solution 1.3

Code

```
#define HEADER "C++ Problem 1.3 by Rick Conn using Borland C++"

#include <stdio.h>

struct P {
  int value;
public:
  void set (int);
  void print (void);
  friend void printit(P);
};

void P::set (int new_value)
{
  value = new_value;
}

void P::print (void)
{
  printf("%10d\n", value);
}

void printit (P inP)
{
  printf("%10d\n", inP.value);
}

void main(void)
{
  printf("%s\n", HEADER);

  P pobject;
  pobject.set (12);
  pobject.print();
  pobject.set (14);
  printit (pobject);
}
```

Output

```
C++ Problem 1.3 by Rick Conn using Borland C++
        12
        14
```

Solution 1.4

Code

```
#define HEADER "C++ Problem 1.4 by Rick Conn using Borland C++"

#include <stdio.h>

class counter {
  int count;
public:
  void set(int);
  void increment(void);
  void display(void);
};

void counter::set (int new_value) {
  count = new_value;
}

void counter::increment (void) {
  count++;
}

void counter::display (void) {
  printf("The count of the object at address %p is %d\n",
         this, count);
}

void main(void)
{
  printf("%s\n", HEADER);

  counter c1, c2;
  c1.set (5);
  c2.set (-12);
  c1.display();
  c2.display();
  c1.increment();
  c2.increment();
  c1.display();
  c2.display();
}
```

Output

```
C++ Problem 1.4 by Rick Conn using Borland C++
The count of the object at address FFF4 is 5
The count of the object at address FFF2 is -12
The count of the object at address FFF4 is 6
The count of the object at address FFF2 is -11
```

Solution 1.5

Code

```
#define HEADER "C++ Problem 1.5 by Rick Conn using Borland C++"

#include <stdio.h>

void print (int value) {
  printf("The integer value is %10d\n", value);
}

void print (double value) {
  printf("The double  value is %10.2lf\n", value);
}

void main(void)
{
  printf("%s\n", HEADER);

  int i;
  double d;

  i = 4;
  d = 12.2;

  print(i);
  print(d);
}
```

Output

```
C++ Problem 1.5 by Rick Conn using Borland C++
The integer value is          4
The double  value is      12.20
```

Solution 1.6

Code

```
#define HEADER "C++ Problem 1.6 by Rick Conn using Borland C++"

#include <stdio.h>

void print(int value = 1) {
  printf("The value is %2d\n", value);
}

void main(void)
{
  printf("%s\n", HEADER);

  print();
  print(20);
}
```

Output

```
C++ Problem 1.6 by Rick Conn using Borland C++
The value is  1
The value is 20
```

Solution 2.1

Code, Part 1 of 3

```
#define HEADER "C++ Problem 2.1 by Rick Conn using Borland C++"

int fcppv21(int x) { // function definition
  return x+1;  // something simple
}
```

Code, Part 2 of 3

```
#define HEADER "C++ Problem 2.1 by Rick Conn using Borland C++"
#include <stdio.h>

int fcppv21(float);  // prototype with wrong arg type

void main(void)
{
  int i;

  printf("%s\n", HEADER);
  i = fcppv21(2.0);
  printf("I = %d\n", i);
}
```

Code, Part 3 of 3

```
#define HEADER "C++ Problem 2.1 by Rick Conn using Borland C++"
#include <stdio.h>

float fcppv21(int);  // prototype with wrong return type

void main(void)
{
  float i;

  printf("%s\n", HEADER);
  i = fcppv21(2);
  printf("I = %d\n", i);
}
```

Make File, 1 of 2

```
# Makefile for Demonstrating Problem 2.1
# Part A: Attempt to create executable from CPPV2-12.CPP

all:
      # Compile definition of function into object module
      bcc -c cppv2-11.cpp

      # Compile declaration and use of function into object
      bcc -c cppv2-12.cpp

      # Attempt a link (fails due to argument types)
      bcc cppv2-12.obj cppv2-11.obj
```

