APPENDIX A

Answers to Part Reviews

Friday Evening Review Answers

1.

```
1.  Grab jack handle
2.  While car not on ground
3.      Move jack handle down
4.      Move jack handle up
5.  Release jack handle
```

This solution is simpler than the program presented for removing wheels. There is no need to begin by grabbing the jack. Because the car is already in the air, being held up by the jack, we can assume that we know where the jack is.

The loop in steps 2 through 4 moves the jack handle up and down until the car has been lowered to the ground. Step 5 completes the program by releasing the handle of the jack so that the processor's hand is available for the next steps.

2. Removing the minus sign between the 212 and 32 causes Visual C++ to generate the incorrect “missing ;” error message.

3. I fixed the problem by adding a semicolon after the 212 and rebuilt. Visual C++ built the “corrected” program without complaint.

4. The resulting corrected program calculates a Fahrenheit temperature of 244, which is obviously incorrect.

5. Apparently both “nFactor = 212;” and “32;” are legal commands. However, the resulting incorrect value for *nFactor* results in an incorrect conversion calculation.
6. Forgetting a minus sign is an understandable mistake for a poor typist such as I. Had Visual C++ corrected what it thought was a simple missing semicolon, the program would have compiled and executed without complaint. I would have had to search through the program to find the error in my calculations. Such an error would have thrown suspicion on the formula used to convert Celsius degrees to Fahrenheit when a simple typo is the real problem. This represents a waste of time when the error message generated by Visual C++, although incorrect, vectors my attention directly to the problem.
7. Rebuilding Conversion.cpp after removing the quote symbol generates this error message:

   ```
   Conversion.cpp(29) Error: unterminated string or character
   constant
   ```

   This error message indicates that GNU C++ thinks that the error occurred on line 29. (That’s the significance of the “29” in the error message itself.)
8. Because GNU C++ didn’t see a quote symbol to terminate the string on line 26, it thought that the string continued to the next quote that it saw, which was on line 29. (This quote is actually the beginning of another string, but GNU C++ doesn’t know that.)
9. If GNU C++ were to try to fix the quote problem on its own, it might do one of two things. It could add an extra quote at the point that it detected the problem, which is the open quote on line 29, thinking that the open quote should be terminated. Alternatively, it could remove the quote that it found there thinking that it was put there in error. Both solutions “fix” the problem to the extent that rebuilding the resulting program generates no build errors. Try it: add an extra quote to line 29 or remove the open quote that is there and rebuild. As it turns out, both solutions work.
10. With an input of 100 degrees Celsius, the program generates the completely nonsensical output shown below.

    ```
    C:\wecc\Programs\lesson3>exit
    Enter the temperature in Celsius:100
    Fahrenheit value is:;
    ```

```
    cout << nFahrenheit;

    cout << Time to return_
```

11. The GNU C++ solution masks the real problem. The resulting output from the program is nonsensical and, therefore, gives no hint as to the real problem. With such output, it is difficult to even get started fixing the problem unless the programmer knows that there is a syntax error in the source code that was "fixed" by the compiler.

(Fortunately, the output from this program does lead the knowledgeable programmer in the direction of the problem; however, not nearly as well as an error message during the build.)

12. In general, when a compiler attempts to fix a problem automatically it ends up creating a new problem that is more difficult to trace back to the original problem.

13.
- **a.** Yes
- **b.** No — variables must begin with a letter.
- **c.** No — variable names may not contain spaces.
- **d.** Yes — variable names may contain digits.
- **e.** No — variables cannot begin with an ampersand.

14.
```
// Sum     - output the sum of three integers
//           input from the user
#include <stdio.h>
#include <iostream.h>

int main(int nArg, char* nArgs[])
{
    int nValue1;
    int nValue2;
    int nValue3;
    cout << "Input three numbers (follow each with newline)";
    cout << "#1:";
    cin  > nValue1;

    cout << "#2:";
    cin  > nValue2;

    cout << "#3:";
    cin  > nValue3;

    cout << "The sum is:";
    cout << nValue1 + nValue2 + nValue3;

    cout << "\n";
    return 0;
  }
```

Extra credit:

replace

```
cout << nValue1 + nValue2 + nValue3;
```

with

```
cout << (nValue1 + nValue2 + nValue3)/3;
```

Saturday Morning Review Answers

1. a. Dividing nTons by 1.1 causes a rounding-off error.
   b. 2/1.1 equals 1.8 which rounds off to 1. The function returns 1000 kg.
   c. The result of dividing 2 by 1.1 is a double-precision floating point. Assigning this value to the integer `nLongTons` results in a demotion that the compiler should note.

