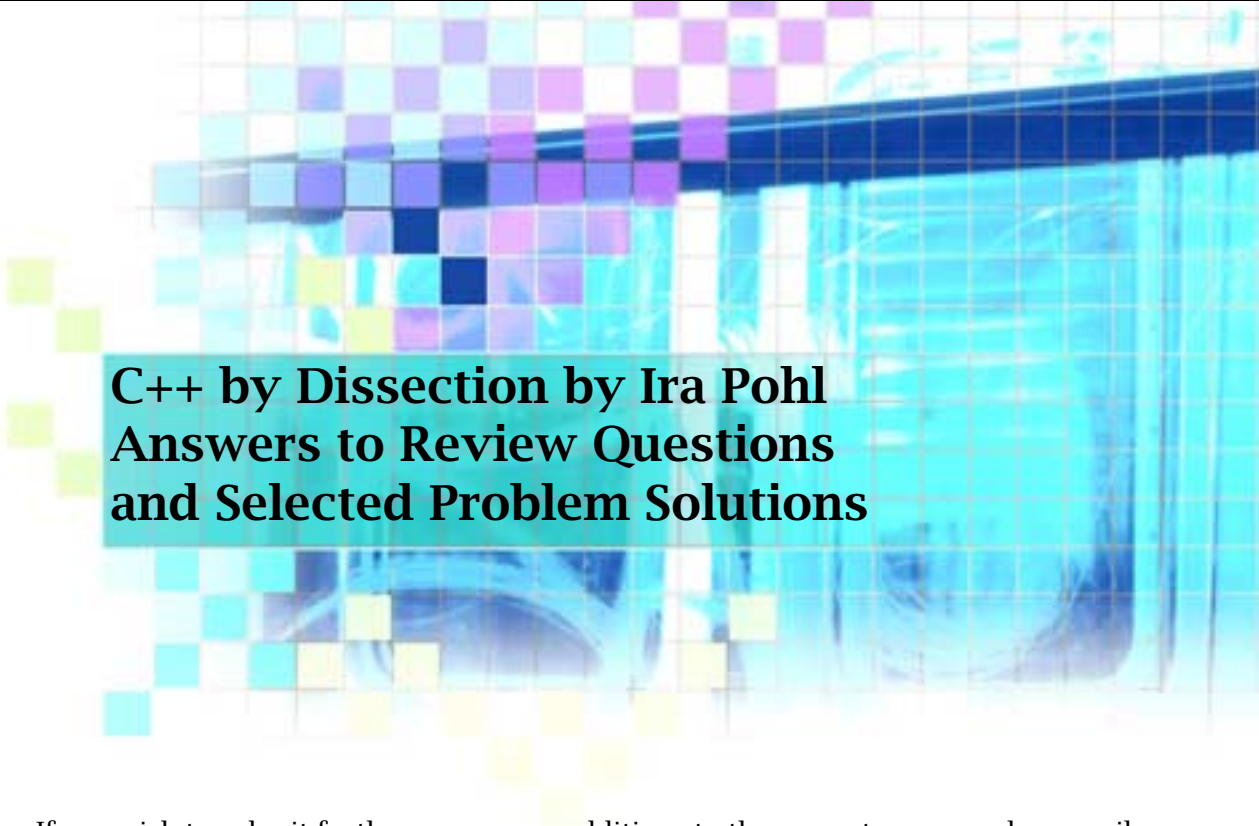

C++ by Dissection by Ira Pohl
Answers to Review Questions and Selected Problem Solutions

If you wish to submit further answers or additions to the current ones, send an email to Ira Pohl at pohl@cse.ucsc.edu and include your name as you would like it to appear in the credit for the answer.

Errata for the book can be found at Ira's website at www.cse.ucsc.edu/~pohl. Click books and then click on the picture of the C++ by Dissection book cover.

CHAPTER 1 ANSWER TO REVIEW QUESTIONS

1. C++ uses the operators << and >> for **output** and **input**, respectively.

2. A step-by-step procedure that accomplishes a desired task is called an **algorithm**.

3. The operating system consists of programs and has two main purposes. First, the operating system oversees and coordinates **the resources** of the machine as a whole. Second, the operating system provides **tools to users**.

4. The compiler takes **source** code and produces **object** code.

5. In the code `std::cout`, `cout` is **the standard output stream**, `::` is the **scope resolution operator** and `std` is **the namespace for the standard library**.

6. A **flowchart** is a graphical means for displaying an algorithm.

7. `int price, change, dimes, pennies;`
   This declares four **integer variables**. These hold the **values** to be manipulated.

8. The text uses **Bell Labs** style. There is **one blank line** following the `#include`s, and one between the declarations and statements in the body of `main()`. An **indentation** of two, three, four, five, or eight spaces is common.

9. In Windows and in UNIX, a **control-c** is commonly used to effect an interrupt.

10. A common error is **to misspell a** variable name or forget to **declare** it.

Solutions for Exercises in Chapter 1

Answer to Chapter 1 Exercise 1

Write on the screen the words

```
she sells sea shells by the seashore
```

(a) all on one line, (b) on seven lines, and (c) inside a box.

In file ch01ex01a.cpp

```
// she sells ... C++
// by Olivia Programmer

#include <iostream>       // I/O library
using namespace std;

int main()
{
   cout << "she sells sea shells by the seashore" << endl;
}
```

In file ch01ex01b.cpp

```
/ she sells ...C++
// by Olivia Programmer

#include <iostream>       // I/O library
using namespace std;

int main()
{
   cout << "she\nsells\nsea\nshells\nby\nthe\nseashore" << endl;
}
```

In file ch01ex01c.cpp

```
// she sells ...C++
// by Olivia Programmer

#include <iostream>       // I/O library
using namespace std;

int main()
{
   cout <<"----------------------------------------"
        << endl;
   cout << "|she sells sea shells by the seashore   |"
        << endl;
   cout <<"----------------------------------------"
        << endl;
}
```

ANSWER TO CHAPTER 1 EXERCISE 2

Here is part of a program that begins by having the user input three integers:

```
#include <iostream>
using namespace std;

int main()
{
   int   a, b, c, sum;

   cout << "Enter three integers:  ";
   .....
```

Complete the program so that when the user executes it and types in 2, 3, and 7, this is what appears on the screen:

```
Enter three integers:  2  3  7
Twice the sum of your integers plus 7 is 31 - bye!
```

In file ch01ex02.cpp

```
#include <iostream>
using namespace std;

int main()
{
   int   a, b, c, sum;

   cout << "Enter three integers:  ";
   cin >> a >> b >> c ;
   cout << “Twice the sum of your integers plus 7 is “;
   cout << 2*(a + b + c) + 7 << “ - bye!” << endl;
}
```

ANSWER TO CHAPTER 1 EXERCISE 3

Rewrite this program so that it prints your name instead.

In file ch01ex03.cpp

```
// Print I R A

#include <iostream.h>
int main()
{
cout << "I RRRRR    A"   << endl;
cout << "I R   R   A A"  << endl;
cout << “I R  R   A   A" << endl;
cout << "I R   R A     A "<<endl;
cout << endl;

// Print P O H L

cout << "PPPP OOOOO H   H L "     << endl;
cout << "P  P O   O H   H L "     << endl;
cout << "P P  O   O HHHHH L "     << endl;
cout << "P    O   O H   H L "     << endl;
cout << "P    OOOOO H   H LLLLL"  << endl;
cout << endl << endl;
cout << " By Ira Pohl " << endl << endl;
}
```

Answer to Chapter 1 Exercise 4

The purpose of this exercise is to help you become familiar with some of the error messages produced by your compiler. You can expect some error messages to be helpful and others to be less so. Correct each syntax error.

In file ch01ex04.cpp

```
// Full of syntax mistakes.

#include <iostreem>   //iostream
using namespace st;   //std

int main()
{
   int   a = 1, b = 2, c = 3;    //3; not ,

   cout << a + b << " = a + b";  // a + b << (remove *)
   cout << "\nc = " << c;
}
```

Answer to Chapter 1 Exercise 5

Here is part of an interactive program that computes the sum of the value of some coins. The user is asked to input the number of half dollars, quarters, dimes, etc.

```
#include <iostream>
using namespace std;

int main()
{
   int   h,       // number of half dollars
         q,       // number of quarters
         d,       // number of dimes
         n,       // number of nickels
         p;       // number of pennies
```

```
.....
cout << "Your change will be computed."<< endl;
cout << "Enter how many half dollars.";
cin >> h;
cout << "\nEnter how many quarters.";
cin >> q;
.....
```

Complete the program, causing it to print out relevant information. For example, you may want to create output that looks like this:

```
You entered:   0 half dollars
               3 quarters
               2 dimes
               17 nickels
               1 pennies
The value of your 23 coins is equivalent to
181 pennies.
```

Notice that `pennies` is plural, not singular as it should be. After you learn about the `if-else` statement in Section 2.8.3, *The `if` and `if-else` Statements*, on page 59, you can modify your program so that its output is grammatically correct.

