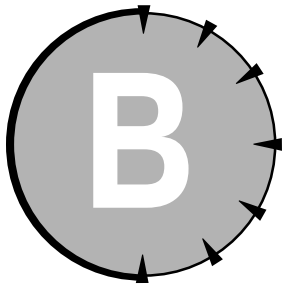
APPENDIX



Supplemental Problems

This appendix contains supplemental problems from different parts of the book to give you additional practice in working with C++ and honing your new skills. The problems are given below in one section divided by part number; following is another section with answers to the problems.

Problems

Saturday Morning

1. Which of the following statements *should*:

a. generate no messages?

b. generate warnings?

c. generate errors?

```
1. int n1, n2, n3;
2. float f1, f2 = 1;
3. double d1, d2;
4. n1 = 1; n2 = 2; n3 = 3; f2 = 1;
5. d1 = n1 * 2.3;
6. n2 = n1 * 2.3;
7. n2 = d1 * 1;
```

8. `n3 = 100; n3 = n3 * 1000;`
9. `f1 = 200 * f2;`

2. Given that `n1` is equal to 10, evaluate the following:

- **a.** `n1 / 3`
- **b.** `n1 % 3`
- **c.** `n1++`
- **d.** `++n1`
- **e.** `n1 %= 3`
- **f.** `n1 -= n1`
- **g.** `n1 = -10; n1 = +n1;` what is `n1`?

3. What is the difference between the following `for` loops?

```
for(int i = 0; i < 10; i++)
{
    // ...
}
for (int i = 0; i < 10; ++i)
{
    // ...
}
```

4. Write the function `int cube(int n)`, which calculates the `n * n * n` of `n`.

5. Describe exactly what happens in the following case using the function written in problem 1?

```
int n = cube(3.0);
```

6. The following program demonstrates a very crude function that calculates the integer square root of a number by repetitively comparing the square to the actual value. In other words, 4 * 4 = 16, therefore the square root of 16 is 4. This function reckons that 3 is the square root of 15 because 3 * 3 is less than 15 but 4 * 4 is greater than 15.

However, the program generates unexpected results. Fix it!

```
// Lesson - this function demonstrates a crude but
//          effective method for calculating a
//          square root using only integers.
//          This version does not work.
#include <stdio.h>
#include <iostream.h>
```

```
// squareRoot - given a number n, return its square
//              root by calculating nRoot * nRoot
//              for ever increasing values of
//              nRoot until n is surpassed.
void squareRoot(int n)
{
    // start counting at 1
    int nRoot = 1;

    // loop forever
    for(;;)
    {
        // check the square of the current
        // value (and increment to the next)
        if ((nRoot++ * nRoot) > n)
        {
            // close as we can get, return
            // what we've got
            return nRoot;
        }
    }

    // shouldn't be able to get here
    return 0;
}

// test the function squareRoot() with a
// single value (a much more thorough test
// would be required except that this already
// doesn't work)
int main(int argc, char* pszArgs[])
{
    cout << "This program doesn't work!\n";

    cout << "Square root of "
         << 16
         << " is "
         << squareRoot(16)
         << "\n";
```

```
    return 0;
}
```

Hint: Be very wary of the autoincrement feature, as well as the postincrement feature.

Saturday Afternoon

1. The C++ library function `strchr()` returns the index of a given character in a string. For example, `strchr("abcdef", 'c')` returns 2. `strchr()` returns a –1 if the character does not exist anywhere within the string.

Write `myStrchr()` which works the same as `strchr()`.

When you think that your function is ready, execute it against the following:

```
// MyStrchr - search a given character within
//            a string. Return the index of
//            of the result.
#include <stdio.h>


#include <iostream.h>

// myStrchr - return the index of a test
//            character in a string. Return
//            a -1 if the character not found.
int myStrchr(char target[], char testChar);

// test the myStrchr function with different
// string combinations
void testFn(char szString[], char cTestChar)
{
    cout << "The offset of "
         << cTestChar
         << " in "
         << szString
         << " is "
         << myStrchr(szString, cTestChar)
         << "\n";
}
```

```
int main(int nArgs, char* pszArgs[])
{
    testFn("abcdefg", 'c');
    testFn("abcdefg", 'a');
    testFn("abcdefg", 'g');
    testFn("abcdefc", 'c');
    testFn("abcdefg", 'x');

    return 0;
}
```

Hint: Make sure that you do not run off the end of the string in the event that you don't find the desired character.