   Bonus: "Assign `double` to `int`. Possible loss of significance." or words to that effect.
2. The following function suffers much less from rounding off than its predecessor:

```
int ton2kg(int nTons)
{
return (nTons * 1000) / 1.1;
}
```

3. a. 80 tons converts to roughly 4,500,000,000 g. This is beyond the range of an `int` on an Intel-based PC.
   b. The only possible solution is to return a `float` or a `double` instead of an `int`. The range of a `float` is much larger than that of an `int`.

4. **a.** false

b. true

c. true

d. indeterminate but probably true

You cannot depend on two independently calculated floating-point variables to be equal.

e. false

The value of `n4` is 4. Because the left hand side of the `&&` is false, the right hand side is not evaluated and the == is never executed (see section on short circuit evaluation)

5. **a.** 0x5D

b. 93

c. $1011\ 1010_2$

It's easiest to perform the addition in binary. Just remember the rules:

1. $0+0 \rightarrow 0$

2. $1+0 \rightarrow 1$

3. $0 + 1 \rightarrow 1$

4. $1 + 1 \rightarrow 0$ carry the 1.

Alternatively convert 93 * 2 = 186 back to binary.

Bonus: $0101\ 1101_2$ * 2 has the same bit pattern shifted to the left one position and a 0 shoved into the empty position on the right.

d. $0101\ 1111_2$

Convert 2 into binary $0000\ 0010_2$ and OR the two numbers together.

e. true

The $0000\ 0010_2$ bit is not set; thus, ANDing the two together results in a 0.

6. Remember that C++ ignores all white space, including tabs. Thus, while the `else` clause appears to belong to the outer `if` statement, it actually belongs with the inner `if` statement, as if it had been written:

```
int n1 = 10;
if (n1 > 11)
{
    if (n1 > 12)
    {
        n1 = 0;
    }
    else
    {
        n1 = 1;
    }
}
```

The outer `if` statement has no `else` clause and, thus, `n1` remains unchanged.

7. Because the `n1` is not less than 5, the body of the `while()` control is never executed. In the case of the `do...while()` control, the body is executed once even though `n1` is no less than 5. In this case, `n1` ends up with a value of 11.

The difference between the two loops is that the `do...while()` always executes its body at least once, even if the conditional clause is false from the beginning.

8.

```
double cube(double d)
{
    return d * d * d;
}
```

Because the Intel processor in your PC handles integers and floating-point variables differently, the machine code generated by the two functions `cube(int)` and `cube(double)` are completely different.

9. The expression `cube(3.0)` matches the function `cube(double)`; thus, `cube(double)` is passed the double value 3.0 and returns the double value 9.0, which is demoted to 9 and assigned to the variable `n`. Although the result is the same, how you got there is different.

10. The compiler generates an error because the first `cube(int)` function and the final function have identical names. Remember that the return type is not part of the function's full name.

Saturday Afternoon Review Answers

1.
```
class Student
{
    public:
        char szLastName[128];
        int nGrade; // 1->first grade, 2->second grade, etc.
        double dGPA;
}
```

2.
```
void readAndDisplay()
{
    Student s;

    // input student information
    cout << “Enter the student’s name:”;
    cin.getline(s.szLastName);
    cout << “Grade (1-> first grade, 2->second...)\n”;
    cin  > s.nGrade;
  cout << “GPA:”;
    cin  > s.dGPA;

    // output student info
    cout << “\nStudent information:\n”;
    cout << s.szLastName << “\n”;
    cout << s.nGrade << “\n”;
    cout << s.dGPA << “\n”;
}
```