Make File, 2 of 2

```
# Makefile for Demonstrating Problem 2.1
# Part B: Create executable from CPPV2-13.CPP

all:
      # Compile definition of function into object module
      bcc -c cppv2-11.cpp

      # Compile declaration and use of function into object
      bcc -c cppv2-13.cpp

      # Link
      bcc cppv2-13.obj cppv2-11.obj

      # Run program
      cppv2-13
```

Output, 1 of 2

```
MAKE Version 3.6  Copyright (c) 1991 Borland International

Available memory 1195344 bytes

        bcc -c cppv2-11.cpp
Borland C++  Version 3.00 Copyright (c) 1991 Borland International
cppv2-11.cpp:

        Available memory 824833
        bcc -c cppv2-12.cpp
Borland C++  Version 3.00 Copyright (c) 1991 Borland International
cppv2-12.cpp:

        Available memory 770905
        bcc cppv2-12.obj cppv2-11.obj
Borland C++  Version 3.00 Copyright (c) 1991 Borland International
Turbo Link  Version 5.0 Copyright (c) 1991 Borland International
Error: Undefined symbol fcppv21(float) in module cppv2-12.cpp

        Available memory 822621

** error 1 ** deleting all
```

Output, 2 of 2

```
MAKE Version 3.6  Copyright (c) 1991 Borland International

Available memory 1195344 bytes

        bcc -c cppv2-11.cpp
Borland C++  Version 3.00 Copyright (c) 1991 Borland International
cppv2-11.cpp:

        Available memory 824833
        bcc -c cppv2-13.cpp
Borland C++  Version 3.00 Copyright (c) 1991 Borland International
cppv2-13.cpp:

        Available memory 770905
        bcc cppv2-13.obj cppv2-11.obj
Borland C++  Version 3.00 Copyright (c) 1991 Borland International
Turbo Link  Version 5.0 Copyright (c) 1991 Borland International

        Available memory 822621
        cppv2-13

Floating point error: Domain
Abnormal program termination
** error 3 ** deleting all
```

Solution 2.2

Code

```
#define HEADER "C++ Problem 2.2 by Rick Conn using Borland C++"

#include <stdio.h>

class simple {
public:
  simple(); // constructor
  ~simple();  // destructor
};

simple::simple() {
  printf("Constructor invoked for object at address %p\n", this);
}

simple::~simple() {
  printf("Destructor invoked for object at address %p\n", this);
}

void main(void)
{
  printf("%s\n", HEADER);

  simple a, b, c;
  simple d;
  simple e;
}
```

Output

```
C++ Problem 2.2 by Rick Conn using Borland C++
Constructor invoked for object at address FFF4
Constructor invoked for object at address FFF2
Constructor invoked for object at address FFF0
Constructor invoked for object at address FFEE
Constructor invoked for object at address FFEC
Destructor invoked for object at address FFEC
Destructor invoked for object at address FFEE
Destructor invoked for object at address FFF0
Destructor invoked for object at address FFF2
Destructor invoked for object at address FFF4
```

Solution 2.3

Code

```
#define HEADER "C++ Problem 2.3 by Rick Conn using Borland C++"

#include <stdio.h>

const array_size = 8;

void main(void)
{
  printf("%s\n", HEADER);

  int iarray[array_size];

  printf("The size  of the array is %d\n", sizeof iarray);
  printf("The value of the const is %d\n", array_size);
}
```

Output

```
C++ Problem 2.3 by Rick Conn using Borland C++
The size  of the array is 16
The value of the const is 8
```

Solution 2.4

Code

```
#define HEADER "C++ Problem 2.4 by Rick Conn using Borland C++"

#include <stdio.h>

inline void print_plus_5 (int value)
{
  value += 5;
  printf("The value plus 5 is %d\n", value);
}

void main(void)
{
  printf("%s\n", HEADER);

  print_plus_5(2);
  print_plus_5(-12);
  print_plus_5(10);
  print_plus_5(5);
  print_plus_5(-5);
  print_plus_5(-20);
  print_plus_5(20);
  print_plus_5(200);
  print_plus_5(2000);
  print_plus_5(20000);
}
```