In file ch01ex05.cpp

```
#include <iostream>
using namespace std;

int main()
{
   int   h,       // number of half dollars
         q,       // number of quarters
         d,       // number of dimes
         n,       // number of nickels
         p;       // number of pennies
   int coins;
```

```
double change;
cout << "Your change will be computed." << endl;
cout << "Enter how many half dollars.  ";
cin >> h;
cout << "\nEnter how many quarters.  ";
cin >> q;
cout << "\nEnter how many dimes.  ";
cin >> d;
cout << "\nEnter how many nickels.  ";
cin >> n;
cout << "\nEnter how many pennies.  ";
cin >> p;

change = (h * .5) + (q * .25) + (d * .1)
                  + (n * .05) + (p * .01);
cout << "You entered:    " << h;
if (h == 1 )
   cout << " half dollar." << endl;
else
   cout << " half dollars." << endl;
cout << "                " << q;
if (q == 1 )
   cout << " quarter." << endl;
else
   cout << " quarters." << endl;
      cout << "                " << d;
if (d == 1 )
   cout << " dime." << endl;
else
   cout << " dimes." << endl;
cout << "                " << n;

if (n == 1 )
   cout << " nickel." << endl;
else
   cout << " nickels." << endl;
cout << "                " << p;
if (p == 1 )
   cout << " penny." << endl;
else
   cout << " pennies." << endl;
coins = h + q + d + n + p;
```

```
   cout << "\nThe value of your " << coins << " coin";
   if (coins != 1)
      cout << "s";
   cout << " is equivalent to "
        << change * 100;
   if (change == .01)
      cout << " penny." << endl;
   else
      cout << " pennies." << endl;
}
```

Answer to Chapter 1 Exercise 6

Modify the program that you wrote in the previous exercise so that the last line of the output looks like this:

```
The value of your 23 coins is $1.81
```

Here is the last segment of code. Note that the modulo operator is used to assure that the output doesn't look like $1.6 instead of $1.60. Check for use of plurals is also done.

In file ch01ex06.cpp

```
cout << "\nThe value of your " << coins << " coin";
if (coins != 1)
   cout << "s";
cout << " is $" << change;
if ((((int(change*100))%10) == 0) && (coins > 0))
   cout << "0";   // put on trailing 0 if ends in zero
cout << endl;
```

CHAPTER 2 ANSWER TO REVIEW QUESTIONS

1. A type in C++ that C and early C++ does not have is `bool` or `wchar_t`.

2. Three keywords in C++ that are not in C are `bool wchar_t public protected private`. Describe their use as far as you currently understand it.

   The first two are new types, and the rest are access keywords.

3. What token does the new comment style in C++ involve? Why should it be used?

```
// --rest of line comments is less error prone
// and easier to use
/* --than needs termination old style comment */
```

4. What two literal values does the `bool` type have? Can they be assigned to `int` variables? With what result?

   `true` **and** `false`**. They can be assigned to int variables where true converts to 1 and false to 0.**

5. What is the distinction between `static_cast<>` and `reinterpret_cast<>`? Which is the more dangerous? Why?

   The `static_cast<>` **is available for a conversion that is well-defined, portable, and essentially invertible. This makes it a safe cast, namely, one with predictable and portable behavior.**

   The `reinterpret_cast<>` **converts in a system dependent way that is dangerous and not portable.**

6. C++ uses the semicolon as a statement **terminator. It is also a sequence point, meaning all the actions must be accomplished before going on to the next statement.**

7. The general form of a `for` statement is

   `for` (*for-init-statement*; *condition*; *expression*)
   statement

There are two important differences between the C++ `for` and the C `for`. What are they? Explain with an example.

In C++, the `for` can have a local declaration of the `for` loop variable as in

```
for (int i = 0; i < N; .....)
```

and the second expression (here `i < N`) in the `for`-loop header is a `bool` expression.

8. The `goto` should **never** be used.

9. What happens when the condition part of the `for` statement is omitted?

   It will potentially be an infinite loop, because the condition part is interpreted as true if omitted.

10. It is customary in C++, to place an opening brace **on the same** line as the starting keyword for a statement, such as an `if` or `for`. The closing brace **lines up with the first character** of this keyword.

Solutions for Exercises in Chapter 2

Answer to Chapter 2 Exercise 1

Rewrite the `gcd()` function from Section 2.3, *Program Structure*, on page 38, with a `for` loop replacing the `while` loop.

In file ch02ex01.cpp

```
int gcd(int m, int n)      // function definition
{                          // block begin
   int  r;                 // declaration of remainder

   for (; n != 0; ) {      // for goes until n!= 0
      r = m % n;           // modulus operator
      m = n;               // assignment
      n = r;
   }                       // end for loop
   return m;               // exit gcd with value m
}
```

Answer to Chapter 2 Exercise 2

Write a program that finds the maximum and minimum integer value of a sequence of inputted integers. The program should first prompt the user for how many values will be entered. The program should print this value out and ask the user to confirm this value. If the user fails to confirm the value, she must enter a new value.

In file ch02ex02.cpp

```
int main()
{
   int  current, howMany = 0, min, max;
   char confirm;

   cout << "\nPROGRAM Find Min and Max";
   while (howMany == 0) {
      cout << "\nEnter how many numbers to compare: ";
      cin >> howMany;
      cout << "Do you want to enter " << howMany
           << " numbers? (Y or N) " << endl;
      cin >> confirm;
      if (confirm != 'Y' && confirm != 'y')
         howMany = 0;
   }   //can't get out unless we have a confirmed howMany

   for (int i = 0; i < howMany; ++i) {
      cout << "\nEnter an integer: ";
      cin >> current;
      if (i == 0)
         min = max = current; //first time in - set both
      if (min > current)
         min = current;
      if (max < current)
         max = current;
    }
      cout << howMany << " numbers were entered." << endl ;
      cout << min << " was the smallest and " << max
           << " was the largest." << endl;
}
```

ANSWER TO CHAPTER 2 EXERCISE 3

Short-circuit evaluation is an important feature. The following code illustrates its importance in a typical situation:

```
// Compute the roots of: a * x * x + b * x + c
.....
cin >> a >> b >> c;
assert(a != 0);
discr = b * b - 4 * a * c;
if (discr == 0)
   root1 = root2 = -b / (2 * a);
else if ((discr > 0) && (sqrt_discr = sqrt(discr))) {
   root1 = (-b + sqrt_discr) / (2 * a);
   root2 = (-b - sqrt_discr) / (2 * a);
}
else if (discr < 0) {       // complex roots
   .....
}
.....
```

The `sqrt()` function would fail on negative values, and short-circuit evaluation protects the program from this error. Complete this program by having it compute roots and print them out for the following values:

```
a = 1.0, b = 4.0, c = 3.0
a = 1.0, b = 2.0, c = 1.0
a = 1.0, b = 1.0, c = 1.0
```

In file ch02ex03.cpp

```
#include <iostream>
#include <assert>
#include <math>
using namespace std;

void showRoots(double a, double b, double c)
{
   double discr, root1, root2, sqrt_discr;

   cout << "\na = " << a << "   b = " << b
        << "   c = " << c << endl;
   assert(a != 0);
   discr = b * b - 4 * a * c;
```

```
   if (discr == 0) {
      root1 = root2 = -b / (2 * a);
      cout << "Discriminant is 0: root1 = root2 = "
           << root1 << endl;
   }
   else if ((discr > 0) && (sqrt_discr = sqrt(discr))) {
      root1 = (-b + sqrt_discr) / (2 * a);
      root2 = (-b - sqrt_discr) / (2 * a);
      cout << "Root1 = " << root1
           << "   root2 = " << root2 << endl;
   }
   else if (discr < 0) {       // complex roots
      cout << "Complex roots: Discriminant is " << discr << endl;
   }
}

int main()
{
   double a, b, c;

   showRoots(1.0, 4.0, 3.0);
   showRoots(1.0, 2.0, 1.0);
   showRoots(1.0, 1.0, 1.0);

   cout << "\nEnter 3 integers for roots: " ;
   cin >> a >> b >> c;
   showRoots(a, b, c);
}
```

Answer to Chapter 2 Exercise 9

Write a program to convert from Celsius to Fahrenheit. The program should use integer values and print integer values that are rounded. Recall that zero Celsius is 32 degrees Fahrenheit and that each degree Celsius is 1.8 degrees Fahrenheit.

In file ch02ex09.cpp

```
int main()
{
   int  celcius = 1;
   cout << "\nConvert from Celcius to Fahrenheit" << endl;
   cout << "You will be prompted to enter integer temperatures"
        << endl;
   cout << "Enter 0 to end the program" << endl;
   while (celcius != 0) {
      cout << "Enter a non-zero integer Celcius temperature: " ;
      cin >> celcius;
      cout << celcius << "C = " << ((celcius) * 1.8) + 32
           << "F" << endl;
   }
}
```

Answer to Chapter 2 Exercise 10

Write a program that prints whether water at a given Fahrenheit temperature would be solid, liquid, or gas. In the computation, use an enumerated type:

```
enum state { solid = STMP, liquid = LTMP, gas = GTMP };
```

In file ch02ex10.cpp

```
#include <iostream>
using namespace std;

//This program does not handle the transitional states
//at exactly 32 and 212

const int STMP = 31, LTMP = 32, GTMP = 212 ;

enum state { solid = STMP, liquid = LTMP, gas = GTMP };
```

```
int main()
{
   state  water;
   int  fahrenheit = 1;
   cout << "\nFahrenheit Water Temperatures: Solid, “
        << “Liquid or Gas " << endl;
   cout << "You must enter integer temperatures" << endl;
   cout << "Enter 0 to end the program" << endl;

   while (fahrenheit != 0) {
     cout << "Enter a non-zero integer temperature: " ;
      cin >> fahrenheit;
      cout << "At " << fahrenheit << "F, water is a " ;
      water = liquid;              //assume liquid state
      if (fahrenheit <= STMP)
         water = solid ;
      else if (fahrenheit >= GTMP)
         water = gas;
      switch (water) {
         case solid:
            cout << "solid"; break;
         case liquid:
            cout << "liquid"; break;
         case gas:
            cout << "gas";    break;
         default:
            cout << "unknown!";
      }
      cout << endl;
   }
}
```

ANSWER TO CHAPTER 2 EXERCISE 11

Write a program that accepts either Celsius or Fahrenheit and produces the other value as output. For example, input 0C, output 32F; input 212F, output 100C.