2. Although the following is bad programming, this mistake may not cause problems. Please explain why problems might not be caused.

```
void fn(void)
{
    double d;
    int*   pnVar = (int*)&d;
    *pnVar = 10;
}
```

3. Write a pointer version of the following `displayString()`. Assume that this is a null terminated string (don't pass the length as an argument to the function).

```
void displayCharArray(char sArray[], int nSize)
{
    for(int i = 0; i< nSize; i++)
    {
        cout << sArray[i];
    }
}
```

4. Compile and execute the following program. Explain the results:

```
#include <stdio.h>
#include <iostream.h>

// MyClass - a meaningless test class
class MyClass
{
```

```
    public:
        int n1;
        int n2;
};

int main(int nArg, char* nArgs[])
{
    MyClass* pmc;
    cout << "n1  = " << pmc->n1
         << ";n2 = " << pmc->n2
         << "\n";
    return 0;
}
```

Sunday Morning

1. Write a class `Car` that inherits from the class `Vehicle` and that has a `Motor`. The number of wheels is specified in the constructor to the class `Vehicle` and the number of cylinders specified in the constructor for `Motor`. Both of these values are passed to the constructor for the class `Car`.

Answers

Saturday Morning

1. 1. No problem.
   2. No warning: you can initialize however you like.
   3. Nope.
   4. Nothing here.
   5. No problem: `n1` is automatically promoted to a double in order to perform the multiplication. Most compilers will not note this conversion.
   6. `n1 * 2.3` results in a double. Assigning this back to an `int` results in a demotion warning.

7. Similar to 6. Although 1 is an `int`, `d1` is a `double`. The result is a `double` that must be demoted.

8. No warning but it doesn't work. The result is beyond the range of an `int`.

9. This should generate a warning (it does under Visual C++ but not under GNU C++). The result of multiplying an `int` times a `float` is a `double` (all calculations are performed in double precision). This must be demoted to be assigned to a `float`.

2. **a.** 3

Round off error as described in Session 5 converts the expected 3.3 to 3.

b. 1

The closest divisor to 10 / 3 is 3 (same as 1a). 10 – (3 * 3) is 1, the remainder after division.

c. 10

`n1++` evaluates to the value of the variable before it is incremented. After the expression is evaluated, `n1` = 11.

d. 11

`++n` increments the value of the variable before it is returned.

e. 1

This is the same as `n1 = n1 % 3`

f. 0

This is the same as `n1 = n1 - n1`. `n1 - n1` is always zero.

g. –10

The unary plus (+) operator has no effect. In particular, it does not change the sign of a negative number.

3. No difference.

The increment clause of the `if` statement is considered a separate expression. The value of `i` after a preincrement or a postincrement is the same (it's only the value of the expression itself that is different).

4.
```
int cube(int n)
{
    return n * n * n;
}
```

5. The double 3.0 is demoted to the integer 3 and the result is returned from `cube(int)` as the integer 9.

6. Error #1: The program doesn't compile properly because the function `squareRoot()` has been declared as returning a `void`. Change the return type to `int` and rebuild.

   Error #2: The program now figures that the square root of 16 is 6! To sort out the problem, I break the compound `if` in two so that I can output the resulting value:

```
// loop forever
for(;;)
{
    // check the square of the current
    // value (and increment to the next)
    int nTest = nRoot++ * nRoot;
    cout << "Test root is "
         << nRoot
         << " square is "
         << nTest
         << "\n";
    if (nTest > n)
    {
        // close as we can get, return
        // what we've got
        return nRoot;
    }
}
```

   The result of executing the program is shown below.

```
This program doesn't work!
Test root is 2 square is 1
Test root is 3 square is 4
Test root is 4 square is 9
Test root is 5 square is 16
Test root is 6 square is 25
Square root of 16 is 6
```

   The program output is not correct at all; however, careful examination reveals that the left side is one off. Thus, the square of 3 is 9, but the displayed value of `nRoot` is 4. The square of 4 is 16 but the displayed value of `nRoot` is 5. By incrementing `nRoot` in expression, `nRoot` is one

more than the nRoot actually used in the calculation. Thus, nRoot needs to be incremented after the if statement.