Output

```
C++ Problem 2.4 by Rick Conn using Borland C++
The value plus 5 is 7
The value plus 5 is -7
The value plus 5 is 15
The value plus 5 is 10
The value plus 5 is 0
The value plus 5 is -15
The value plus 5 is 25
The value plus 5 is 205
The value plus 5 is 2005
The value plus 5 is 20005
```

Solution 2.5

Code (C)

```
#define HEADER "C++ Problem 2.5 by Rick Conn using Borland C++"

#include <stdio.h>

void main(void)
{
  printf("%s\n", HEADER);

  puts("This is a test\n");
  int i;
  i = 5;
  printf("I = %d\n", i);
}
```

Code (C++)

```
#define HEADER "C++ Problem 2.5 by Rick Conn using Borland C++"

#include <stdio.h>

void main(void)
{
  printf("%s\n", HEADER);

  puts("This is a test\n");
  int i;
  i = 5;
  printf("I = %d\n", i);
}
```

Output

```
> bcc cppv2-5.c
Borland C++  Version 3.00 Copyright (c) 1991 Borland International
cppv2-5.c:
Error cppv2-5.c 10: Declaration is not allowed here in function main
*** 1 errors in Compile ***

        Available memory 905112
> bcc cppv2-5.cpp
Borland C++  Version 3.00 Copyright (c) 1991 Borland International
cppv2-5.cpp:
Turbo Link  Version 5.0 Copyright (c) 1991 Borland International

        Available memory 894596
> cppv2-5
C++ Problem 2.5 by Rick Conn using Borland C++
This is a test

I = 5
```

Solution 2.6

Code

```
#define HEADER "C++ Problem 2.6 by Rick Conn using Borland C++"

#include <stdio.h>

struct book {
  char title[40];
  char author[20];
};

void print_book (book &name)
{
  printf(" Title: %s\n", name.title);
  printf("Author: %s\n", name.author);
}

void main(void)
{
  printf("%s\n", HEADER);

  book textbook = { "Turbo C++ DiskTutor",
                    "Voss & Chui" };
  book refbook = { "The Annotated C++ Reference Manual",
                   "Ellis & Stroustrup" };
  print_book (textbook);
  print_book (refbook);
}
```

Output

```
C++ Problem 2.6 by Rick Conn using Borland C++
 Title: Turbo C++ DiskTutor
Author: Voss & Chui
 Title: The Annotated C++ Reference Manual
Author: Ellis & Stroustrup
```

Solution 2.7

Code

```
#define HEADER "C++ Problem 2.7 by Rick Conn using Borland C++"

#include <stdio.h>

class person {
  char *name;
public:
  person (char *my_name);  // create a person with
                           // a given name
  void print_me(void);  // print the name of the person
                        // and his address using this
};

person::person (char *my_name)
{
  name = my_name;
}

void person::print_me(void)
{
  printf("The name is %s\n", name);
  printf("The address is %p\n", this);
}

void main(void)
{
  printf("%s\n", HEADER);

  person ck ("Clark Kent");
  person s ("Superman");
  person bw ("Bruce Wayne");
  person b ("Batman");
  person bs ("Bjarne Stroustrop");

  ck.print_me();
  s.print_me();
  bw.print_me();
  b.print_me();
  bs.print_me();
}
```

Output

```
C++ Problem 2.7 by Rick Conn using Borland C++
The name is Clark Kent
The address is FFF4
The name is Superman
The address is FFF2
The name is Bruce Wayne
The address is FFF0
The name is Batman
The address is FFEE
The name is Bjarne Stroustrop
The address is FFEC
```

Solution 3.1

Code

```
#define HEADER "C++ Problem 3.1 by Rick Conn using Borland C++"

#include <stdio.h>

class counter {
  static int object_count;
public:
  counter();
  static int get_count(void);
};

int counter::object_count = 0;  // init count

counter::counter() { counter::object_count++; }

int counter::get_count(void)
{
  return counter::object_count;
}

void main(void)
{
  printf("%s\n", HEADER);

  counter c1;
  printf("The count is %d\n",
    c1.get_count());

  counter c2;
  printf("The count is %d\n",
    counter::get_count());

  counter c3;
  printf("The count is %d\n",
    c3.get_count());

  counter c4;
  printf("The count is %d\n",
    c4.get_count());

  counter c5;
  printf("The count is %d\n",
    c5.get_count());
}
```