In file ch02ex11.cpp

```
int main()
{
   int temp;
   char CorF = 'C'; // so we won't exit loop on first try
   cout << "\nConvert between Celcius to Fahrenheit" << endl;
   cout << "You will be prompted to enter an integer temperature"
        << endl;
   cout << "Followed by a C or F" << endl;
   cout << "Enter anything instead of C or F to end the program"
        << endl;

   while (CorF != 'E') {
      cout << "Enter an integer temperature: " ;
      CorF = 'E';          //just in case bad input
      cin >> temp >> CorF;
      if (CorF == 'C' || CorF == 'c')
          cout << temp << "C is "
                 << ((temp) * 1.8) + 32 << "F" << endl;
      else if (CorF == 'F' || CorF == 'f')
         cout << temp << "F is "<< int((temp -32)/1.8) << "C"
              << endl;
      else   {
         CorF = 'E';
         cout << "End of input - goodbye" << endl;
      }
   }
}
```

ANSWER TO CHAPTER 2 EXERCISE 13

In the C world, more flexible file I/O is available by using the `FILE` declaration and file operations found in *stdio*. The C++ community uses *fstream*, as discussed in Section 9.5, *Files*, on page 430. Familiarize yourself with this library. Convert the *gcd* program the redirection exercise on page 41 to use *fstream*s. The program should get its arguments from the command line, as in

gcd gcd.dat gcd.ans

In file ch02ex13.cpp

```
// GCD greatest common divisor program
// modified to use file io for exercise 13
// messages go to screen, but main input and output
// are tied to files specified on command line

// no need to ask for how many - it will read to end of file

#include <iostream>
#include <fstream>
#include <cstdlib>        //for exit
using namespace std;

int gcd(int m, int n)     // function definition
{                         // block begin
   int  r;                // declaration of remainder

   while (n != 0) {       // not equal
      r = m % n;          // modulus operator
      m = n;              // assignment
      n = r;
   }                      // end while loop
   return m;              // exit gcd with value m
}

int main(int argc, char** argv)
{
   if (argc != 3) {
      cout << "Usage: " << argv[0] << " infile  outfile" << endl;
      exit(1);
   }

   ifstream  f_in(argv[1]);
   ofstream  f_out(argv[2]);

   if (!f_in) {
      cout << "Can't open input file " << argv[1] << endl;
      exit(1);
   }
    if (!f_out) {
      cout << "Can't open output file " << argv[2] << endl;
      exit(1);
   }
```

```
   int  x, y;

   cout << "\nPROGRAM GCD C++" << endl;
   while (f_in.good()) {
      f_in >> x >> y;
      f_out << "\nGCD(" << x << ", " << y << ") = "
           << gcd(x, y) << endl;
   }
}
```

Answer to Chapter 2 Exercise 16

The constant RAND_MAX is the largest integer that rand() generates. Use RAND_MAX/2 to decide whether a random number is to be heads or tails. Generate 1,000 randomly generated heads and tails. Print out the ratio of heads to tails. Is this a reasonable test to see whether rand() works correctly? Print out the size of the longest series of heads thrown in a row.

In file ch02ex16.cpp

```
#include <iostream>
#include <time>       //needed to seed random
using namespace std;

int main()
{
   int   cnt_heads = 0, cnt_tails = 0, heads_together = 0,
         max_heads_together = 0, val;

   srand(time(NULL));  /* seed the rand. no. generator */
   cout << "SIMULATED 1000 COIN TOSSES" << endl;
```

```
   for (int i = 0; i < 1000; ++i) {
      val = rand() % 2;
      if (val == 1) {
         cnt_heads++;
         heads_together ++;
      }
      else {
         cnt_tails++;
         if (heads_together > max_heads_together)
            max_heads_together = heads_together;
         heads_together = 0;
      }
   }

// Note: we need to cast either heads or tails to get float not int
// on the ratio of heads to tails

   cout << "\nNumber of heads is " << cnt_heads
      << "\nNumber of tails is " << cnt_tails
      << "\nRatio is " << cnt_heads/(static_cast<double>(cnt_tails))
      << "\nMost heads together is " << max_heads_together
      << endl;
}
```

CHAPTER 3 ANSWER TO REVIEW QUESTIONS

1. If not explicitly returned, the value **0** is returned by `main()`.

2. Replace `#define ABS(X) ((X <0) ? -X: X)` with inline code.

   ```
   inline unsigned abs(int x) {return (x< 0)? -x : x;}
   ```

3. Discuss the difference between using the macro `ABS(f(y))` and the equivalent inline call. Assume that `f(y)` calls a nontrivial function.

   Macro expansion provides no type safety as is given by the C++ parameter-passing mechanism. Since the macro argument has no type, no assignment type conversions are applied to it, as they would be in a function. Also there is the possibility of unexpected side-effects when using macros.

4. What is wrong with overloading `int foo();` and `void foo();` in the same scope? Note that the only difference in their declarations is the return types.

   The compiler chooses the function with matching types and arguments. The signature-matching algorithm determines which function is called. The signature is the list of types that are used in the function declaration, not the return type. These two functions would therefore have the identical signatures.

5. The C++ STL `vector` can be used to replace **arrays** in C and C++ programs.

6. In C, control of an `if` statement depends on whether an `if` statement expression is zero or nonzero. In C++, this condition is type `bool`.

7. In C, the function `strlen()` is found in **string.h** in C++, it is found in **cstring**. Can you think of a reason for this difference? **The use of c as the first character of the library header file in C++ indicates that this library was originally a C library. It also means that the library declarations are wrapped in the namespace std.**

8. The `bad_alloc` exception is thrown when `new` fails to properly allocate memory.

9. The operator `delete` is used in place of the *cstdlib* function `free()` to return memory to free store.

10. In C, call-by-reference requires the use of pointers, but in C++, **reference passing** may be used as well.

Solutions for Exercises in Chapter 3

Answer to Chapter 3 Exercise 4

The greatest common divisor of two integers is recursively defined in pseudocode as follows, as seen in Section 3.7, *Recursion*, on page 97:

```
GCD(m,n) is:
   if m mod n equals 0 then n;
   else GCD(n, m mod n);
```

Recall that the modulo operator in C++ is `%`. Code this routine using recursion in C++. We have already done this iteratively.

In file ch03ex04.cpp

```
int gcd(int m, int n)      // will get warning to
{                          // return a value
   if (m % n == 0)
      return n;            //return is always through here
   else
      gcd(n, (m % n));
}
```

Answer to Chapter 3 Exercise 5

We wish to count the number of recursive function calls by `gcd()`. It is generally bad practice to use globals inside functions. In C++, we can use a local `static` variable instead of a global. Complete and test the following C++ `gcd()` function:

```
int gcd(int m, int n)
{
   static int  fcn_calls = 0;      // happens once
   int         r;                  // remainder

   fcn_calls++;
   .....
}
```

In file ch03ex05.cpp

```
int gcd(int m, int n)
{                                  // return a value
   static int  fcn_calls = 1;      // happens once

   if (m % n == 0) {
      cout << "Current recursive calls: " << fcn_calls << endl;
      fcn_calls = 1;                //set to 1 for next time
      return n;                     //return is always through here
   }
   else {
      gcd(n, (m % n));
      fcn_calls++;
   }
}
```

ANSWER TO CHAPTER 3 EXERCISE 6

The following C program uses traditional C function syntax:

```
/* Compute a table of cubes. */
#include <stdio.h>
#define  N    15
#define  MAX  3.5
```

```
int main()
{
   int  i;
   double  x, cube();

   printf("\n\nINTEGERS\n");
      for (i = 1; i <= N; ++i)
         printf("cube(%d) = %f\n", i, cube(i));
   printf("\n\nREALS\n");
   for (x = 1; x <= MAX; x += 0.3)
      printf("cube(%f) = %f\n", x, cube(x));
   return 0;
}

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

The program gives the wrong answers for the integer arguments because integer arguments are passed as if their bit representation were `double`. It is unacceptable as C++ code. Recode, as a proper function prototype, and run, using a C++ compiler. C++ compilers enforce type compatibility on function argument values. Therefore, the integer values are properly promoted to `double` values.

```
// Compute a table of cubes.

const int N = 15;
const double  MAX = 3.5;

double  cube(double x)
{ return (x * x * x); }

int main()
{
   int  i;
   double  x;

   cout << "\n\nINTEGERS" << endl;
   for (i = 1; i <= N; ++i)
      cout << "cube( " << i << ") = " << cube(i) << endl;
   cout << "\n\nREALS" <<endl;
   for (x = 1; x <= MAX; x += 0.3)
      cout << "cube( " << x << ") = " << cube(x) << endl;
}
```

Answer to Chapter 3 Exercise 7

Predict what the following program prints:

```
int foo(int n)
{
   static int  count = 0;

   ++count;
   if ( n <= 1) {
      cout << " count = " << count << endl;
      return n;
   }
   else
      foo(n / 3);
}

int main()
{
   foo(21);
   foo(27);
   foo(243);
}
```

The output is

```
count = 4
count = 8
count = 14
```

Answer to Chapter 3 Exercise 11

Write the function `double maximum(double a[], int n)`. This is a function that finds the maximum of the elements of an array of `double` of size `n`. (See Section 3.21, *Software Engineering: Structured Programming*, on page 128.)

In file ch03ex11.cpp