The new squareRoot() function becomes

```
// loop forever
for(;;)
{
    // check the square of the current
    // value (and increment to the next)
    int nTest = nRoot * nRoot;
    cout << "Test root is "
         << nRoot
         << " square is "
         << nTest
         << "\n";
    if (nTest > n)
    {
        // close as we can get, return
        // what we've got
        return nRoot;
    }

    // try the next value of nRoot
    nRoot++;
}
```

The autoincrement has been moved until after the test (the autoincrement always looked fishy where it was). The output of the new, improved program is shown below.

```
This program doesn't work!
Test root is 1 square is 1
Test root is 2 square is 4
Test root is 3 square is 9
Test root is 4 square is 16
Test root is 5 square is 25
Square root of 16 is 5
```

The square is being calculated, but for some reason the function doesn't stop when nRoot is equal to 4. After all, 4 * 4 == 16. This is exactly the problem — the comparison is for nTest > n when it should be nTest >= n. The corrected program generates the desired result as shown below.

```
This program works!
Test root is 1 square is 1
Test root is 2 square is 4
Test root is 3 square is 9
Test root is 4 square is 16
Square root of 16 is 4
```

After checking several other values, I convince myself that the function does work and I remove the output statements.

Saturday Afternoon

1. The following `MyStrchr` represents my solution to the problem:

```
// myStrchr - return the index of a test
//            character in a string. Return
//            a -1 if the character not found.
int myStrchr(char target[], char testChar)
{
    // loop through the character string;
    // stop if we hit the end of the string
    int index = 0;
    while(target[index])
    {
        // if the current member of the
        // string matches the target
        // character...
        if (target[index] == testChar)
        {
            // ...then exit
            break;
        }

        // skip to the next character
        index++;
    }

    // if we ended up at the end of the
    // string without encountering the
    // character...
```

```
    if (target[index] == '\0')
    {
        // ...return a -1 rather than
        // the length of the array
        index = -1;
    }

    // return the index calculated
    return index;
}
```

The actual `myStrchr()` function begins by iterating through the string `target` stopping when the current character, referenced by `target[index]`, is equal to 0 meaning that the program has reached the end of the string. This test safeguards that the function doesn't go too far if the test character is not encountered.

Within this loop, the program compares the current character to the desired character contained in `testChar`. If the character is found, the function exits the loop prematurely.

Once outside the loop, the program terminated either because the end of the string was encountered or because it found the desired character. If the end of the string was the reason for exit then `target[index]` will equal 0, or '\0' to use the character equivalent. In that case, `index` is forced to –1.

The resulting `index` is returned to the caller.

The result of executing this program is shown below.

```
The offset of c in abcdefg is 2
The offset of a in abcdefg is 0
The offset of g in abcdefg is 6
The offset of c in abcdefg is 2
The offset of x in abcdefg is -1
Press any key to continue
```

The above program could have been simplified by simply exiting from within the loop if the character were found:

```
int myStrchr(char target[], char testChar)
{
    // loop through the character string;
    // stop if we hit the end of the string
    int index = 0;
```

```
    while(target[index])
    {
        // if the current member of the
        // string matches the target
        // character...
        if (target[index] == testChar)
        {
            // ...then return the index
            return index;
        }

        // skip to the next character
        index++;
    }

    // if we exited the loop then we must
    // have encountered the end without
    // finding the desired character
    return -1;
}
```

If the target character is found, the index value is returned immediately. If control exits the loop, it can only mean that the end of the string was encountered without finding the desired character.

Personally, I find this style more straightforward; however, some organizations have a rule against multiple returns in a single function.

2. A `double` occupies 8 bytes whereas an `int` occupies only 4 bytes. It's entirely possible that C++ would use 4 of the 8 bytes to store the integer 10, leaving the other 4 bytes unused. This would not cause an error; however, you should not assume that your compiler would work this way.

3.
```
void displayString(char* pszString)
{
    while(*pszString)
    {
        cout << *pszString;
        pszString++;
    }
}
```

4. The output of `pmc->n1` and `pmc->n2` are garbage because the pointer `pmc` was not initialized to point to anything. In fact, the program might abort without generating any output due to a bad pointer.

Sunday Morning

1.
```
class Vehicle
{
  public:
    Vehicle(int nWheels)
    {
    }
};

class Motor
{
  public:
    Motor(int nCylinders)
    {
    }
};

class Car : pubic Vehicle
{
  public:
    Car(int nCyliners, int nWheels)
      : Vehicle(nWheels), motor(nCylinders)
    {
    }

    Motor motor;
};
```