Output

```
C++ Problem 3.1 by Rick Conn using Borland C++
The count is 1
The count is 2
The count is 3
The count is 4
The count is 5
```

Solution 3.2

Code

```
#define HEADER "C++ Problem 3.2 by Rick Conn using Borland C++"

#include <stdio.h>
#include <string.h>  // for strcpy()

const max_string_length = 100;

class string {
  char data[max_string_length];
  static int number_of_strings;
public:
  string (char *);
  static int count (void);
  void print (void);
};

int string::number_of_strings = 0;

string::string(char *new_string) {
  strcpy(data, new_string);
  string::number_of_strings++;
}

int string::count(void) {
  return number_of_strings;
}

void string::print(void) {
  printf("String = \"%s\"\n", data);
}

void main(void)
{
  printf("%s\n", HEADER);

  string s1("This is a test");
  string s2("This is only a test");
  string s3("This is fun");
  printf("The count is %d\n", string::count());

  string s4("Another string");
  string s5("Yet another string");
  printf("The count is %d\n", string::count());

  s1.print();
  s2.print();
  s3.print();
  s4.print();
  s5.print();
}
```

Output

```
C++ Problem 3.2 by Rick Conn using Borland C++
The count is 3
The count is 5
String = "This is a test"
String = "This is only a test"
String = "This is fun"
String = "Another string"
String = "Yet another string"
```

Solution 3.3

Code

```
#define HEADER "C++ Problem 3.3 by Rick Conn using Borland C++"

#include <stdio.h>
#include <string.h>  // for strcpy()
#include <mem.h>     // for memcpy()

class note {
  char text[40];
public:
  note (char *cp = "");
  void print (void);
};

class note_book {
  note *narray[10];
  int number_of_notes;
public:
  note_book();
  void add (note *);
  void print(void);
};

note::note (char *value) { strcpy(text, value); }

void note::print(void) { printf("Note: %s\n", text); }

note_book::note_book() { number_of_notes = 0; }

void note_book::add (note *newnote) {
  narray[number_of_notes] = newnote;
  number_of_notes++;
}

void note_book::print(void) {
  int i;

  for (i=0; i<number_of_notes; i++) {
    printf("%2d: ", i);
    narray[i] -> print();
  }
}

void main(void)
{
  printf("%s\n", HEADER);

  note_book nb;

  note n1("This is a test");     note n2("This is only a test");
  note n3("How far will I go?"); note n4("Perhaps just so far");
  note n5("This is fun");        note n6("This is boring");
  note n7("This works");

  nb.add (&n1);  nb.add (&n2);  nb.add (&n3);  nb.add (&n4);
  nb.add (&n5);  nb.add (&n6);  nb.add (&n7);

  nb.print();
}
```

Output

```
C++ Problem 3.3 by Rick Conn using Borland C++
 0: Note: This is a test
 1: Note: This is only a test
 2: Note: How far will I go?
 3: Note: Perhaps just so far
 4: Note: This is fun
 5: Note: This is boring
 6: Note: This works
```

Solution 4.1

Header File

```
// CPPV4-1.H by Rick Conn Using Borland C++
#ifndef COMPLEX_H_
#define COMPLEX_H_

// COMPLEX Class
class complex {
  float real_part;
  float imag_part;
  char *name;
public:
  complex (char *, float rp=0.0, float ip=0.0);
  void set (float rp=0.0, float ip=0.0);
  complex & operator= (complex &);
  complex operator+ (complex &right);
  complex operator- (complex &right);
  complex operator* (complex &right);
  void print(void);
};

complex::complex (char *n, float rp, float ip) {
  name = n;  real_part = rp;  imag_part = ip;
}

void complex::set (float rp, float ip) {
  real_part = rp;  imag_part = ip;
}

complex & complex::operator= (complex &arg) {
  real_part = arg.real_part;
  imag_part = arg.imag_part;
  return *this;
}

complex complex::operator+ (complex &right) {
  complex result ("Temp");
  result.real_part = real_part + right.real_part;
  result.imag_part = imag_part + right.imag_part;
  return result;
}

complex complex::operator- (complex &right) {
  complex result ("Temp");
  result.real_part = real_part - right.real_part;
  result.imag_part = imag_part - right.imag_part;
  return result;
}

complex complex::operator* (complex &right) {
  complex result ("Temp");
  result.real_part = real_part * right.real_part -
                     imag_part * right.imag_part;
  result.imag_part = imag_part * right.real_part +
                     real_part * right.imag_part;
  return result;
}

void complex::print(void) {
  printf("  %s: %10.5f + %10.5fi\n",
          name, real_part, imag_part);
}
```

```
#endif // COMPLEX_H_
```