```
double maximum(const double a[], const int n)
{
   double max = a[0];

   for (int i = 1; i < n; ++i)
   if (max < a[i])
      max = a[i];
   return max;
}
```

CHAPTER 4 ANSWER TO REVIEW QUESTIONS

1. In C++, the structure name, or **tag name**, is a type.

2. Member functions that are defined within `class` are implicitly **private**.

3. A function invocation `w1.print();` means that print is a **member** function.

4. A private member (can or cannot) **can** be used by a member function of that class.

5. The `static` modifier used in declaring a data member means that the data member is **independent of any given class variable**.

6. The preferred style is to have members of **public** access first and members of **private** access declared last in a class declaration.

7. A stack is a LIFO container. A container is a data structure whose main purpose is **to store and retrieve a large number of values**.

8. LIFO means **last-in-first-out**.

SOLUTIONS FOR EXERCISES IN CHAPTER 4

ANSWER TO CHAPTER 4 EXERCISE 2

Write a structure `point` that has three coordinates `x`, `y`, and `z`. How can you access the individual members?

In file ch04ex02.cpp

```
class point {
public:                 // place public members first
   void  print() const { cout << "(" << x << "," << y
                              << "," << z <<  ")"; }
   void  print(const string& name) const;
   void  set(double u, double v, double w)
    { x = u; y = v; z = w;}
   void  plus(point c);
private:
   double x, y, z;
};

// offset the existing point by point c

void  point::plus(point c)   // definition not inline
{
   x += c.x;
   y += c.y;
   z += c.z;
}

void point::print(const string& name) const
{
   cout << name << " (" << x << "," << y
        << "," << z << ")";
}
```

```
int main()
{
   point w1, w2;
   w1.set(0, 0.5, 1.5);
   w2.set(-0.5, 1.5, 3.0);
   cout << "\npoint w1 = ";
   w1.print();
   cout << "\npoint w2 = ";
   w2.print();
   w1.plus(w2);
   w1.print();                        // standard print
   w1.print("\npoint w1 = ");        // print with name
   cout << "\nAll Done!" << endl;
}
```

You must code member functions to access the individual members, or make the members `public`.

ANSWER TO CHAPTER 4 EXERCISE 4

The following declarations do not compile correctly. Explain what is wrong.

```
struct brother {
   char           name[20];
   int            age;
   struct sister  sib;
} a;

struct sister {
   char            name[20];
   int             age;
   struct brother  sib;
} a;
```

The declarations of `brother` and `sister` inside one another produce a circular reference. This is not allowed. One can reference the other, but they can't both reference each other.

Answer to Chapter 4 Exercise 6

Rewrite the functions `push()` and `pop()` discussed in Section 4.11, *A Container Class Example: ch_stack*, on page 188, to test that `push()` is not acting on a full `ch_stack` and that `pop()` is not acting on an empty `ch_stack`. If either condition is detected, print an error message, using `cerr`, and use `exit(1)` (in *cstdlib*) to abort the program. Contrast this to an approach using asserts.

```
void ch_stack::push(char c)
{
   if ( !full() )
       s[++top] = c;
   else {
      cerr << "\nCan't push " << c << "onto full stack" << endl;
      exit(1);
   }
}

char ch_stack::pop()
{
   if (empty()) {
      cerr << "\nCan't pop from empty stack" << endl;
      exit(1);
   }
   return (s[top--]);
}
```

Answer to Chapter 4 Exercise 7

Write `reverse()` as a member function for type `ch_stack`, discussed in Section 4.11, *A Container Class Example: ch_stack*, on page 187. Test it by printing normally and reversing the string

Gottfried Leibniz wrote Toward a Universal Characteristic

```
class ch_stack {
public:
   void  reset() { top = EMPTY; }
   void  push(char c);
   char  pop();
   char  top_of() const { return s[top]; }
   bool  empty() const { return (top == EMPTY); }
   bool  full() const { return (top == FULL); }
   void  print() const;
   void  reverse();
private:
   enum { max_len = 100, EMPTY = -1, FULL = max_len - 1 };
   char  s[max_len];
   int   top;
};

void ch_stack::push(char c)
{
   if ( !full() )
       s[++top] = c;
   else {
      cerr << "\nCan't push " << c << "onto full stack" << endl;
      exit(1);
   }
}

char ch_stack::pop()
{
   if (empty()) {
      cerr << "\nCan't pop from empty stack" << endl;
      exit(1);
   }
   return (s[top--]);
}

void ch_stack::print() const
{
   if (!empty())
      for (int i = EMPTY + 1; i <= top; ++i)
        cout << s[i];
   cout << endl;
}
```

```
void ch_stack::reverse()
{
   char temp;

   if (!empty())
     for (int i = EMPTY + 1; i < ((top + 1)/ 2); ++i) {
        temp = s[i];
        s[i] = s[top - i];
        s[top - i] = temp;
     }
}

// Reverse a string with a ch_stack
int main()
{
   ch_stack  s;
   char   str[] =
    { "Gottfreid Leibniz wrote Toward a Universal Characteristic" };
   int i = 0;

   cout << str << endl;
   s.reset();              // s.top = EMPTY; is illegal
   while (str[i] && !s.full())
      s.push(str[i++]);
   s.print();
   s.reverse();
   s.print();
   cout << endl;
}
```

ANSWER TO CHAPTER 4 EXERCISE 14

Write the following member functions and add them to the *poker* program found in Section 4.7, *An Example: Flushing*, on page 174. Let `pr_deck()` use `pr_card()` and `pr_card()` use `print()`. Print the deck after it is initialized.

```
void  pips::print() const;
void  card::pr_card() const;
void  deck::pr_deck() const;
```

In file ch04ex14.cpp

```
class pips {
public:
   void  set_pips(int n) { p = n % 13 + 1; }
   int   get_pips() const { return p; }
   void  print() const;
private:
   int  p;          // meant to hold values [1,13]
};

void  pips::print() const
{
   if (p > 1 && p < 11)
       cout << p;
   else
      switch (p) {
      case 1:
         { cout << "A"; break; }
      case 11:
         { cout << "J"; break; }
      case 12:
         { cout << "Q"; break; }
      case 13:
         { cout << "K"; break; }
      default:
          { cout << "Unknown!"; }
      }
}

class card {
public:
   void  set_card(int n)
      { s = static_cast<suit>(n/13); p.set_pips(n); }
   void  pr_card() const;
   suit  get_suit() const { return s; }
   pips  get_pips() const { return p; }
private:
   suit  s;
   pips  p;
};
```

```
void  card::pr_card() const
{
   p.print();
   switch (s) {
   case clubs:
      {cout << "C"; break; }
   case diamonds:
      {cout << "D"; break; }
   case hearts:
      { cout << "H"; break; }
   case spades:
      { cout << "S"; break; }
   default:
      {cout << "Unknown Suit!"; }
   }
}

class deck {
public:
   void  set_deck();
   void  shuffle();
   void  deal(int, int, card*);
   void  pr_deck() const;
private:
   card  d[52];
};

// print out deck so it is nicely aligned
void deck::pr_deck() const
{
   pips tempPips;

   for (int i = 0; i < 52; ++i) {
      d[i].pr_card();
      cout << " ";       //space out card output
      tempPips = d[i].get_pips();
      if (tempPips.get_pips() != 10)
         cout << " ";    //forces 4 places per card
      if ((i + 1) % 13 == 0)
         cout << endl;   //new line every 13 cards
   }
   cout << endl;         //blank line after entire deck
}
```

Answer to Chapter 4 Exercise 16

In Section 4.7, *An Example: Flushing*, on page 174, `main()` detects flushes. Write a function

```
bool  isflush(const card hand[], int nc) const;
```

that returns `true` if a `hand` is a flush.

```
bool isflush(card my_hand[], int nc)
{
   int i, sval[4] = {0, 0, 0, 0};         //array for suit counts

   for (i = 0; i < nc; ++i)
      sval[my_hand[i].get_suit()]++;      // +1 to suit
   for (i = 0; i < 4; ++i)
      if (sval[i] >= 5)                   // 5 or more is flush
         return true;
      else
         return false;
}

int main()
{
   card one_hand[9];                      // max hand is 9 cards
   deck dk;
   int  i, j, k, flush_count = 0;
   int  ndeal, nc, nhand;

   do {
      cout << "\nEnter no. cards in a hand (5-9): ";
      cin >> nc;
   } while (nc < 5 || nc > 9);
   nhand = 52 / nc;
   cout << "Enter no. of hands to deal: ";
   cin >> ndeal;
   srand(time(NULL));                     // seed rand() from time()
   dk.set_deck();
   dk.pr_deck();
```

```
   for (k = 0; k < ndeal; k += nhand) {
      for (i = 0; i < nc * nhand; i += nc) {
         if ((nhand + k) > ndeal)
            nhand = ndeal - k;
         dk.shuffle();
         dk.deal(nc, i, one_hand);          // deal next hand
         if (isflush(one_hand, nc))
           ++flush_count;
      }
   }

   dk.pr_deck();
   cout << "\nIn " << ndeal << " ";
   cout << nc << "-card hands there were ";
   cout << flush_count << " flushes\n";
}
```

CHAPTER 5 ANSWER TO REVIEW QUESTIONS

1. What is the signature in the following declaration: `void f(int x, double y);`? `(int, double)`

2. How can you disable a conversion constructor? Use the keyword `explicit`.

3. How many arguments can a user-defined conversion have? **One.**

4. Outline the signature-matching algorithm.
   1. **Use an exact match if found.**
   2. **Try standard type promotions.**
   3. **Try standard type conversions.**
   4. **Try user-defined conversions.**
   5. **Use a match to ellipsis if found.**

5. Explain how `cout << x` uses overloading and why this was important. **All of the built-in types have been overloaded for ostream so that variables can be output without having to cast them for appropriate output. The overloaded operator << is implemented in such a way that multiple output statements can be done in a single cout statement.**

6. The keyword `friend` is a function specifier. **It gives a nonmember function access to the private and protected members of the class to which it is a friend.**

7. One reason for using friend functions is for overloading operators.
   - **Some functions need privileged access to more than one class.**
   - **A friend function passes all of its arguments through the argument list, and each argument value is subject to assignment-compatible conversions.**
   - **Conversions apply to a class variable passed explicitly and are especially useful in cases of operator overloading.**

8. Binary operators, such as +, should be overloaded by _____nonmember functions because **you want a natural use of the operator for user-defined types in which the operator makes sense. *Overloading operators* allows infix expressions of both ADTs and built-in types to be written. In many instances, this important notational convenience leads to shorter, more readable programs.**

9. When a pointer operator is overloaded, it must be a **nonstatic class member** function.

10. Some operators can be overloaded only as nonstatic member functions. Name three such operators. **=, (), [], and ->**.

Solutions for Exercises in Chapter 5

Answer to Chapter 5 Exercise 1

The following table contains a variety of mixed-type expressions. To test your understanding of conversions and type, fill in the blanks.

Declarations and Initializations		

`int i = 3, *p = &i;`
`char c = 'b';`
`double x = 2.14, *q = &x;`		

Expression	**Type**	**Value**
`i + c`	`int`	101
`x + i`	`double`	5.14
`p + i`	pointer	System dependent
`p == & i`	`bool`	`true`
`* p - * q`	`double`	0.86
`static_cast<int>(x + i)`	`int`	5

Answer to Chapter 5 Exercise 2

To test your understanding use the `slist` type to code the following member functions:

```
// slist ctor with initializer char* string
slist::slist(const char* c);
```

```
// length returns the length of the slist
int slist::length() const;

// return number of elements whose data value is c
int slist::count_c(char c) const;
```

Let us redisplay the basic class code.

In file ch05ex02.cpp

```
struct slistelem {
   char       data;
   slistelem*  next;
};

class slist {              // a singly linked list
public:
   slist() : h(0) { }      // 0 denotes empty slist
   ~slist() { release(); }
   void  prepend(char c);  // adds to front of slist
   void  del();
   slistelem*  first() const { return h; }
   void  print() const;
   void  release();
private:
   slistelem*  h;          // head of slist
};
```

Here are the implementations.

In file ch05ex02.cpp

```
int slist::length() const  // object is unchanged
{
   slistelem*  temp = h;
   int  len = 0;

   while (temp != 0) {     // detect end of slist
      temp = temp -> next;
      ++len;
   }
   return len;
}
```

```
// Return number of elements whose data value is c
int slist::count_c(char c) const
{
   slistelem*  temp = h;
   int  c_cnt = 0;

   while (temp != 0) {        // detect end of slist
      if (c == temp -> data)
         ++c_cnt;
      temp = temp -> next;
   }
   return c_cnt;
}

slist::slist(const char* c) : h(0) {
   int  i = 0;
   while (c[i] != '\0')
      i++;
   while (--i >= 0)
      prepend(c[i]);
}
```

ANSWER TO CHAPTER 5 EXERCISE 4

To test your understanding, write a `rational` constructor that, given two integers as dividend and quotient, uses a greatest common divisor algorithm to reduce the internal representation to its smallest `a` and `q` value.

In file ch05ex04.cpp

```
// Overloading functions
class rational {
public:
   rational(int n = 0) : a(n),q(1) { }
   rational(int i, int j);
   rational(double r) : a(r * BIG), q(BIG){ }
   void  print() const { cout << a << " / " << q; }
   operator double()
      { return static_cast<double>(a) / q; }
private:
   long  a, q;
   enum { BIG = 100 };
};
```

```
int gcd(int m, int n)       // function definition
{                           // block
   int  r;                  // declare remainder

   while (n != 0) {         // not equal
      r = m % n;            // modulus operator
      m = n;                // assignment
      n = r;
   }                        // end while loop
   return m;                // exit gcd with value m
}

rational::rational(int i, int j)
{
   int  m = gcd(i, j);
   a = i / m;
   q = j / m;
}
```

ANSWER TO CHAPTER 5 EXERCISE 12

Test your understanding of `my_string` by implementing additional members.

In file ch05ex12.cpp

```
// Reference counted my_strings
class str_obj {
public:
   int    len, ref_cnt;
   char*  s;
   str_obj() : len(0), ref_cnt(1)
      { s = new char[1]; assert(s != 0); s[0] = 0; }
   str_obj(const char* p) : ref_cnt(1)
      { len = strlen(p); s = new char[len + 1];
        assert(s != 0); strcpy(s, p); }
   ~str_obj() { delete []s; }
};
```

```
class my_string {
public:
   my_string() { st = new str_obj; assert(st != 0);}
   my_string(const char* p)
      { st = new str_obj(p); assert(st != 0);}
   my_string(const my_string& str)
       { st = str.st; st -> ref_cnt++; }
   ~my_string();
   void  assign(const my_string& str);
   int   strcmp(const my_string& s1);
   void  print() const { cout << st -> s; }
private:
   str_obj*  st;
};

// strcmp is negative if s < s1,
//        is 0 if s == s1,
//        and is positive if s > s1
//        where s is the implicit argument
int my_string::strcmp(const my_string& s1)
{
   //use strcmp declared in iostream
   return(std::strcmp(st -> s, s1.st -> s));
}

// Print overloaded to print the first n characters
void  my_string::print(int n) const
{
   for (int i = 0; i < n; ++i)
      cout << st->s[i];
}
```

ANSWER TO CHAPTER 5 EXERCISE 16

Why would the following be buggy?

```
char& my_string::operator[](int position)
{
   return st[position];
}
```

This might not return the null string value 0. The first definition, given below, makes sure that the array that represents the string is not null terminated in an earlier position.

In file ch05ex16.cpp

```
char& my_string::operator[](int position)
{
   char* s = st -> s;
   for (int i = 0; i != position; ++i) {
      if (*s == 0)
         break;
      s++;
   }
   return *s;
}
```

CHAPTER 6 ANSWER TO REVIEW QUESTIONS

1. In C, one can use `void*` to write generic code, such as `memcpy()`. In C++, writing generic code uses the keyword `template`.

2. Rewrite as a template function the macro

```
#define  SQ(A) ((A) * (A))
```

```
template<class T> T SQ(T A) { return A * A;}
```

   Mention a reason why using the template is preferable to using the macro.

   The macro's disadvantages include type safety, unanticipated evaluations, and scoping problems. Using define macros can often work, but doing so is not type safe. Macro substitution is a preprocessor textual substitution that is not syntactically checked until later. Another problem with define macros is that they can lead to repeated evaluation of a single parameter. Definitions of macros are tied to their position in a file and not to the C++ language rules for scope.

3. Using templates to define classes and functions allows us to reuse code in a simple, type-safe manner that lets the compiler automate the process of type **instantiation** —that is, when a type replaces a type parameter that appeared in the template code.

4. A(n) **container** is an object whose primary purpose is to store values.

5. An iterator is a pointer or a pointer-like variable used for **traversing and accessing container elements**.

6. One downside is that for each use of a template function with different types, **a code body** is generated.

7. A friend function that does not use a template specification is a friend of **all instantiations of the template class**.

8. Are static template members universal or specific to each instantiation? **Specific to each instantiation**.

9. Unlike the `#define` macro, templates are **type safe** and **properly scoped**.

10. The keyword `typename` can be used inside templates instead of the keyword `class` to declare template type parameters.

SOLUTIONS FOR EXERCISES IN CHAPTER 6

ANSWER TO CHAPTER 6 EXERCISE 1

Rewrite `stack<T>` in Section 6.1, *Template Class* `stack`, on page 283, to accept an integer value for the default size of the stack. Now client code can use such declarations as

```
stack<int, 100>  s1, s2;
stack<char, 5000>  sc1, sc2, sc3;
```

Discuss the pros and cons of this additional parameterization.

In file ch06ex01.cpp

```
#include <iostream>
#include <assert>
using namespace std;

//template stack implementation
template <class TYPE, int n>
class stack {
public:
   explicit stack(int size = n)
      : max_len(size), top(EMPTY), s(new TYPE[size])
       { assert(s != 0); }
   ~stack() { delete []s; }
   void  reset() { top = EMPTY; }
   void  push(TYPE c) { s[++top] = c; }
   TYPE  pop() { return s[top--]; }
   TYPE  top_of()const { return s[top]; }
   bool  empty()const { return top == EMPTY; }
   bool  full()const { return top == max_len - 1; }
   void  print() const
      { if (!empty()) for (int i = 0; i <= top; ++i)
           cout << s[i]; }
```

```
private:
   enum   { EMPTY = -1 };
   TYPE*  s;
   int    max_len;
   int    top;
};

int main()

{
   stack<char, 30>      stk_ch;         // 30 char stack
   stack<char*, 200>    stk_str;        // 200 char* stack
   stack<int, 100>      stk_int;        // 100 integers

   stk_ch.push('A');
   stk_ch.push('B');

   stk_str.push("ABCD");
   stk_str.push("EFGH");
   stk_int.push(1);
   stk_int.push(2);

    stk_ch.print();
    cout << endl;
    stk_str.print();
    cout << endl;
    stk_int.print();
}
```

The disadvantage to this use is that for each different `n`, compiler generates a new code body for the class, which would be very inefficient. A more effecient way would be to create an array of size `n` within the stack of `int`s.

ANSWER TO CHAPTER 6 EXERCISE 4

Write a generic `exchange()` function with the following definition, and test it:

```
template<class TYPE>
void exchange(TYPE& a, TYPE& b, TYPE& c)
{
// replace a's value by b's and b's by c's
// and c's by a's
}
```

Here is the answer:

In file ch06ex04.cpp

```
template<class TYPE>
void exchange(TYPE& a, TYPE& b, TYPE& c)
{
   TYPE  temp;

   temp = a;
   a = b;
   b = temp;
   temp = b;
   b = c;
   c = temp;
}
```

Note that this does not work when contents of arrays need to be exchanged.

ANSWER TO CHAPTER 6 EXERCISE 6

For the *vect_it* program in Section 6.5, *Parameterizing the Class vector*, on page 302, write the member function template

```
<class T> void vector<T>::print()
```

This function prints the entire vector range.

In file ch06ex06.h

```
template <class T>
void vector<T>::print()
{
   for (int i = 0; i < size; ++i)
      cout << p[i] << ' ';
}
```

Answer to Chapter 6 Exercise 11

Sorting functions are natural candidates for parameterization. Rewrite the following generic bubble sort using templates.

