Just-in-time C++

An introduction to programming

using five puzzle projects

developed incrementally

by

Vince Huston

Table of Contents

Introduction

Why does the world need yet another C++ book? This book is peculiar because:

- This is a workbook – and you can never have too many problem sets or projects on which to practice.
- The projects are logic games and puzzles – not as flashy as a modern video game, but engaging and stimulating none the less.
- C++ is presented only as needed – "just in time". The idea is for the reader to "pull" more understanding of C++, instead of the author trying to "push" each new feature.

"Good judgment is the result of experience. Experience is the result of bad judgment." [Mark Twain]

This workbook focuses on practicing judgment: what are the features that are necessary and sufficient for each project, what stepwise increments would result in demonstrable progress and maintain a "working" system, what kind of leverage could be achieved by investing in data structures and algorithms, and what constitutes a "good enough" design. The C++ language is merely a "means". The "end" is practicing and appropriating insight.

Who is this book for?

- For beginners who want a crash course in programming, or have been disaffected by traditional introductory courses.
- For students who have completed the first programming course and are interested in additional "problem sets".
- For instructors who would like additional projects to integrate into their courses.
- For hackers-to-be looking for challenges on which to practice.

Paul Graham in his "Hackers and Painters" paper [http://www.paulgraham.com/hp.html] suggests that Computer Science departments are populated by four distinct personalities: mathematicians, scientists, engineers, and hackers. "Good software designers are no more engineers than architects are … What hackers, painters, composers, architects, and writers all have in common is that they're makers. What hackers and painters are trying to do is make good things … You learn to paint mostly by doing it. Ditto for hacking. Most hackers don't learn to hack by taking college courses in programming. They learn to hack by writing programs of their own at age thirteen."

A dominant motivation for this book is to facilitate the assertion: you learn to program by programming.

The core values of this book are:

- **Demo**

  Example is the school of mankind, and they will learn at no other. [Edmund Burke]

- **Do**

  I hear and I forget. I see and I remember. I do and I understand. [Chinese proverb]

- **Distill**

  Doing that well with one dollar which any bungler can do with two. [Arthur Wellington]

Demo. All material and discussion is presented almost exclusively through examples. I would rather paint a picture in code, than bury insight in a thousand words.

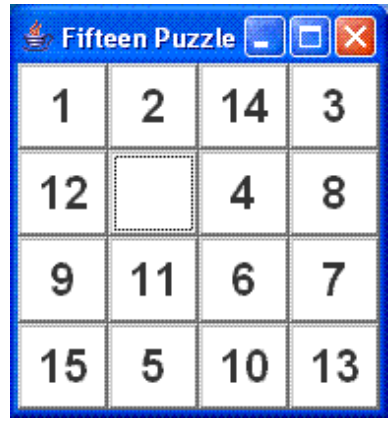
Do. Understanding does not come – or does not stick around – until the learner has had to go "hand-to-hand" with the material over and over.

Distill. The projects in this book are compelling and approachable, but they are not simple. Much of each project will involve serious thought about data structures and algorithms. These subjects are about abstraction, about leverage, about standing firm against the temptation to "let's just brute force this implementation", and insisting that simplicity and elegance are worth the up-front investment. I am reminded of a passage from C. S. Lewis' The Lion, the Witch and the Wardrobe. The visitor asks if Aslan the Lion is safe. The response is, "Of course he isn't safe. But he's good." Emphasizing more advanced material like data structures and algorithms in an introductory C++ book is not safe – but it **is** good.

Whenever this icon appears, it is used to highlight a programming pearl – a rule of thumb for design or problem solving.

The five logic games and puzzles are introduced here. Each can be implemented with a graphical user interface. But a console/keyboard user interface is much simpler, sufficiently interesting, and the target for this book. Additionally, console/keyboard user interfaces can be implemented in any language. This book will focus on C++, but the reader can use their target language instead.

Fifteen Puzzle

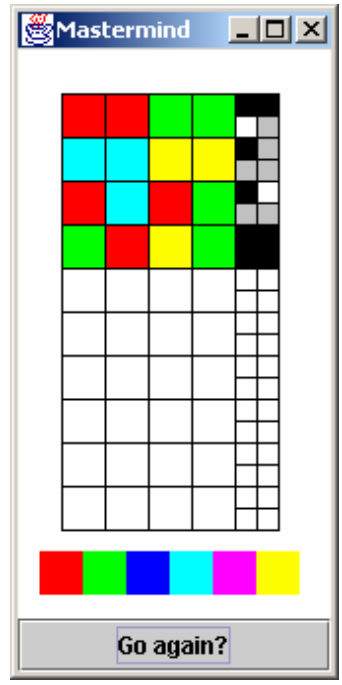


The puzzle consists of a 4 by 4 grid containing 15 slideable squares and one blank space. The puzzle is initialized by randomly sliding the squares until they are sufficiently "shuffled". The player then moves individual squares, or entire rows/columns until the pieces are back in ascending order.

Below, the "board" has been randomized, and the player inputs the digit '6'. The puzzle finds the '6', and moves it and the "10" toward the blank space. Next, the player asks for the '1' (and the intervening '2' and "13") to be moved toward the blank space.

```
     6  4 14
  2  1  7  8
  9  3 13 12
 11  5 10 15

Number: 6

  6     4 14
  2  1  7  8
  9  3 13 12
 11  5 10 15

Number: 1

  6  1  4 14
  2     7  8
  9  3 13 12
 11  5 10 15

Number: 2

  6  1  4 14
     2  7  8
  9  3 13 12
 11  5 10 15
```

Mastermind

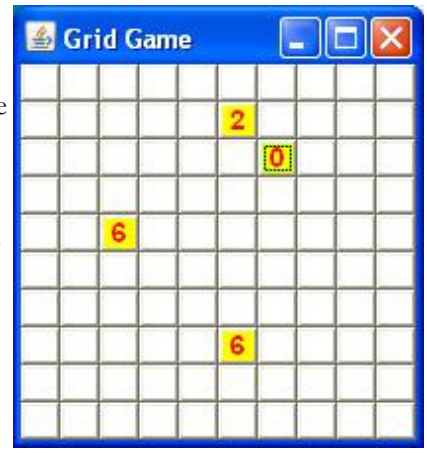


In Mastermind, player one selects a sequence of four colored pegs, and player two makes guesses until she has identified the sequence. There are six colors, and each color may be used more than once. As guesses are made, the feedback provided is: a black peg for each correct color in the correct position, and a white peg for each correct color in the wrong position.