Code

```
#define HEADER "C++ Problem 4.1 by Rick Conn using Borland C++"

#include <stdio.h>
#include "cppv4-1.h"

void main(void)
{
  printf("%s\n", HEADER);

  complex a("A"), b("B", 2.0, 3.0), c("C");

  a = b;
  printf("A = B\n");
  a.print(); b.print(); c.print();
  a.set(5.0, -4.0);
  printf("A = 5 - 4i\n");
  a.print(); b.print(); c.print();
  c = a + b;
  printf("C = A + B\n");
  a.print(); b.print(); c.print();
  c = a - b;
  printf("C = A - B\n");
  a.print(); b.print(); c.print();
  c = a * b;
  printf("C = A * B\n");
  a.print(); b.print(); c.print();
}
```

Output

```
C++ Problem 4.1 by Rick Conn using Borland C++
A = B
  A:    2.00000 +    3.00000i
  B:    2.00000 +    3.00000i
  C:    0.00000 +    0.00000i
A = 5 - 4i
  A:    5.00000 +   -4.00000i
  B:    2.00000 +    3.00000i
  C:    0.00000 +    0.00000i
C = A + B
  A:    5.00000 +   -4.00000i
  B:    2.00000 +    3.00000i
  C:    7.00000 +   -1.00000i
C = A - B
  A:    5.00000 +   -4.00000i
  B:    2.00000 +    3.00000i
  C:    3.00000 +   -7.00000i
C = A * B
  A:    5.00000 +   -4.00000i
  B:    2.00000 +    3.00000i
  C:   22.00000 +    7.00000i
```

Solution 4.2

Code

```
#define HEADER "C++ Problem 4.2 by Rick Conn using Borland C++"

#include <stdio.h>

class number {
protected:
  int value;
public:
  number (int new_value = 0);
  void set (int new_value = 0);
};

class pnumber : public number {
public:
  // Note: I had to add a constructor for pnumber
  // because number's constructor required an argument
  pnumber (int new_value = 0);
  void print(void);
};

number::number(int new_value) {
  value = new_value;
}

void number::set (int new_value) {
  value = new_value;
}

pnumber::pnumber(int new_value) {
  value = new_value;
}

void pnumber::print (void) {
  printf("The value is %d\n", value);
}

void main(void)
{
  printf("%s\n", HEADER);

  pnumber a (12), b(20), c(0);
  a.print(); b.print(); c.print();

  a.set(1); b.set(2); c.set(3);
  a.print(); b.print(); c.print();
}
```

Output

```
C++ Problem 4.2 by Rick Conn using Borland C++
The value is 12
The value is 20
The value is 0
The value is 1
The value is 2
The value is 3
```

Solution 4.3

Code

```
#define HEADER "C++ Problem 4.3 by Rick Conn using Borland C++"

#include <stdio.h>
#include <time.h>

class time_stamp {
  time_t stamp;
public:
  time_stamp();
  void showtime(void);
};

time_stamp::time_stamp() {
  time (&stamp);
}

void time_stamp::showtime(void) {
  printf("Time Stamp: %s", ctime (&stamp));
}

class message : private time_stamp {
  char *msg;
public:
  message (char *);
  void print(void);
};

message::message (char *m) {
  msg = m;
}

void message::print(void) {
  printf("Message \"%s\"  ", msg);
  showtime();
}

class message2 : public time_stamp {
  char *msg;
public:
  message2 (char *);
  void print(void);
};

message2::message2 (char *m) {
  msg = m;
}

void message2::print(void) {
  printf("Message2 \"%s\"  ", msg);
  showtime();
}

void main(void)
{
  printf("%s\n", HEADER);

  message  m1("This is a test");
  message2 m2("Another test");

  // All member functions of message
```

```
  m1.print();

  // All member functions of message2
  m2.print();
  m2.showtime();
}
```