```
void bubble(int d[], int how_many)
{
   int  temp;

   for (int i = 0; i < how_many - 2;  ++i)
      for (int j= 0; j < how_many - 1 - i; ++j)
         if (d[j] < d[j+1]) {
            temp = d[j];
            d[j]= d[j + 1];
            d[j+1] = temp;
         }
}
```

Here is the generic version:

In file ch06ex11.cpp

```
template <class T>
void bubble(T d[], int how_many)
{
   T temp;

   for (int i = 0; i < how_many - 2;  ++i)
      for (int j= 0; j < how_many - 1 - i; ++j)
         if (d[j] < d[j+1]) {
            temp = d[j];
            d[j]= d[j + 1];
            d[j+1] = temp;
         }
}
```

CHAPTER 7 ANSWER TO REVIEW QUESTIONS

1. The three components of STL are **containers, iterators, and algorithms**.

2. An iterator is like a **pointer** type in the kernel language.

3. The member `end()` is used as a guard for determining the last position in a container.

4. Name two STL sequence container classes. **vectors, lists, and deques**

5. Name two STL associative container classes. **sets, multisets, maps, and multimaps**

6. Can STL be used with ordinary array types? Explain. **Yes, if pointers are used to access the arrays, the pointers can then be used to iterate over the array.**

7. True or false: A template argument can be only a type. **False. Other template arguments include constant expressions, function names, and character strings.**

8. A nonmutating STL algorithm, such as `find()`, has the property **that they do not modify the contents of the containers they work on.**

Solutions for Exercises in Chapter 7

Answer to Chapter 7 Exercise 2

Recode `print(const list<double> &lst)` to be a template function that is as general as possible. (See Section 7.1, *A Simple STL Example*, on page 323.)

In file ch07ex02.cpp

```
#include <iostream>
#include <list>                        // list container
#include <numeric>                     // for accumulate
using namespace std;

// Using the list container
template <class Can>
void print(Can& container)
{
   Can::iterator p;     // traverse iterator

   for (p = container.begin(); p != container.end(); ++p)
      cout << *p << '\t';
   cout << endl;
}

int main()
{
   double w[4] = { 0.9, 0.8, 88, -99.99 };
   list<double> z;

   for (int i = 0; i < 4; ++i)
      z.push_front(w[i]);
   print(z);
   z.sort();
   print(z);
   cout << "sum is "
        << accumulate(z.begin(), z.end(), 0.0)
        << endl;
}
```

In this version any container will be printed. Also on p324 is a classical generalization using an iterator range.

Answer to Chapter 7 Exercise 3

Write an algorithm for `vector<> v` that adds the values stored in the elements `v[2 * i]`, the even-valued indices of the vector. Test it on ints and doubles.

In file ch07ex03.cpp

```
#include <iostream>
#include <vector>
using namespace std;

template <class Summable>
Summable sumEvenElements(vector<Summable> &vec)
{
   vector<Summable>::iterator p;
   Summable s = 0;

   for (p = vec.begin(); p != vec.end(); p += 2)
      s += *p;
   return s;
}

int d[] = { 3, 7 , -99, 0, 14, 19, 22, -34 };
double w[] = { 0.9, 0.8, 88, -99.99 };

int main()
{
   vector<int> intData(d, d+8) ;

   vector<double> dblData(w, w+4);

   cout << "sum of even elements is " << sumEvenElements(intData)
        << endl;
   cout << "sum of even elements is " << sumEvenElements(dblData)
        << endl;
}
```

Comment on Exercises 5 and 6

These two exercises erroneously require the programmer to implement an existing function in the STL. Instead of implementing `erase()` and `insert()` as member functions, write them as ordinary template functions which would act on lists.

Answer to Chapter 7 Exercise 7

In file ch07ex07.cpp

```
#include <iostream>
#include <list>
#include <vector>
#include <set>
using namespace std;

template <class Iterator>
Iterator secondLargest(Iterator b, Iterator e)
{

   Iterator max1 = b, max2 = ++b, t;
   if (*max1 < *max2){ t = max1; max1 = max2; max2 = t;}
   for (++b; b != e; ++b)
      if (*b >= *max1)
          { max2 = max1; max1 = b;}
      else if (*b > *max2)
          max2 = b;
   return max2;
}

int d[] = { 3, 7 , -99, 0, 14, 19, 22, -34 };

int main()
{
   list<int> intData(d, d+8) ;
   cout << "2nd largest is " << (*secondLargest(intData.begin(),
            intData.end())) << endl;
}
```

Answer to Chapter 7 Exercise 8

Write and test the template code for the STL library function `count_if(b, e, p, n)`, where `p` is a predicate and n is the summing variable.

In file ch07ex08.cpp

```
#include <iostream>
#include <numeric>
#include <vector>

using namespace std;

int d[] = { 3, 7 , -99, 0, 14, 19, 22, -34 };

int main()
{
   int n = 0;
   vector<int> intData(d, d+8) ;

   count_if(intData.begin(), intData.end(),
         bind2nd(less<int>(), 20), n);
   cout << n << "  are less than 20 "    << endl;
   n = 0;
   count_if(intData.begin(), intData.end(),
         bind2nd(equal_to<int>(), 14), n);
   cout << n << "  are equal to 14 "    << endl;
}
```

Notice how to use `bind2nd()` to create a predicate of one argument. We used logical function objects found in table 7.24.

CHAPTER 8 ANSWER TO REVIEW QUESTIONS

1. In `class X : Y { ····· }`, `X` is a **derived** class and `Y` is a **base** class.

2. True or false: If D inherits from B privately, D is a subtype of B. **False. In private inheritance, we reuse a base class for its code. We call private derivation a *LIKEA* relationship, or *implementation inheritance*, as opposed to the *ISA* interface inheritance.**

3. The term overriding refers to **virtual** functions.

4. An abstract base class contains a **pure virtual function**.

5. The subtyping relationship is called the **ISA relationship or implementation inheritance**.

6. True or false: Template classes cannot be base classes. **False. A template class can derive from an ordinary class, an ordinary class can derive from an instantiated template class, and a template class can derive from a template class.**

7. What is wrong with the following?

```
class A:B {·····};
class B:C {·····};
class C:A {·····};
```

   This is a circular definition and is illegal. No class may, through its inheritance chain, inherit from itself.

8. In multiple inheritance, why is virtual inheritance used? **If both classes are used as base classes in the ordinary way by their derived class, the new class has two subobjects of the common ancestor. Using virtual inheritance eliminates this problem.**

9. The class `type_info` provides a `name()` member function that **returns a string that is the type name and overloaded equality operators**.

10. True or false: Constructors, destructors, overloaded `operator=`, and friends are not inherited. **True.**

SOLUTIONS FOR EXERCISES IN CHAPTER 8

ANSWER TO CHAPTER 8 EXERCISE 1

For `student` and `person` code, input member functions that read input for each data member in their classes. (See Section 8.1, *A Derived Class*, on page 378.) Use `person::read` to implement `student::read`.

In file ch08ex01.cpp

```
class person {
public:
   person(const string& nm, int a, char g):
          name(nm), age(a), gender(g) { }
          //empty name for person to be read into
   person() : name(30, ' '), age(0), gender('M') {}
   void print() const { cout << *this << endl; }
   friend ostream& operator<<(ostream& out, const person& p);
   int get_age() const { return age; }
   void read();
protected:
   string  name;
   int  age;
   char  gender;        // male == 'M', female == 'F'
};

ostream& operator<<(ostream& out, const person& p)
{
   return (out << p.name << ", age is " << p.age
               << ", gender is " << p.gender);
}

// older can work with student as well
const person& older(const person& a, const person& b)
{
   if (a.get_age() >= b.get_age())
      return a;
   else
      return b;
}
```

```
void person::read()
{
   char tempName[30];
   cout << "\nEnter name: " ;
   cin.get(tempName, 30);          //get won't read into string type
   name = tempName;
   cin.ignore(INT_MAX, '\n');      //clear input line
   cout << "\nEnter gender: M or F";
   cin >> static_cast<char>(gender);   //more checking should be done
   cin.ignore(INT_MAX, '\n');      //clear input line after get number
   cout << "\nEnter Age: ";
   cin >> age;
   cin.ignore(INT_MAX, '\n');      //clear input line after get number
}

#include "person.h"

enum year { frosh, soph, junior, senior };
const string year_label[]= { "freshman", "sophomore",
                             "junior", "senior" };

class student: public person {
public:
   student(const string& nm, int a, char g,
           double gp, year yr)
       : person(nm, a, g), gpa(gp), y(yr) { }
   student() : person(), gpa (0), y(frosh) { }
   void print() const { cout << *this << endl; }
   friend ostream& operator<<(ostream& out,
                              const student& s);
   void read();
protected:
   double gpa;
   year y;
};

ostream& operator<<(ostream& out, const student& s) {
   return(out << static_cast<person>(s) << ", "
              << year_label[s.y] << ", gpa = "
              << s.gpa);
}
```

```
void student::read()
{
   person::read();
   cout << "\nEnter year: fresh = 1, soph = 2, "
    << "junior = 3, senior = 4, grad = 5: ";
   cin >> static_cast<int>(y);
   cin.ignore(INT_MAX, '\n');     //clear input line after get number
   y = y - 1;                     //adjust for enumeration
   cout << "\nEnter GPA: ";
   cin >> gpa;
   cin.ignore(INT_MAX, '\n');     //clear input line after get number
 }

int main()
{
   // declare and initialize
   person abe(string("Abe Pohl"), 92,'M');
   person sam(string("Sam Pohl"), 66, 'M');
   student phil(string("Philip Pohl"), 68, 'M',
                3.8, junior);
   student laura(string("Laura Pohl"), 12, 'F',
                3.9, frosh);
   student robin;
   person  kent;

   cout << abe << endl;    // info on abe is printed
   cout << phil << endl;

   robin.read();
   kent.read();
   cout << "\ndefault print: " << endl;
   robin.print();
   kent.print();
   person* ptr_person;
   ptr_person = &robin;
   cout << "\nperson only print: " << endl;
   ptr_person -> print();
   cout << endl;
}
```

Answer to Chapter 8 Exercise 2

In file ch08ex02.cpp

Pointer conversions, scope resolution, and explicit casting create a wide selection of possibilities. Using `main()`, discussed in Section 8.3, ***A Student ISA Person***, on page 382, which of the following work, and what is printed?