A "command line" user interface for Mastermind is almost as enjoyable as the graphical UI. And – implementing a "command line" UI is very straight-forward. The user is prompted for 4 integers, and then receives feedback in the form of 2 integers: the number of black pegs, and the number of white pegs.

```
Input: 1 1 2 2
                  2  1
Input: 4 4 6 6
                  1  0
Input: 1 4 1 2
                  1  1
Input: 2 1 6 2
                  4  0
```

Grid Game



I created this game as a Java GUI lab. Click on a grid location and the "distance" to the target is displayed. "Distance" is defined as "delta X plus delta Y" (the number of horizontal and vertical steps from the guess to the target). In the example to the right, I clicked on the cell at row 5 and column 3. My second selection was 3 rows down and 3 columns right. My third selection was 6 rows up. You will notice the "2" cell is 6 steps away from both "6" cells. The third selection was off by 2 steps. And the fourth selection "hit" the target.

A console/keyboard user interface (that is the target in this book) appears below. The "board" is drawn, and the player specified row 3 and column 6. The board is redrawn with a "score" of 7 at that location. The player then specified row 5 and column 1, and the score returned is 2.

```
   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -  -  -
2  -  -  -  -  -  -  -  -  -  -
3  -  -  -  -  -  -  -  -  -  -
4  -  -  -  -  -  -  -  -  -  -
5  -  -  -  -  -  -  -  -  -  -
6  -  -  -  -  -  -  -  -  -  -
7  -  -  -  -  -  -  -  -  -  -
8  -  -  -  -  -  -  -  -  -  -
9  -  -  -  -  -  -  -  -  -  -

row col ... 3 6

   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -  -  -
2  -  -  -  -  -  -  -  -  -  -
3  -  -  -  -  -  -  7  -  -  -
4  -  -  -  -  -  -  -  -  -  -
5  -  -  -  -  -  -  -  -  -  -
...

row col ... 5 1

   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -  -  -
2  -  -  -  -  -  -  -  -  -  -
3  -  -  -  -  -  -  7  -  -  -
4  -  -  -  -  -  -  -  -  -  -
5  -  2  -  -  -  -  -  -  -  -
...
```

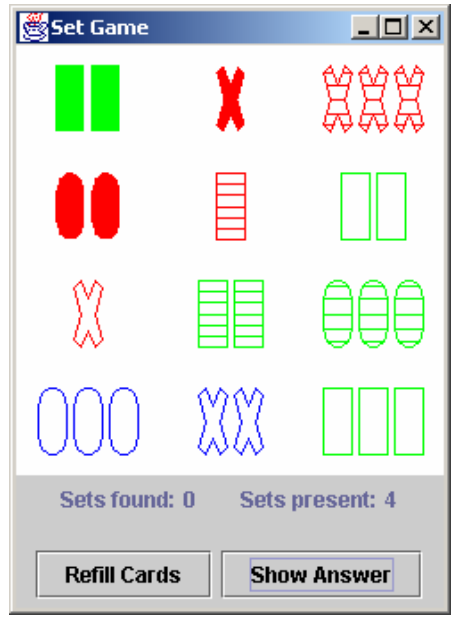
Peg Game

The Cracker Barrel Peg Game consists of fifteen holes arranged in a triangle, and fourteen pegs. The object is to remove pegs by jumping until one peg remains. The player chooses where the empty hole is positioned before play is begun. A graphical user interface might look like the windows shots to the right. The player chooses the 6 peg, and jumps over the 3 peg.

A possible command line user interface appears below. The player chooses the F peg, and jumps over the C peg.

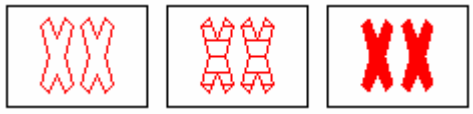
```
                    a

               <B>   <C>

          <D>   <E>   <F>

     <G>   <H>   <I>   <J>

<K>   <L>   <M>   <N>   <O>

Move (from to): f a

               <A>

               <B>    c

          <D>   <E>    f

     <G>   <H>   <I>   <J>

<K>   <L>   <M>   <N>   <O>
```

Set Game



The game is played by finding sets of three cards that are all the same, or all different, across four independent "facets" or "dimensions". These dimensions are: number, shape, color, and fill. Each of these dimensions has three possible values. Number may be: one, two, or three; color may be: red, green, or blue; shape may be: rectangle, X, or O; and fill may be: open, striped, or solid. Because the book is blank-and-white, color will be ignored.

The following three cards form a set because: they are the same in shape (X), they are the same in number (2), and they are different in fill (one open, one striped, one solid).



The following three cards form a set because they are: the same in shape (X), different in number (one 1, one 2, one 3), and different in fill (one open, one striped, one solid).

The following three cards form a set because they are: different in shape (one O, one X, one rectangle), different in number (one 1, one 2, one 3), and different in fill (one open, one striped, one solid).

The following three cards are **not** a set because there are two "open" fills and one "striped" fill. If the "striped" fill were "open" (so that the fills are all the same) then these would be a set. Or – if one of the "open" fills were "solid" (so that the fills are all different) then the set criteria is satisfied.