3.
```
void readAndDisplayAverage()
{
    Student s;

    // input student information
    cout << “Enter the student’s name:”;
    cin.getline(s.szLastName);
    cout << “Grade (1-> first grade, 2->second...)\n”;
    cin  > s.nGrade;
    cout << “GPA:”;
```

```
        // enter in three GPAs to be averaged:
        double dGrades[3];
        cin > dGrade[0];
        cin > dGrade[1];
        cin > dGrade[2]
        s.dGPA = (dGrade[0] + dGrade[1] + dGrade[2]) / 3;

        // output student info
        cout << "\nStudent information:\n";
        cout << s.szLastName << "\n";
        cout << s.nGrade << "\n";
        cout << s.dGPA << "\n";
    }
```

4. **a.** 16 bytes (4 + 4 + 8)

b. 80 bytes (4 * 20)

c. 8 (4 + 4) Remember that the size of a pointer is 4 bytes irrespective of what it points at.

5. **a.** Yes.

b. It allocates memory from the heap but doesn't return it before exiting (this is known as a memory leak).

c. Every time the function is called, another piece of memory is lost until the heap is exhausted.

d. It may take quite awhile for the memory to be consumed. Such a small memory leak may require executing the program for many hours just to detect the problem.

6. `dArray[0]` is at 0x100, `dArray[1]` is at 0x108, and `dArray[2]` is at 0x110. The array stretches from 0x100 up to 0x118.

7. Assignment 1 has the same effect as `dArray[1] = 1.0;`. The second assignment destroys the floating point value stored in `dArray[2]`, but is otherwise not fatal because a 4-byte integer value fits within the 8 bytes allocated for a double.

8.
```
    LinkableClass* removeHead()
    {
        LinkableClass* pFirstEntry;
        pFirstEntry = pHead;
        if (pHead != 0)
        {
            pHead = pHead->pNext;
        }
        return pFirstEntry;
    }
```

The `removeHead()` function first checks whether the head pointer is null. If it is, then the list is already empty. If it is not, then `removeHead()` stores the first entry locally in `pFirstEntry`. It then moves `pHead` down to the next entry. Finally, the function returns `pFirstEntry`.

9.
```
    LinkableClass* returnPrevious(LinkableClass* pTarget)
    {
        // return a null if the list is empty
        if (pHead == 0)
        {
            return 0;
        }

        // now iterate through the list
        LinkableClass* pCurrent= pHead;
        while(pCurrent->pNext)
        {
            // if the next pointer of the current
            // entry is equal to pTarget...
            if (pCurrent->pNext == pTarget)
            {
                // ...then return pCurrent
                return pCurrent;
            }
        }

        // if we make it through the entire list without
        // finding pTarget then return a null
        return 0;
    }
```

The `returnPrevious()` function returns the address of the entry in the list immediately prior to `*pTarget`. The function begins by checking that the linked list is not empty. If it is, then there is no previous entry and the function returns a null.

returnPrevious() then iterates through the list each time saving the address of the current entry in the variable pCurrent. On each pass through the loop, the function checks whether pCurrent's next pointer points to pTarget. If it does, then the function returns pCurrent.

If returnPrevious() makes it all the way through the list without finding pTarget, then it returns a null.

10.
```
LinkableClass* returnTail()
{
    // return the entry immediately prior to the
    // end; i.e., return the entry whose next
    // pointer is null.
    return returnPrevious(0);
}
```

The entry immediately prior to the null is the last entry in the list.

Extra credit:

```
LinkableClass* removeTail()
{
    // find the last entry in the list; if it's null
    // then the list is empty
    LinkableClass* pLast = returnPrevious(0);
    if (pLast == 0)
    {
        return 0;
    }

    // now find the entry that points to this last
    // entry
    LinkableClass* pPrevious = returnPrevious(pLast);

    // if pPrevious is null...
    if (pPrevious == 0)
    {
        // ...then pLast is the only entry;
        // set the head pointer to null
        pHead = 0;
    }
    else
    {
        // ...otherwise, remove pLast from pPrevious
        pPrevious->pNext = 0;
    }

    // either way, return the last pointer
    return pLast;
}
```

The `removeTail()` function removes the last entry in a linked list. It begins by finding the address of the last entry by calling `returnPrevious(0)`. It stores this address in `pLast`. If `pLast` is null, then the list is empty and the function returns a `null` immediately.

Finding the last entry is not enough. To remove the last entry, `removeTail()` must find the entry prior to `pLast` so that it can unlink the two.