```
person* ptr_person;
student* ptr_student;

ptr_person = &abe;
ptr_student = &phil;

reinterpret_cast<student *>(ptr_person) -> print();
dynamic_cast<person *>(ptr_student) -> print();
ptr_person -> student::print();     //illegal use of person
ptr_student -> person::print();
reinterpret_cast<student *>(ptr_person) -> print();
```

Print out and explain the results.

```
int main()
{
   // declare and initialize
   person abe(string("Abe Pohl"), 92,'M');
   student phil(string("Philip Pohl"), 68, 'M',
                3.8, junior);

   cout << abe << endl;    // info on abe is printed
   cout << phil << endl;

   person* ptr_person;
   student* ptr_student;

   ptr_person = &abe;
   ptr_student = &phil;

   reinterpret_cast<student *>(ptr_person) -> print();
   dynamic_cast<person *>(ptr_student) -> print();
```

```
// ptr_person -> student::print();     //illegal use of person
   ptr_student -> person::print();
   reinterpret_cast<student *>(ptr_person) -> print();

   cout << endl;
}
```

The `ptr_person -> student::print()` wil not compile because `person` cannot be expanded to `student`, though `student` can be limited to `person`. The output is:

```
Abe Pohl, age is 92, gender is M
Philip Pohl, age is 68, gender is M, junior, gpa = 3.8
Abe Pohl, age is 92, gender is M, sophomore, gpa = 1.96715e-307
Philip Pohl, age is 68, gender is M
Philip Pohl, age is 68, gender is M
Abe Pohl, age is 92, gender is M, sophomore, gpa = 1.96715e-307
```

ANSWER TO CHAPTER 8 EXERCISE 7

Derive an integer vector class from the STL `class vector<int>` that has 1 as its first index value and `n` as its last index value.

```
int_vector x(n);    // vector whose range is 1 to n
```

In file ch08ex07.cpp

```
#include <iostream>
#include <vector>
using namespace std;

class int_vector : public vector<int> {
public:
   int_vector(int n) : vector<int>(n), base(begin()) { }
   int& operator[](int position) {return *(base + position - 1);}
private:
   vector<int>::iterator base;
};
```

```
int main ()
{
   int_vector v(100);     // 100 is vector's size

   for (int i = 1; i <= 100; ++i)
      v[i] = i;
   for (int i = 1; i <= 100; ++i)
      cout << v[i] << ' ';
   cout << endl;
}
```

ANSWER TO CHAPTER 8 EXERCISE 9

For the following program, explain when both overriding and overloading take place:

```
class B {
public:
   B(int j = 0) : i(j) {}
   virtual void  print() const            //override
      { cout << " i = " << i << endl; }
   void  print(char *s) const             //overload
      { cout << s << i << endl; }
private:
   int  i;
};

class D : public B {
public:
   D(int j = 0) : B(5), i(j) {}
   void  print() const                       //overload
      { cout << " i = " << i << endl; }
   int  print(char *s) const                 //overload
      { cout << s << i << endl; return i; }
private:
   int  i;
};
```

```
int main()
{
   B  b1, b2(10), *pb;
   D  d1, d2(10), *pd = &d2;

   b1.print(); b2.print(); d1.print(); d2.print();
   b1.print("b1.i = "); b2.print("b2.i = ");
   d1.print("d1.i = "); d2.print("d2.i = ");
   pb = pd;
   pb -> print(); pb -> print("d2.i = ");
   pd -> print(); pd -> print("d2.i = ");
}
```

ANSWER TO CHAPTER 8 EXERCISE 10

Modify class `D` in exercise 9 to be

```
class D2 : private B {
public:
   B::i;            // access modification
   void  print_i()
   {
      cout << i << " inside D2 and B::i is "
           << B::i << endl;
   }
};
```

What is changed in the output from that program?

This program works under old compilers. Access modification no longer allows this to compile.

CHAPTER 9 ANSWER TO REVIEW QUESTIONS

1. What two standard output streams are provided by *iostream*? `cout` **and** `cerr`

2. What *ctype* method capitalizes alphabetic characters? `toupper()`

3. How is `EOF` tested for when using `cin`? `cin.eof()`

4. Name two manipulators and describe their purpose. The table is reproduced here:

I/O Manipulators		
Manipulator	Function	File
`endl`	Outputs newline and flush	iostream
`ends`	Outputs null in string	*iostream*
`flush`	Flushes the output	*iostream*
`dec`	Uses decimal	*iostream*
`hex`	Uses hexadecimal	*iostream*
`oct`	Uses octal	*iostream*
`ws`	Skips white space on input	*iostream*
`skipws`	Skips white space	*iostream*
`noskipws`	Does not skip white space	*iostream*
`boolalpha`	Prints true and false	*iostream*
`noboolalpha`	Prints 1 and 0	*iostream*
`fixed`	Prints using format 123.45	*iostream*

I/O Manipulators		
Manipulator	Function	File
`scientific`	Prints using format 1.2345e+02	*iostream*
`left`	Fills characters to right of value	*iostream*
`right`	Fills characters to left of value	*iostream*
`internal`	Fills characters between sign and value	*iostream*
`setw(int)`	Sets field width	*iomanip*
`setfill(int)`	Sets fill character	*iomanip*
`setbase(int)`	Sets base format	*iomanip*
`setprecision(int)`	Sets floating-point precision	*iomanip*
`setiosflags(long)`	Sets format bits	*iomanip*
`resetiosflags(long)`	Resets format bits	*iomanip*

5. What method can be used to read strings from a file? **Use `cin.read()` or if the input is redirected from a file, `cin.get()` and `cin.getline()`.**

6. The class `stringstream` allows strings to be treated as `iostream`s, and the ***sstream*** library must be included.

7. In OOP, objects should know how to print themselves, and it is best to do this by **overloading `<<`** for user-defined ADTs.

8. Synchronization problems can occur when using stdio and iostream in the same program because the two libraries use different buffering strategies, which can be avoided by calling `ios::sync_with_stdio()`.

9. Fill in the C++ stream names and their default connection devices. The table **is reproduced here.**

Standard Files			
C	C++	Name	Connected To
stdin	cin	Standard input file	Keyboard
stdout	cout	Standard output file	Screen
stderr	cerr	Standard error file	Screen
stdprn	cprn	Standard printer file	Printer
stdaux	caux	Standard auxiliary file	Auxiliary port

10. File I/O is handled by including `fstream`, which contains the classes `ofstream` and `ifstream` for output and input file-stream creation and manipulation.

Solutions for Exercises in Chapter 9

Answer to Chapter 9 Exercise 4

Write a program that prints 1,000 random numbers to a file.

In file ch09ex04.cpp

```
#include <fstream>
#include <cstdlib>
using namespace std;

int main(int argc, char** argv)
{
   if (argc != 2) {
      cout << "\nUsage: " << argv[0]
           << " outfile" << endl;
      exit(1);
   }
   ofstream  f_out(argv[1]);

   if (!f_out) {
      cerr << "cannot open " << argv[2] << endl;
      exit(1);
   }
   for (int i = 1; i <= 1000; ++i) {
      f_out << i;
      f_out.put(' ');
   }
 }
```

Answer to Chapter 9 Exercise 7

Read a text file and write it to a target text file, changing all lowercase to uppercase and double spacing the output text.

```
void double_spaceUpper(ifstream& f, ofstream& t)
{
   char  c;

   while (f.get(c)) {
      t.put(toupper(c));
      if (c == '\n')
         t.put(c);
   }
}

int main(int argc, char** argv)
{
   if (argc != 3) {
      cout << "\nUsage: " << argv[0]
           << "  infile  outfile" << endl;
      exit(1);
   }
   ifstream  f_in(argv[1]);
   ofstream  f_out(argv[2]);

   if (!f_in) {
      cerr << "cannot open " << argv[1] << endl;
      exit(1);
   }
   if (!f_out) {
      cerr << "cannot open " << argv[2] << endl;
      exit(1);
   }
   double_spaceUpper(f_in, f_out);
}
```

ANSWER TO CHAPTER 9 EXERCISE 10

Write a program that reads a text file and computes the relative frequency of each of the letters of the alphabet. You can use an array of length 26 to store the number of occurrences of each letter. You can use `tolower()` to convert uppercase letters. Subtracting `'a'` then gives you a value in the range 0 to 25, inclusive, which you can use to index into the array of counts.