A possible command line user interface appears below. The game is played by dealing twelve cards. As sets are picked up, those cards are replaced as the dealer has a chance. "Number" and "shape" are portrayed directly, "fill" is represented with special characters, and color has been eliminated. The cards are numbered 1 to 12 from left to right, and top to bottom. The player "selected" three cards by entering "1 8 12". The game evaluated that those cards do in fact constitute a set, removed them,

and incremented the "sets found" count. Before asking for the missing cards to be replaced, the player input "2 5 10", and the game evaluated, removed, and added the set.

```
. SSS    ^ XX     ^ XXX    + XXX
. O      + SS     + OO     . X
^ O      + SSS    + XX     . OO

sets present 8, sets found 0
Input: 1 8 12

         ^ XX     ^ XXX    + XXX
. O      + SS     + OO
^ O      + SSS    + XX

sets present 3, sets found 1
Input: 2 5 10

                  ^ XXX    + XXX
         + SS     + OO
^ O               + XX

sets present 1, sets found 2
Input: 6 7 9

                  ^ XXX    + XXX

                  + XX

sets present 0, sets found 3
Input:
```

Brute Force versus Leverage

The following two examples demonstrate a fetish of this book: refusing to accept "brute force", and searching for elegance. The complexity of these implementations are entirely pre-mature – but I invite the reader to consider the "spirit of the law" and not get wrapped around the axle by the "letter of the law". These two alternative implementations first appeared in The Elements of Programming Style by Kernighan and Plauger in 1974. The goal of each is: convert a julian date (e.g. 2007-032) to a more traditional date (e.g. 2/1/07). The first implementation settles for an all-too-typical brute force approach: test if the julian number belongs in January, else test if the julian number belongs in February, else test is the julian number belongs in March, else test if the julian number belongs in April, etc.

```
string convert_julian( int year, int julian ) {
  int leap = 0, month, day;
  if (year % 400 == 0)
    leap = 1;
  else if (year % 100 == 0)
    ;
  else if (year % 4 == 0)
    leap = 1;
  if (julian <= 31) {
    month = 1;
    day = julian;
  } else if (julian <= (59+leap)) {
    month = 2;
    day = julian - 31;
  } else if (julian <= (90+leap)) {
    month = 3;
    day = julian - (59+leap);
  } else if (julian <= (120+leap)) {
    month = 4;
    day = julian - (90+leap);
  } else if (julian <= (151+leap)) {
    month = 5;
    day = julian - (120+leap);
  } else if (julian <= (181+leap)) {
    month = 6;
    day = julian - (151+leap);
  } else if (julian <= (212+leap)) {
    month = 7;
    day = julian - (181+leap);
  } else if (julian <= (243+leap)) {
    month = 8;
    day = julian - (212+leap);
  } else if (julian <= (273+leap)) {
    month = 9;
    day = julian - (243+leap);
  } else if (julian <= (304+leap)) {
    month = 10;
    day = julian - (273+leap);
  } else if (julian <= (334+leap)) {
    month = 11;
```

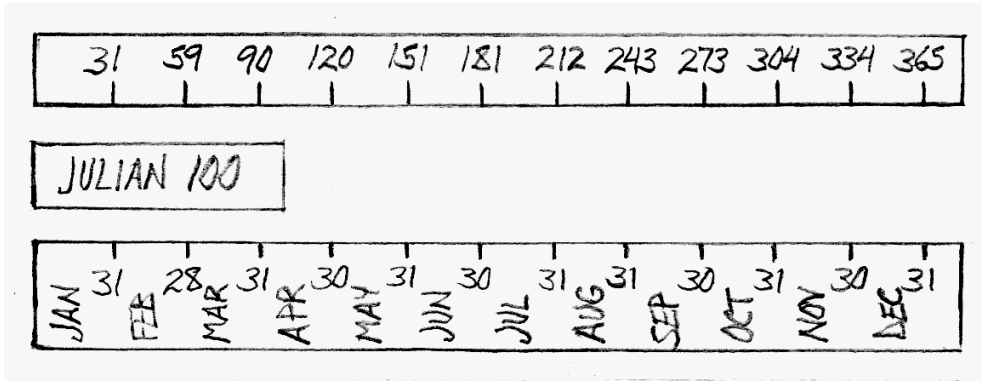
```
    day = julian - (304+leap);
  } else {
    month = 12;
    day = julian - (334+leap);
  }
  int yr = year - (year/100 * 100);
  stringstream ss;
  ss << setfill('0') << setw(2) << month << '/' << setw(2) << day
     << '/' << setw(2) << yr;
  return ss.str();
}

int main( void ) {
   cout << convert_julian( 1900,  60 ) << '\n';   // 03/01/00
   cout << convert_julian( 2000,  60 ) << '\n';   // 02/29/00
   cout << convert_julian( 2002,  60 ) << '\n';   // 03/01/02
   cout << convert_julian( 2004,  60 ) << '\n';   // 02/29/04
   cout << convert_julian( 1900, 365 ) << '\n';   // 12/31/00
   cout << convert_julian( 2000, 365 ) << '\n';   // 12/30/00
   cout << convert_julian( 2002, 365 ) << '\n';   // 12/31/02
   cout << convert_julian( 2004, 365 ) << '\n';   // 12/30/04
}
```

The second implementation is a fraction of the size of the first. The 12 decisions laboriously enumerated above have been rolled into a single table below (i.e. days_per_month_table), and a loop that navigates the table.