`removeTail()` finds the address of the entry prior to `pLast` by calling `returnPrevious(pLast)`. If there is no previous entry, then the list must have only one entry. `removeTail()` zeros the appropriate next pointer before returning `pLast`.

11. To demonstrate this solution, I used Visual C++ as a comparison to `rhide`.

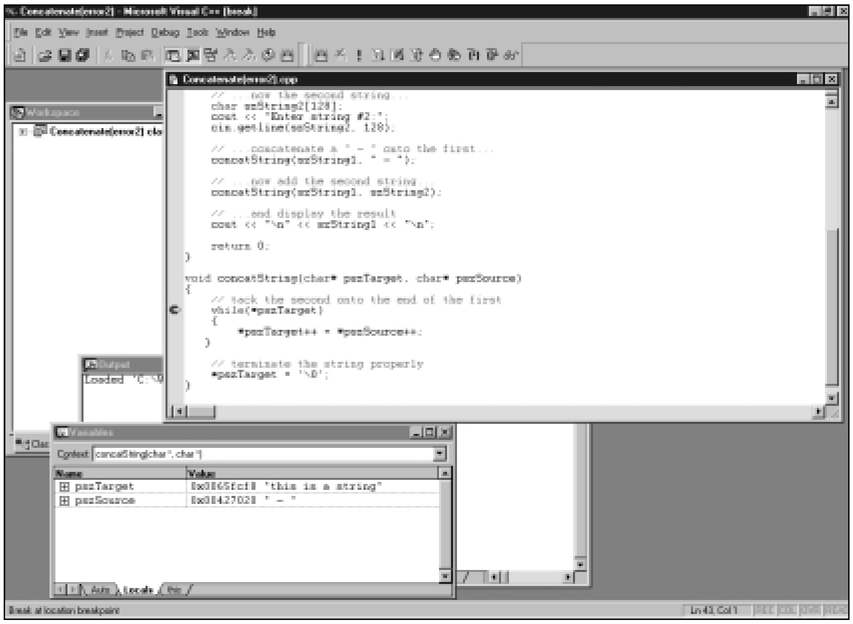
I begin by executing the program just to see what happens. If the program appears to work, I don't stop there, but I would like to know what I'm up against. The results of entering the standard "this is a string" and "THIS IS A STRING" inputs are shown below.

```
This program concatenates two strings
<this version doesn't work.>

Enter string #1:This is a string
Enter string #2:THIS IS A STRING

THI
Press any key to continue
```

Because I'm reasonably sure that the problem is in `concatString()`, I set a breakpoint at the beginning of the function and start over. After encountering the breakpoint, the target and source strings seem correct as shown in Figure A-1.



```
    // ...now the second string...
    char szString2[128];
    cout << "Enter string #2:";
    cin.getline(szString2, 128);

    // ...concatenate a " - " onto the first...
    concatString(szString1, " - ");

    // ...now add the second string...
    concatString(szString1, szString2);

    // ...and display the result
    cout << "\n" << szString1 << "\n";

    return 0;
}

void concatString(char* pszTarget, char* pszSource)
{
    // tack the second onto the end of the first
    while(*pszTarget)
    {
        *pszTarget++ = *pszSource++;
    }

    // terminate the string properly
    *pszTarget = '\0';
}
```

Context: concatString(char *, char *)

Name	Value
⊞ pszTarget	0x0065fcf8 "this is a string"
⊞ pszSource	0x00427028 " - "

Break at location breakpoint

Figure A-1
The initial Variables window displays both the source and destination strings.

From the breakpoint, I begin to single-step. At first the local variables demonstrated in the Variables window seem correct. As I head into the `while` loop, however, it becomes immediately clear that the source string is about to overwrite the target string; that is, the `concatString()` function is more of an overwrite function.

The `concatString()` should have started by moving the `pszTarget` pointer to the end of the target string before starting the transfer process. This problem is not easily fixed in the debugger, so I go back and add the extra code to move the pointer to the end of the target string.

Note

Actually, it is possible to fix this problem with a little debugger sleight-of-hand. In the locals display, click the value of the pointer in the right hand column next to `pszTarget`. Whatever value is there, add 0x10 (there are 16 characters in "this is a string"). The `pszTarget` pointer now points to the end of the target string and you can proceed with the transfer.