Output

```
C++ Problem 4.3 by Rick Conn using Borland C++
Message "This is a test"  Time Stamp: Mon Feb 10 05:35:39 1992
Message2 "Another test"  Time Stamp: Mon Feb 10 05:35:39 1992
Time Stamp: Mon Feb 10 05:35:39 1992
```

Solution 4.4

Code

```
#define HEADER "C++ Problem 4.4 by Rick Conn using Borland C++"

#include <stdio.h>
#include <time.h>

class time_stamp {
  time_t stamp;
public:
  time_stamp();
  ~time_stamp();
  void showtime(void);
};

time_stamp::time_stamp() {
  time (&stamp);
  printf("  Time_Stamp constructor called\n");
}

time_stamp::~time_stamp() {
  printf("  Time_Stamp destructor called\n");
}
void time_stamp::showtime(void) {
  printf("Time Stamp: %s", ctime (&stamp));
}

class message : private time_stamp {
  char *msg;
public:
  message (char *);
  ~message();
  void print(void);
};

message::message (char *m) {
  msg = m;
  printf("  Message constructor called\n");
}
message::~message() {
  printf("  Message destructor called\n");
}
void message::print(void) {
  printf("Message \"%s\"  ", msg);
  showtime();
}

class message2 : public time_stamp {
protected:
  char *msg;
public:
  message2 (char *);
  ~message2();
  void print(void);
};

message2::message2 (char *m) {
  msg = m;
  printf("  Message2 constructor called\n");
}
message2::~message2() {
  printf("  Message2 destructor called\n");
```

```
}
void message2::print(void) {
  printf("Message2 \"%s\"  ", msg);
  showtime();
}

class priority_message : public message2 {
  char *urgency;
public:
  priority_message (char *m, char *u);
  ~priority_message();
  void print(void);  // includes urgency info
};

priority_message::priority_message (char *m, char *u) :
  message2(m) {
    urgency = u;
    printf("  Priority_Message constructor called\n");
}
priority_message::~priority_message() {
  printf("  Priority_Message destructor called\n");
}
void priority_message::print(void) {
  printf("Urgency %s: %s  -- ", urgency, msg);
  showtime();
}

void main(void)
{
  printf("%s\n", HEADER);

  priority_message pm1("This is a test", "Routine");
  priority_message pm2("Em Situation", "Emergency");

  pm1.print();
  pm2.print();
}
```

Output

```
C++ Problem 4.4 by Rick Conn using Borland C++
  Time_Stamp constructor called
  Message2 constructor called
  Priority_Message constructor called
  Time_Stamp constructor called
  Message2 constructor called
  Priority_Message constructor called
Urgency Routine: This is a test  -- Time Stamp: Mon Feb 10 05:50:10 1992
Urgency Emergency: Em Situation  -- Time Stamp: Mon Feb 10 05:50:10 1992
  Priority_Message destructor called
  Message2 destructor called
  Time_Stamp destructor called
  Priority_Message destructor called
  Message2 destructor called
  Time_Stamp destructor called
```

Solution 4.5

Code

```
#define HEADER "C++ Problem 4.5 by Rick Conn using Borland C++"

#include <stdio.h>

class base {
protected:
  char *msg;
public:
  base(char *);
  ~base();
  void print(void);
};

class derived1 : base {
public:
  derived1(char *);
  ~derived1();
};

class derived2 : derived1 {
public:
  derived2(char *);
  ~derived2();
};

base::base (char *m) {
  msg = m;
  printf("Base constructor called with message %s\n", msg);
}

base::~base() {
  printf("  Base destructor called with message %s\n", msg);
}

void base::print(void) {
  printf(" with message %s\n", msg);
}

derived1::derived1 (char *m) : base(m) {
  printf("  Derived1 constructor called");
  print();
}

derived1::~derived1() {
  printf("  Derived1 destructor called");
  print();
}

derived2::derived2 (char *m) : derived1(m) {
  printf("  Derived2 constructor called\n");
}

derived2::~derived2() {
  printf("Derived2 destructor called\n");
}

void main(void)
{
  printf("%s\n", HEADER);
```

```
  derived2 x ("X");
  {
    derived2 y ("Y");
  }
  derived2 z ("Z");
}
```