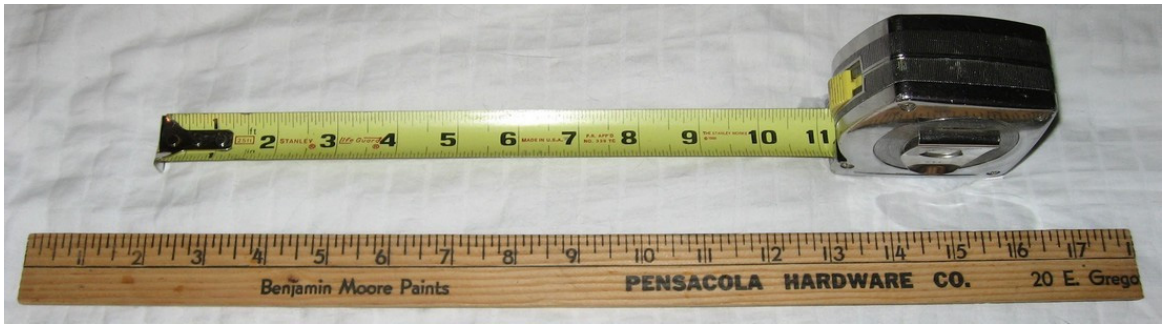
```
string convert_julian( int year, int julian ) {
  int days_per_month_table[2][12] = {
                       {31,28,31,30,31,30,31,31,30,31,30,31},
                       {31,29,31,30,31,30,31,31,30,31,30,31} };
  int leap = (year%400 == 0 || year%4 == 0 && year % 100 != 0) ? 1 : 0;
  int i;
  for (i=0; julian > days_per_month_table[leap][i]; i++)
    julian -= days_per_month_table[leap][i];
  int yr = year - (year/100 * 100);
  stringstream ss;
  ss << setfill('0') << setw(2) << i+1 << '/' << setw(2) << julian
     << '/' << setw(2) << yr;
  return ss.str();
}
```

The `days_per_month_table` is an example of a data structure – a framework of data designed to offer peculiar leverage for a class of problems. The subsequent `for` loop is an example of an algorithm – a set of instructions that produce a problem solution. The first julian conversion implementation is also an example of an algorithm – albeit an unfortunately clumsy example.

Both implementations could be viewed as two different types of yardstick for measuring julian date. The first has computed a "running total" for each month, but the second uses raw days per month.



The first yardstick is easier to use directly, and the second yardstick looks like more trouble than its worth. Perhaps a better analogy is the difference between a yardstick and a tape measure.



A yardstick is fixed and unwieldy. A tape measure is arguably agile.

Using a Compiler

Any C++ compiler will work. Dev-C++ (available at www.bloodshed.net) is an excellent choice and is free. It has a graphical user interface, but I prefer to use its command line user interface. I installed it at \cpp. Compiling and linking is as simple as the first line below, which produces the file "a.exe".

```
C:> \cpp\bin\c++ hello.cpp

C:> a.exe
hello world

C:>
```

Chapter 1 ... Output, Input, Intro

Output. Consider the Grid Game project. The first task is to print/output the board.

```
   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -  -  -
2  -  -  -  -  -  -  -  -  -  -
3  -  -  -  -  -  -  -  -  -  -
4  -  -  -  -  -  -  -  -  -  -
5  -  -  -  -  -  -  -  -  -  -
6  -  -  -  -  -  -  -  -  -  -
7  -  -  -  -  -  -  -  -  -  -
8  -  -  -  -  -  -  -  -  -  -
9  -  -  -  -  -  -  -  -  -  -
```

The command for sending text to the screen is `cout`. The token `cout` can be thought of as a proxy for the computer's monitor. The "<<" token points in the direction of data movement. Text that should be treated as data is enclosed in double quotes, and has historically been referred to as a "string of characters".

```
cout << "here is output from the program";
```

Here is the minimal amount of typing needed to produce a working program.

```
#include <iostream>
using std::cout;

int main() {
   cout << "here is output from the program";
}
```

`cout` allows output to be built up in pieces by "cascading" multiple "<<" operators. Statements are always terminated with a semi-colon.

```
cout << "here is " << "output from" << " the program";
```

Output can also be built up with separate statements.

```
cout << "here is ";
cout << "output from" ;
cout << " the program";
```

A comment is the language feature that allows descriptive text or documentation to be inserted in a program. All text that follows the characters "//" is a comment.

```
// This is a comment in C++
// It will be ignored by the compiler

cout << "here is ";      // all the text after "//" characters
cout << "output from";   // and before a new line is a comment
```

In this book, output produced by a code segment will routinely be appended as a comment.

```
cout << "here is ";
cout << "output from" ;
```

```
cout << " the program"

// here is output from the program
```

To tell the compiler that a "carriage return" is desired, a special code is necessary. The backslash-n pair requests a new line (or carriage return).

```
cout << "line 1\nline followed by a blank line\n\nlast line\n";

// line 1
// line followed by a blank line
//
// last line
```

A sequence of characters intended to be treated as a unit is enclosed in double quotes. When a character "stands alone", it is enclosed in single quotes.

```
cout << "a string of characters\n";
cout << "stand-alone characters" << ':' << ' ' << '1' << 'z' << '\n';

// a string of characters
// stand-alone characters: 1z
```

So … to print the Grid Game board, you could use the following brute force approach. But you would very quickly run out of leverage to implement the remainder of the game.

```
cout << "   0  1  2  3  4  5  6  7  8  9\n";
cout << "0  -  -  -  -  -  -  -  -  -  -\n";
cout << "1  -  -  -  -  -  -  -  -  -  -\n";
...
```

Practice. Complete increment 1 of Appendix B and Appendix D.

Input. Receiving input from the keyboard is as easy as sending output to the monitor. Input received from the keyboard has to have a place to go – like mail received from the postal service. Each mailbox is a repository – a location where "puts" and "gets" can rendezvous. A mailbox represents an address – a unique identifier that distinguishes it from every other mailbox.