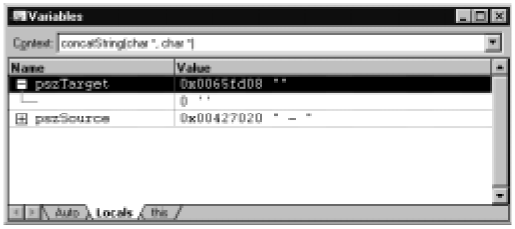
The updated `concatString()` is as follows:

```
void concatString(char* pszTarget, char* pszSource)
{
    // move pszTarget to the end of the source string
    while(*pszTarget)
    {
        pszTarget++;
    }

    // tack the second onto the end of the first
    while(*pszTarget)
    {
        *pszTarget++ = *pszSource++;
   }

    // terminate the string properly
    *pszTarget = '\0';
}
```

Replacing the breakpoint on the second `while`, the one that performs the copy, I start the program over with the same "this is a string" inputs. The local variables shown in Figure A-2 now appear correct.



Context: concatString(char *, char *)

```
Name            Value
⊟ pszTarget     0x0065fd08 ""
  └             0 ''
⊞ pszSource     0x00427020 " - "
```

Figure A-2
pszTarget should point at a null before the copy takes place.

Note

The `pszTarget` variables should point at the null at the end of the source string before the program can copy the source string.

With confidence I attempt to step through the transfer `while` loop. As soon as I press Step Over, however, the program steps over the loop entirely. Apparently, the `while` condition is not true even on the first pass through the loop. Reflecting for a moment, I realize that the condition is wrong. Rather than stop when `pszTarget` points to `null`, I should be stopping when `pszSource` points to a `null`. Updating the `while` condition as follows solves the problem:

```
while(*pszSource)
```

I start over again with the same inputs , and the debugger reveals that all seems to be OK. At completion, the program displays the proper output as well.

Saturday Evening Review Answers

1. I find shirts and pants, which are subclasses garments, which are clothing. Also in the clothing chain are shoes and pairs of socks. The pairs of socks can be further divided into pairs that match and those that don't match and so on.
2. Shoes have at least one hole for the insertion of a foot. Shoes have some type of restraint device to hold them on the foot. Shoes have some type of covering on the bottom to protect the foot from the ground. That's about all you can say about my shoes.
3. I have dress shoes and biking shoes. You can wear biking shoes to work, but it is extremely difficult to walk in them. However, they do cover your feet, and work would not come to a grinding halt.

   By the way, I also have a special pair of combination shoes. These have connections for bike pedals while retaining a conventional sole. Going to work in these hybrid shoes wouldn't be that bad at all.
4. The constructor for a linked list object should make sure that the next link points to a null when the object is constructed. It is not necessary to do anything to the static data member.

```
class Link
{
    static Link* pHead;
    Link* pNextLink;
    Link()
    {
        pNextLink = 0;
    }
};
```

The point is that the static element `pHead` cannot be initialized in the constructor lest it be reinitialized each time a `LinkedList` object is created.

5. The following is my version of the destructor and copy constructor.

```
    // destructor - remove the object and
    //              remove the entry
    ~LinkedList()
    {
        // if the current object is in a list...
        if (pNext)
        {
            // ...then remove it
            removeFromList();
        }

        // if the object has a block of memory...
        if (pszName)
        {
            // ...return it to the heap
            delete pszName;
        }
        pszName = 0;
    }

    // copy constructor - called to create a copy of
    //                    an existing object
    LinkedList(LinkedList& l)
    {
        // allocate a block of the same size
        // off of the heap
        int nLength = strlen(l.pszName) + 1;
        this->pszName = new char[nLength];

        // now copy the name into this new block
        strcpy(this->pszName, l.pszName);

        // zero out the length as long as the object
        // is not in a linked list
        pNext = 0;
    }
```

If the object is a member of a linked list, then the destructor must remove it before the object is reused and the `pNext` pointer is lost. In addition, if the object "owns" a block of heap memory, it must be returned as well. Similarly the copy constructor performs a deep copy by allocating a new block of memory off of the heap to hold the object name. The copy constructor does not add the object onto the existing linked list (though it could).