Output

```
C++ Problem 4.5 by Rick Conn using Borland C++
Base constructor called with message X
  Derived1 constructor called with message X
  Derived2 constructor called
Base constructor called with message Y
  Derived1 constructor called with message Y
  Derived2 constructor called
Derived2 destructor called
  Derived1 destructor called with message Y
  Base destructor called with message Y
Base constructor called with message Z
  Derived1 constructor called with message Z
  Derived2 constructor called
Derived2 destructor called
  Derived1 destructor called with message Z
  Base destructor called with message Z
Derived2 destructor called
  Derived1 destructor called with message X
  Base destructor called with message X
```

Solution 5.1

Code

```
#define HEADER "C++ Problem 5.1 by Rick Conn using Borland C++"

#include <stdio.h>

class wheel {
  int wheel_diameter;
public:
  wheel (int diameter);
  void print(void);
};

class vehicle {
  int horse_power;
  wheel lfront, rfront, lrear, rrear;
public:
  vehicle (int hp,
           int diameter_of_each_wheel);
  void print(void);
};

wheel::wheel (int diameter) { wheel_diameter = diameter; }

void wheel::print(void) {
  printf("Wheel diameter = %d\n", wheel_diameter);
}

vehicle::vehicle (int hp,
                  int diameter_of_each_wheel) :
  horse_power(hp),
  lfront(diameter_of_each_wheel),
  rfront(diameter_of_each_wheel),
  lrear(diameter_of_each_wheel),
  rrear(diameter_of_each_wheel)
{
  // nothing else to be done
}

void vehicle::print(void) {
  printf("Horse Power = %d\n", horse_power);
  lfront.print();
  rfront.print();
  lrear.print();
  rrear.print();
}

void main(void)
{
  printf("%s\n", HEADER);

  vehicle v (190, 32);
  v.print();
}
```

Output

```
C++ Problem 5.1 by Rick Conn using Borland C++
Horse Power = 190
Wheel diameter = 32
Wheel diameter = 32
Wheel diameter = 32
Wheel diameter = 32
```

Solution 5.2

Code

```
#define HEADER "C++ Problem 5.2 by Rick Conn using Borland C++"

#include <stdio.h>

enum pet_kind {doggy, kitty, neither};

class pet {
  pet_kind pk;
public:
  pet(pet_kind);
  void speak(void);
  pet_kind type(void);
};

class dog : public pet {
public:
  dog();
  void speak(void);
};

class cat : public pet {
public:
  cat();
  void speak(void);
};

pet::pet(pet_kind kind) {
  pk = kind;
}

void pet::speak(void) {
  printf("silence\n");
}

pet_kind pet::type(void) {
  return pk;
}

dog::dog() : pet(doggy) {
  // nothing to do
}

void dog::speak(void) {
  printf("woof\n");
}

cat::cat() : pet(kitty) {
  // nothing to do
}

void cat::speak(void) {
  printf("meow\n");
}

void talk (pet *p) {
  switch (p->type()) {
    case doggy : ((dog *)p) -> speak();
                 break;
    case kitty : ((cat *)p) -> speak();
                 break;
```

```
    default     : p -> speak();
                  break;
  }
}

void main(void)
{
  printf("%s\n", HEADER);

  dog scotty;
  cat fluffy;
  pet funny (neither);

  talk(&scotty);
  talk(&fluffy);
  talk(&funny);
}
```