The mechanism needed for receiving input is called a "variable". Like the mailbox, each variable has a unique identifier – its name. Unlike the mailbox, each variable has a "type" – it can hold a certain type of data (e.g. a whole number, or a real number, or a single character, or a string of characters). Since the variable is an invention of the programmer, it must be "declared" to the compiler before it can be put in use.

A declaration follows. The type of the variable goes first, followed by its name. The next section will introduce several types supported by C++. For now, know that the type `string` can hold any combination of letters, digits, and punctuation. The name below is "word_var". There are quite a few rules for how to form legal variable names; for our purposes – any combination of upper and lower case letters, digits, and the '_' character are legal (and sufficient).

```
// variable declaration
string  word_var;
```

The token `cin` can be thought of as a proxy for the computer's keyboard. The ">>" token points in the direction of data movement. The statements at [1] are needed for the compiler to understand the tokens `cout`, `cin`, and `string`. The `#include` and `using` statements routinely appear together when you want to reference a "resource" that is part of C++'s standard library (which is distinct from the C++ standard language).

The variable declaration for the anticipated input is at [2], and the actual input occurs at [3]. The gray background at [4] represents input from the user.

```
#include <iostream>                    [1]
using std::cout;                       [1]
using std::cin;                        [1]
#include <string>                      [1]
using std::string;                     [1]

int main() {
   string word_var;                    [2]
   cout << "Enter a word: ";
   cin >> word_var;                    [3]
   cout << "The word is: " << word_var << '\n';
}
```

```
// Enter a word: hello                [4]
// The word is: hello
```

`cin` can load a single variable [5], or multiple variables, in each statement. At [6], the user types three words, and presses <Enter>. `cin` parses the input searching for "white space" (spaces, tabs, or carriage returns). Each "token" or "word" found is assigned to the next variable specified by the programmer.

```
string first;
string second;
string third;
cout << "Enter 3 words: ";
cin >> first;                          [5]
cin >> second;                         [5]
cin >> third;                          [5]
cout << "The words are: ";             [9]
cout << first << ", ";
cout << second << ", ";
cout << third << '\n';

// Enter 3 words: one two three        [6]
// The words are: one, two, three
```

Multiple variables can be declared in a single statement by separating the variable names with commas [7]. Multiple variables can be loaded in a single `cin` statement [8]. There is no difference in functionality between [5] above and [8] below. This is true for `cout` as well (see [9] above and [10] below).

```
string first, second, third;           [7]
cout << "Enter 3 words: ";
cin >> first >> second >> third;       [8]
cout << "The words are: " << first << ", " << second     [10]
                         << ", " << third << '\n';

// Enter 3 words: tom dick harry
// The words are: tom, dick, harry
```

Practice. Complete increment 1 of Appendix C.

Variables. Let's now focus on the notion of "variables". Two other "types" supported by C++ are `int` and `double`. A variable can be declared with no initial value [1], and subsequently assigned a value [2]. Or – it can be declared and assigned an initial value in a single statement [3]. The `int` type is comparable to whole numbers (or counting numbers). It does not support the notion of a decimal point and digits less than 1. If you assign a number with a fractional component [2], that piece of the number is truncated [4] (not rounded up or down).

Numbers of type `double` **do** support decimal points [5].

```
int whole_number;                      [1]
whole_number = 12.75;                  [2]
```

```
double real_number = 12.75;                     [3]
cout << "whole_number is " << whole_number  << '\n';
cout << "real_number  is " << real_number   << '\n';

// whole_number is 12                           [4]
// real_number  is 12.75                        [5]
```

In the "**Output**" section, the distinction between a single character and a string of characters was introduced. The former is enclosed in single quotes, and the latter is enclosed in double quotes. The C++ type that can hold a single character is `char` [6]. The type that holds a sequence of characters is `string` [7].

```
char    single = 'a';                           [6]
string  many   = "one and two";                 [7]
cout << "single is " << single << '\n';
cout << "many   is " << many   << '\n';

// single is a
// many   is one and two
```

Here are some interesting arithmetic demos. Doubling an integer is no big deal [8]. When the variable "fraction" was initialized, its value was "0.333333333..." [9]. When it was doubled, its value went to "0.666666666...". Then when `cout` formatted the value, it rounded to six significant digits [10]. [11] demonstrates that the '+' operator on string variables performs string concatenation.

Performing addition on `char` variables has no real-world significance – but it does demonstrate an underlying implementation detail. Every character is represented by a value in the range of 1 to 127. '0' is 48, '1' is 49, 'A' is 65, 'B' is 66, 'a' is 97, 'b' is 98. What's happening at [12] is the value 49 is doubled, and then interpreted as the character 'b'.

```
int    number   = 123;
double fraction = 1.0 / 3.0;                     [9]
string word     = "123";
char   letter   = '1';
number   = number + number;                      [8]
fraction = fraction + fraction;
word     = word + word;                         [11]
letter   = letter + letter;                     [12]
cout << "number   is " << number   << '\n';
cout << "fraction is " << fraction << '\n';
cout << "word     is " << word     << '\n';
cout << "letter   is " << letter   << '\n';

// number   is 246                               [8]
// fraction is 0.666667                         [10]
// word     is 123123                           [11]
// letter   is b                                [12]
```

The C++ type `bool` holds a boolean value: true or false, on or off, yes or no, 1 or 0. The reserved words "true" and "false" allow for initialization. When a `bool` variable is output, the legacy values of 1 and 0 are reported. The `bool` type will become more interesting in the "**Expressions**" section that follows.

```
bool  on_var  = true;
```

```
bool  off_var = false;
cout << "on_var  is " << on_var  << '\n';
cout << "off_var is " << off_var << '\n';

// on_var  is 1
// off_var is 0
```

The type `string` is different from the other types previously discussed – it is a "class" rather than a primitive type. The "class" feature encompasses the largest segment of the C++ language: object-oriented programming. For now, it is sufficient to understand that a "class" is like a type on steroids. Instances of a "class" (called "objects") are smart (perhaps even sentient) – they can respond to requests from the programmer.