```
#include <fstream>
#include <cstdlib>
using namespace std;
```

```
void getDistribution(ifstream& f)
{
   char  c;
   long charCounts[26] = {0, 0}; // array for counts on letters
   long nonAlpha = 0;            // count nonalpha chars
   long totalAlpha = 0;

   while (f.get(c)) {
      if (isalpha(c)) {
         totalAlpha++;
         charCounts[tolower(c) - 'a']++;
      }
      else
         nonAlpha++;
   }

   for (int i = 0; i < 26; ++i) {
      cout << static_cast<char>('A' + i) << " = "
         << (static_cast<double>(charCounts[i])/totalAlpha) * 100
         << "%   ";
      if ((i + 1) % 5 == 0)
         cout << endl;
   }
   cout << "\n\nNon-alpha = "
      << (static_cast<double>(nonAlpha)/(totalAlpha+nonAlpha)) * 100
      << "%" << endl;
}

int main(int argc, char** argv)
{
   if (argc != 2) {
      cout << "\nUsage: " << argv[0]
           << "  infile" << endl;
      exit(1);
   }
   ifstream  f_in(argv[1]);


   if (!f_in) {
      cerr << "cannot open " << argv[1] << endl;
      exit(1);
   }

   getDistribution(f_in);
}
```

CHAPTER 10 ANSWER TO REVIEW QUESTIONS

1. True or false: In C++, `new` cannot throw an exception. **False, new can throw the `bad_alloc` exception.**

2. System exceptions, such as `SIGFPE`, are defined in **signal**.

3. The context for handling an exception is a `try` block.

4. The system-provided handler `unexpected()` is called when a function throws an exception that was not in its exception-specification list.

5. A standard exception class is `exception` and is used **for deriving further classes. Two derived classes are `logic_error` and `runtime_error`. `Logic-error` types include `bad_cast`, `out_of_range`, and `bad_typeid`, which are intended to be thrown, as indicated by their names. The `runtime_error` types include `range_error`, `overflow_error`, and `bad_alloc`.**

6. The system-provided handler `terminate()` is called when no other handler has been provided to deal with an exception.

7. Handlers are declared at the end of a `try` block, using the keyword `catch`.

8. The **type list** is the list of types a throw expression can have.

9. Name three standard exceptions provided by C++ compilers and libraries. `bad_alloc`, `bad_cast`, `out_of_range`, `bad_typeid`, `range_error`, `overflow_error`.

10. What two actions should most handlers perform? **Issue an error message and abort.**

SOLUTIONS FOR EXERCISES IN CHAPTER 10

ANSWER TO CHAPTER 10 EXERCISE 1

The following bubble sort does not work correctly. Place assertions in this code to test that it is working properly. Besides detecting errors, the placing of assertions in code as a discipline aids you in writing a correct program. Correct the program.

```
// Incorrect bubble sort

void swapIt(int a, int b)
{
   int  temp = a;

   a = b;
   b = temp;
}

void bubble(int a[], int size)
{
   int  i, j;

   for (i = 0; i != size - 1; ++i)
      for (j = i ; j != size - 1; ++j)
         if (a[j] < a [j + 1])
            swapIt (a[j], a[j + 1]);
}

int main()
{
   int  t[10] = { 9, 4, 6, 4, 5, 9, -3, 1, 0, 12};

   bubble(t, 10);
   for (int i = 0; i < 10; ++i)
      cout << t[i] << '\t';
   cout << "\nsorted?" << endl;
}
```

In file ch10ex01.cpp

```
void bubble(int a[], int size)
{
   int  i, j;
   for (i = 0; i != size - 1; ++i)
      for (j = i; j != size - 1; ++j) {
         if (a[j] < a [j + 1])
            swapIt (a[j], a[j + 1]);
         assert(a[j] >= a[j+1]);
         // this fails on unordered data
      }
}

int main()
{
   int  t[10] = { 9, 4, 6, 4, 5, 9, -3, 1, 0, 12};

   bubble(t, 10);
   for (int i = 0; i < 10; ++i) {
      cout << t[i] << '\t';
      assert(i < 9 || t[i] >= t[i+1]);
      // this fails on unordered data
   }
   cout << "\nsorted? " << endl;
}
```

ANSWER TO CHAPTER 10 EXERCISE 4

Write a program that asks the user to enter a positive integer. Have it throw an exception when the user fails to enter a correct value. Have the handler write out the incorrect value and abort.

In file ch10ex04.cpp

```
/*****************************************************************

   Write a program that asks the user to enter a positive integer.
   Have it throw an exception when the user fails to enter a correct
   value.
   Have the handler write out the incorrect value and abort.

*****************************************************************/

#include <iostream>
#include <stdlib>
#include <string>
using namespace std;


int main()
{
   try {
      while(true){
         int data;
         cout << " Enter a positive integer: " << endl;
         cin >> data;
         if (data < 1)
            throw data;
         cout << " Data was correct and could do more " << endl;
      }
   }
   catch(int n)
      { cerr << "exception caught\n " << n << endl; abort(); }

}
```

ANSWER TO CHAPTER 10 EXERCISE 5

Rewrite the previous program to require the handler to ask the user for a correct value. The program should terminate printing the correct value. Many programs try to ensure that input is failure proof. This is an aspect of good software engineering.

In file ch11ex05.cpp

```
#include <iostream>
#include <stdlib>
#include <string>
using namespace std;

int main()
{

   int data;
   over:try {
      cout << " Enter a positive integer: " << endl;
      cin >> data;
      if (data < 1)
         throw data;
      }
      catch(int n)
      {
         cerr << "must be greater than 0\n " << n << endl;
         goto over;
      }
      cout << " Data was correct and could do more " << endl;
}
```

This is not the previous exercise rewritten because the previous exercise had an infinite loop that was terminated by the exception.

CHAPTER 11 ANSWER TO REVIEW QUESTIONS

1. Name three typical characteristics of an object-oriented programming language. **Encapsulation with data hiding, type extensibility, inheritance, polymorphism with dynamic binding.**

2. True or false: Conventional academic wisdom is that excessive concern with efficiency is detrimental to good coding practices. **True.**

3. Through **inheritance**, a hierarchy of related ADTs can be created that share code and a common interface.

4. Name three properties of a black box for the client. **Simple to use, easy to understand, familiar, in a component relationship within the system, cheap, efficient, and powerful.**

5. Name three properties of a black box for the manufacturer. **Easy to reuse and modify, difficult to misuse and reproduce, profitable to produce with a large client base, cheap, efficient, and powerful.**

6. **Structured programming** methodology has a process-centered view and relies on stepwise refinement to nest routines but does not adequately appreciate the need for a corresponding view of data.

7. **Polymorphism** is the genie in OOP, taking instruction from a client and properly interpreting its wishes.

8. Give an example of ad hoc polymorphism. **Coersion, overloading.**

9. Describe at least two separate concepts for the keyword `virtual` as used in C++. Does this cause conceptual confusion? Virtual functions, virtual inheritance, pure virtual functions, virtual destructors. **The keyword virtual provides overriding. It also is used in multiple inheritance to avoid ambiguities.**

SOLUTIONS FOR EXERCISES IN CHAPTER 11

ANSWER TO CHAPTER 11 EXERCISE 1

Consider the following three ways to provide a Boolean type:

```
// Traditional C using the preprocessor

#define  TRUE  1
#define  FALSE 0
#define  Boolean int

// ANSI C and C++ using enumerated types
// Prior to adoption of the native bool type

enum Boolean { false, true };

// C++ as a class

class Boolean {
   .....
public:
   // various member functions
   // including overloading  !  &&  ||  == !=
};
```

What would be the advantages and disadvantages of each style? Keep in mind scope, naming and conversion problems. In what ways is it desirable for C++ to now have a native type `bool`?

Discuss the advantages and disadvantages of each style. Keep in mind scope, naming, and conversion problems. In what ways is it desirable for C++ to now have a native type `bool`?

ANSWER: The use of defines means the preprocessor controls the scope. This is an error prone technique. It is very efficient and consistent with how C originally provided true and false. The use of the `enum` creates a proper type. It is efficient and straightforward. It sufferes from the defect of all these ad hoc techniques of not being universally used.
The use of class is the most costly, but most flexible of all these approaches. Having the native type bool gives a properly scoped type that is universal.

Answer to Chapter 11 Exercise 2

C++ originally allowed the `this` pointer to be modifiable. One use was to have user-controlled storage management by assigning directly to the `this` pointer. The assignment of 0 meant that the associated memory could be returned to free store. Discuss why this is a bad idea. Write a program with an assignment of `this = 0`. What error message does your compiler give you? Can you get around this with a cast? Would this be a good idea?

ANSWER: A very error prone technique. This lead to many pointer leaks and very hard to maintain obscure code.

In file ch11ex02.cpp

```
//Assigning to this
#include <iostream>

class X {
public:
   X(int n){p = new char[n]; len = n; top = 0;}
   void useThis(){ reinterpret_cast<void*>(this) = 0;}
   void push(char c){ p[top++] = c; }
   char pop(){ return p[--top];}

private:
   char* p;
   int len;
   int top;
};


int main()
{
   X y(20), z(30);
   y.push('A'); y.push('B');
   cout << y.pop() << endl;
   y.useThis();
      cout << y.pop() << endl;
}
```

It is possible to cast this pointer so that it can be assigned to. Otherwise the code will not compile as the this pointer is const.

Answer to Chapter 11 Exercise 8

List three things that you would drop from the C++ language. Argue why each would not be missed. For example, it is possible to have protected inheritance, although it was never discussed in this text? Should it be in the language for completeness' sake? Can you write code that uses protected inheritance that demonstrates that it is a critical feature of language, as opposed to an extravagance?

Answer:

register declaration - unused and is largely better left for compiler to optimize.

static file scope declarations - unneeded now that C++ have namespaces.

protected inheritance - I have never used it or seen it used.

multiple inheritance - controversial and mostly misused. even when used correctly can be very inefficient.