Output

```
C++ Problem 5.2 by Rick Conn using Borland C++
woof
meow
silence
```

Solution 5.3

Code

```
#define HEADER "C++ Problem 5.3 by Rick Conn using Borland C++"

#include <stdio.h>

enum pet_kind {doggy, kitty, neither};

class pet {
  pet_kind pk;
public:
  pet(pet_kind);
  virtual void speak(void);
  pet_kind type(void);
};

class dog : public pet {
public:
  dog();
  void speak(void);
};

class cat : public pet {
public:
  cat();
  void speak(void);
};

pet::pet(pet_kind kind) {
  pk = kind;
}

void pet::speak(void) { }

pet_kind pet::type(void) {
  return pk;
}

dog::dog() : pet(doggy) {
  // nothing to do
}

void dog::speak(void) {
  printf("woof\n");
}

cat::cat() : pet(kitty) {
  // nothing to do
}

void cat::speak(void) {
  printf("meow\n");
}

void talk (pet *p) {
  p -> speak();
}

void main(void)
{
  printf("%s\n", HEADER);
```

```
  dog scotty;
  cat fluffy;

  talk(&scotty);
  talk(&fluffy);
}
```

Output

```
C++ Problem 5.3 by Rick Conn using Borland C++
woof
meow
```

Solution 5.4

Code

```
#define HEADER "C++ Problem 5.4 by Rick Conn using Borland C++"

#include <stdio.h>

// COMPLEX Class
class complex {
  float real_part;
  float imag_part;
  char *name;
public:
  complex (char *, float rp=0.0, float ip=0.0);
  void set (float rp=0.0, float ip=0.0);
  complex & operator= (complex &);
  complex operator+ (complex &right);
  complex operator- (complex &right);
  complex operator* (complex &right);
  void print(void);
};

complex::complex (char *n, float rp, float ip) {
  name = n;  real_part = rp;  imag_part = ip;
}

void complex::set (float rp, float ip) {
  real_part = rp;  imag_part = ip;
}

complex & complex::operator= (complex &arg) {
  real_part = arg.real_part;
  imag_part = arg.imag_part;
  return *this;
}

complex complex::operator+ (complex &right) {
  complex result ("Temp");
  result.real_part = real_part + right.real_part;
  result.imag_part = imag_part + right.imag_part;
  return result;
}

complex complex::operator- (complex &right) {
  complex result ("Temp");
  result.real_part = real_part - right.real_part;
  result.imag_part = imag_part - right.imag_part;
  return result;
}

complex complex::operator* (complex &right) {
  complex result ("Temp");
  result.real_part = real_part * right.real_part -
                     imag_part * right.imag_part;
  result.imag_part = imag_part * right.real_part +
                     real_part * right.imag_part;
  return result;
}

void complex::print(void) {
  printf("  %s: %10.5f + %10.5fi\n",
          name, real_part, imag_part);
}
```

```
void main(void)
{
  printf("%s\n", HEADER);

  complex a("A"), b("B", 2.0, 3.0), c("C");

  a = b;
  printf("A = B\n");
  a.print(); b.print(); c.print();
  a.set(5.0, -4.0);
  printf("A = 5 - 4i\n");
  a.print(); b.print(); c.print();
  c = a + b;
  printf("C = A + B\n");
  a.print(); b.print(); c.print();
  c = a - b;
  printf("C = A - B\n");
  a.print(); b.print(); c.print();
  c = a * b;
  printf("C = A * B\n");
  a.print(); b.print(); c.print();
}
```

Output

```
C++ Problem 5.4 by Rick Conn using Borland C++
A = B
  A:    2.00000 +    3.00000i
  B:    2.00000 +    3.00000i
  C:    0.00000 +    0.00000i
A = 5 - 4i
  A:    5.00000 +   -4.00000i
  B:    2.00000 +    3.00000i
  C:    0.00000 +    0.00000i
C = A + B
  A:    5.00000 +   -4.00000i
  B:    2.00000 +    3.00000i
  C:    7.00000 +   -1.00000i
C = A - B
  A:    5.00000 +   -4.00000i
  B:    2.00000 +    3.00000i
  C:    3.00000 +   -7.00000i
C = A * B
  A:    5.00000 +   -4.00000i
  B:    2.00000 +    3.00000i
  C:   22.00000 +    7.00000i
```