The syntax for making a request on an object appears below. Notice the use and placement of the period and parentheses. We'll see the parentheses again in the **Functions** section.

```
object.request();
```

The next example declares two `string` objects: first and second. Each is then asked to respond to `size()` and `empty()` requests. The former reports the length of the `string` object, the latter returns a boolean value that represents whether the `string`'s length is zero. Another request to which `string` objects respond is `clear()` [13] – the result is to initialize the object.

```
string one = "0123456789";
string two;
cout << "one.size()  is "  << one.size()  << ", two.size()  is "
                           << two.size()  << '\n';
cout << "one.empty() is  " << one.empty() << ", two.empty() is "
                           << two.empty() << '\n';
one.clear();                                                        [13]
cout << "one.size()  is  " << one.size()  << ", one.empty() is "
                           << one.empty() << '\n';

// one.size()  is 10, two.size()  is 0
// one.empty() is  0, two.empty() is 1
// one.size()  is  0, one.empty() is 1
```

In addition to a name and a type, each variable has a "life time". A variable lives for the duration of its enclosing "scope". To over-simplify: scope is a block of code enclosed in curly braces. This will become important in the next section. Attempting to compile the following code, would produce the error message "redeclaration of 'int common_name'" at [14]. The compiler can not deal with the same name defined twice in the same scope.

```
int main()
{                               // beginning of scope
   int common_name = 123;
   cout << "common_name is " << common_name << '\n';
   int common_name = 234;       // ERROR                    [14]
}                               // end of scope
```

Below, the first declaration of `common_name` "goes away" at the end of its enclosing scope [15]. At that point, the name `common_name` can be reused [16].

```
{
   int common_name = 123;
```

```
   cout << "common_name is " << common_name << '\n';
}                                                                    [15]
int common_name = 234;                                               [16]
cout << "common_name is " << common_name << '\n';

// common_name is 123
// common_name is 234
```

C++ also allows "hierarchical" reuse of variable names. By opening a new level of scope at [18], the variable name declared at [17] is "put on hold", and it can be reused at [19]. When the right curly brace is encountered at [20], the second instance of `common_name` "dies", and the first instance "comes back to life" [21].

```
int common_name = 123;                                               [17]
cout << "common_name is " << common_name << '\n';
{                                                                    [18]
   int common_name = 234;                                            [19]
   cout << "common_name is " << common_name << '\n';
}                                                                    [20]
cout << "common_name is " << common_name << '\n';                    [21]

// common_name is 123
// common_name is 234
// common_name is 123                                                [21]
```

Expressions. An expression is usually an arithmetic computation like "6 * 8" and "3 – 4". Technically, it is an operator surrounded by two operands. [1] and [6] demonstrate the possibility of negative numbers. Since the two operands in [2] and [4] are integers, the operator is interpreted as integer division (which throws away the remainder, and does not round up). The 6.0 in [3] is a `double`, so the '/' operator is interpreted to mean "real world" division. The '%' in [5] is the modulo operator, but it is better understood as remainder division. It throws away the traditional quotient, and returns the remainder.

```
cout <<    6 * 8  << '\n';       // 48
cout <<   -6 * 8  << '\n';       // -48          [1]
cout <<    6 / 8  << '\n';       // 0            [2]
cout <<  6.0 / 8  << '\n';       // 0.75         [3]
cout <<   18 / 4  << '\n';       // 4            [4]
cout <<   18 % 4  << '\n';       // 2            [5]
cout <<    3 - 4  << '\n';       // -1
cout <<   -4 + 3  << '\n';       // -1           [6]
```

A boolean expression is an expression that returns a boolean value (true or false). Because of C++'s "precedence of operators", each of these boolean expressions must be enclosed in parentheses so that the compiler is not confused. Comparison operators like '>' and "<=" are self-documenting. Testing for equality [7] uses "==" and testing for inequality [8] uses "!=". [9] demonstrates the "negation" operator – it inverts whatever it is applied to.

```
cout <<   (18 < 4)   << '\n';    // 0
cout <<   (18 >= 4)  << '\n';    // 1
cout <<   (18 == 4)  << '\n';    // 0            [7]
cout <<   (18 != 4)  << '\n';    // 1            [8]
```

```
cout << ! (18 != 4)  << '\n';    // 0           [9]
```

A `bool` variable can be used to capture the value of a boolean expression. That variable can then be manipulated: at [10] it is negated and assigned back on itself.

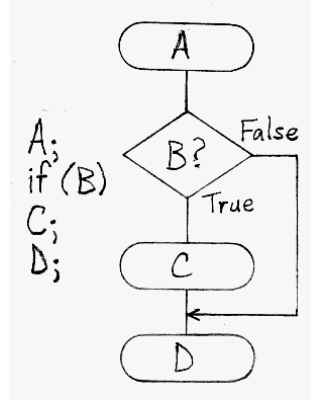
```
bool equal_to = (5 == 10);
cout << "equal_to is " << equal_to << '\n';
equal_to = ! equal_to;                          [10]
cout << "equal_to is " << equal_to << '\n';

// equal_to is 0
// equal_to is 1
```