Sunday Morning Review Answers

1. The output sequence is:

```
Advisor:student data member
Student
Advisor:student local
Advisor:graduate data member
Graduate Student
Advisor:graduate student local
```

Let's step through the sequence slowly:

Control passes to the constructor for `GraduateStudent` and from there to the constructor for the base class `Student`.

The data member `Student::adv` is constructed.

Control passes to the body of the `Student` constructor.

A local object of class `Advisor` is created from the heap.

Control returns to the `GraduateStudent` constructor which constructs the data member `GraduateStudent::adv`.

Control enters the constructor for `GraduateStudent` which subsequently allocates an `Advisor` object off of the heap.

2.

```
// PassProblem - use polymorphism to decide whether
//               a student passes using different
//               criteria for pass or fail
#include <stdio.h>
#include <iostream.h>

class Student
{
  public:
    virtual int pass(double dGrade)
    {
        // if passing grade...
        if (dGrade > 1.5)
        {
            // ...return a pass
            return 1;
        }
        // ...otherwise return a fail
        return 0;
    }
};
class GraduateStudent : public Student
{
  public:
    virtual int pass(double dGrade)
    {
        if (dGrade > 2.5)
        {
            return 1;
        }
        return 0;
    }
};
```

3. My version of the `Checking` class is as follows:

```
// Checking - same as a savings account except that
//            withdrawals cost a dollar
class Checking : public CashAccount
{
  public:
    Checking(unsigned nAccNo,
             float fInitialBalance = 0.0F)
       : CashAccount(nAccNo, fInitialBalance)
    {
    }

    // a savings account knows how to process
    // withdrawals as well (don't worry about overdrafts)
    virtual void withdrawal(float fAmount)
    {
        // take out withdrawal
        fBalance -= fAmount;

        // now take out charge
        fBalance -= 1;
    }
};
```

The entire program is on the CD-ROM under the name AbstractProblem.

4. My solution to the problem is:

```
// MultipleVirtual - create a PrinterCopier class by
//                   inheriting a class Sleeper and
//                   a class Copier
#include <stdio.h>
#include <iostream.h>
class ElectronicEquipment
{
  public:
    ElectronicEquipment()
    {
        this->nVoltage = nVoltage;
    }
    int nVoltage;
};
class Printer : virtual public ElectronicEquipment
```

```
{
  public:
    Printer() : ElectronicEquipment()
    {
    }
    void print()
    {
       cout << “print\n”;
    }
};
class Copier : virtual public ElectronicEquipment
{
  public:
    Copier() : ElectronicEquipment()
    {
    }
    void copy()
    {
       cout << “copy\n”;
    }
};
class PrinterCopier : public Printer, public Copier
{
  public:
    PrinterCopier(int nVoltage) : Copier(), Printer()
    {
        this->nVoltage = nVoltage;
    }
};

int main(int nArgs, char* pszArgs)
{
   PrinterCopier ss(220);

   // first let’s print
   ss.print();

   // now let’s copy
   ss.copy();
```

```
    // now let’s output the voltage
    cout << “voltage = “ << ss.nVoltage << “\n”;

    return 0;
}
```

Sunday Afternoon Review Answers

1. My version of the two functions are as follows:

```
MyClass(MyClass& mc)
{
    nValue = mc.nValue;
    resource.open(nValue);
}
MyClass& operator=(MyClass& s)
{
resource.close();

    nValue = s.nValue;
    resource.open(nValue);
}
```

   The copy constructor opens the current object with the value of the source object.

   The assignment operator first closes the current resource because it was opened with a different value. It then reopens the resource object with the new value assigned it.

2. My class and inserter are as follows:

```
// Student - your typical beer-swilling undergraduate
class Student
{
    friend ostream& operator<<(ostream& out, Student& d);
  public:
    Student(char* pszFName, char* pszLName, int nSSNum)
    {
        strncpy(szFName, pszFName, 20);
        strncpy(szLName, pszLName, 20);
        this->nSSNum = nSSNum;
    }

  protected:
    char szLName[20];
    char szFName[20];
    int  nSSNum;
};

// inserter - output a string description
//            (this version handles the case of cents
//            less than 10)
ostream& operator<<(ostream& out, Student& s)
{
    out << s.szLName
        << ", "
        << s.szFName
        << "("
        << s.nSSNum
        << ")";
    return out;
}
```