Let's revisit integer division and remainder division, and examine some subtle functionality. Notice the output at [11]. As integer division is applied across a range of integers, a "step" function results – the quotient remains constant for the magnitude of the divisor. In similar fashion, remainder division creates a "saw tooth" function [12] – the result grows to some maximum value and then dives back to zero. The maximum value is one less that the divisor, and the period of the resulting "signal" is equal to the divisor.

```
cout << "integer division:   ";
cout << 0/3 << ' ' << 1/3 << ' ' << 2/3 << ' ' << 3/3 << ' '
     << 4/3 << ' ' << 5/3 << ' ' << 6/3 << ' ' << 7/3 << '\n';
cout << "remainder division: ";
cout << 0/3 << ' ' << 1%3 << ' ' << 2%3 << ' ' << 3%3 << ' '
     << 4%3 << ' ' << 5%3 << ' ' << 6%3 << ' ' << 7%3 << '\n';

// integer division:   0 0 0 1 1 1 2 2         [11]
// remainder division: 0 1 2 0 1 2 0 1         [12]
```

Chapter 2 ... Flow of Control

Selection. Computer languages provide a mechanism for directing flow of control based on the result of a boolean expression. "If" the boolean expression evaluates to true, then the "if" statement's "body" is executed; if the expression evaluates to false, then the "body" is skipped. In the "flow chart" to the right, B represents the boolean expression, and C represents the "body". C++ does not infer any meaning from the presence or absence of formatting. If the B expression evaluates to true, then the C statement (or block) will be executed. If B evaluates to false, then C is skipped, and execution proceeds to D.



The most basic form of an "if" statement appears below. The boolean expression goes inside parentheses, and there is no semi-colon until after the body.

```
if (boolean expression)
   body;
```

The body can be a single statement [1], or a block of statements enclosed in curly braces [2]. If a single statement is defined, the curly braces are optional [3]. Indenting the statement(s) that constitute the body of the `if` statement is a common convention for readability. Where the curly braces are positioned is a matter of personal taste. The two most popular practices are: placing the left brace immediately after the boolean expression [2], and putting the left brace on a line by itself [3].

```
// Notice the use and positioning of curly braces
int value = 5;
if (value < 5)
   cout << "value is less than 5\n";                    [1]
if (value > 5) {                                        [2]
   cout << "value is "
   cout << "greater than to 5\n";
}
if (value == 5)
{                                                       [3]
   cout << "value is equal to 5\n";
}

// value is equal to 5
```

Here is an example using a boolean expression with two variables.

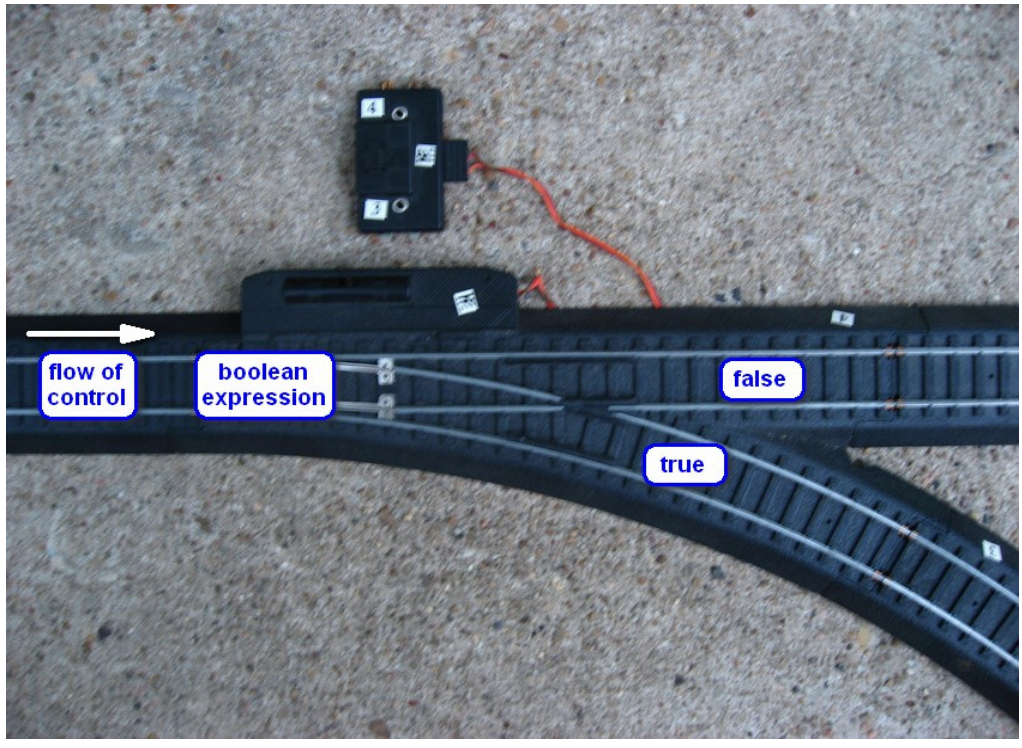
```
string first = "one";
string second = "two";
if (first == second)
{
   cout << first;
   cout << " is equal to ";
   cout << second << '\n';
```

```
}
if (first != second)  // the inequality operator
   cout << first << " is not equal to " << second << '\n';

// one is not equal to two
```

An `if` statement may be combined with an `else` statement. The `else` statement and its body will be ignored if the boolean expression returns true. The `else` statement's body will be executed if the boolean expression returns false.

```
if (boolean expression)
   body;
else
   body;
```

`if-else` is like a railroad switch. A boolean expression that returns true sends flow of control down one track, and false sends flow of control down the other track.



In the code below, notice the placement of semi-colons below: no semi-colon at the end of the `if` and `else` lines, and a semi-colon after each "body" statement.

```
int num = 42;
if (num < 42)
   cout << "num is less than 42\n";
else
   cout << "num is greater than or equal to 42\n";
```

```
// num is greater than or equal to 42
```

Compound boolean expressions join multiple simple boolean expressions with "and" and "or" operators. The compound boolean "and" uses the "`&&`" operator [4].

```
int num = 42;
if (num > 10  &&  num < 100)                              [4]
   cout << "num is between 10 and 100\n";
else
   cout << "num is outside the range of 10 to 100\n";

// num is between 10 and 100
```

The compound boolean "or" uses the "`||`" operator [5].

```
int num = 42;
if (num < 1  ||  num > 10)                                [5]
   cout << "num is NOT in the range 1..10\n";
else
   cout << "num is in the range 1..10\n";

// num is NOT in the range 1..10
```

The code below cascades if-else statements as the body of an `else` statement. The indentation demonstrates how the compiler parses the code.

```
int num = 60;
if (num < 20)
   cout << "num is less than 20\n";
else
   if (num < 40)
      cout << "num is between 20 and 39\n";
   else
      if (num < 60)
         cout << "num is between 40 and 59\n";
      else
         if (num < 80)
            cout << "num is between 60 and 79\n";
         else
            cout << "num is greater than or equal to 80\n";

// num is between 60 and 79
```

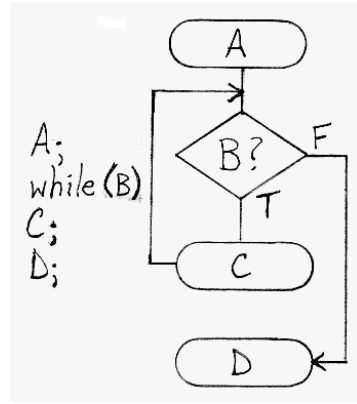
The next example is identical to the previous; the only difference is formatting. It demonstrates how to implement a multi-branch selection statement.

```
int num = 80;
if (num < 20)
   cout << "num is less than 20\n";
else if (num < 40)
   cout << "num is between 20 and 39\n";
else if (num < 60)
   cout << "num is between 40 and 59\n";
else if (num < 80)
   cout << "num is between 60 and 79\n";
else
   cout << "num is greater than or equal to 80\n";
```

```
// num is greater than or equal to 80
```

Practice. No increments are do-able until iteration is introduced.

Iteration. C++ provides several ways of repeating yourself. The most basic of these is the `while` statement. "While" the boolean expression evaluates to true, the `while` statement's "body" is executed; if the expression evaluates to false, then the "body" is skipped, and flow of control moves on. In the "flow chart" to the right, B represents the boolean expression, and C represents the "body". What has been added to the "if" statement's flow chart is the feedback arrow from the "body" back to the boolean expression.



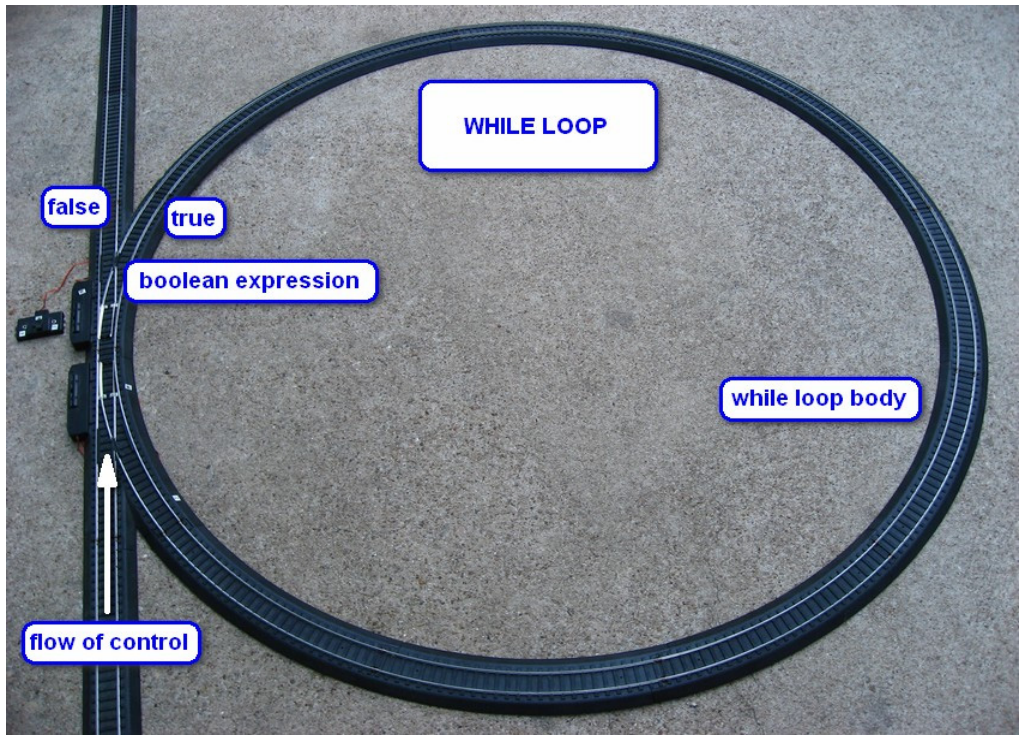
The body of the `while` statement can be a single statement, or a block of statements enclosed in curly braces [1]. Every time the body is executed, the boolean expression is immediately evaluated again. Below, notice the use of parentheses, the absence of the semi-colon after the boolean expression, and the indentation inside the curly braces. While `counter`'s value is less than 5, the block of statements in the curly braces are executed. Once the value reaches 5, the boolean expression fails, and program execution picks up after the right curly brace.

```
int counter = 0;
while (counter < 5) {                          [1]
   cout << counter << ' ';
   counter = counter + 1;
}
cout << '\n';

// 0 1 2 3 4
```

Quiz. What would happen if the curly braces above were left out?

Answer. The compiler would execute the single statement that follows the `while` statement, and immediately re-evaluate the boolean expression. Since the variable `counter` has the same value as before, the same result would ensue – again and again and again. That is known as an "infinite loop".

The `while` construct is like a railroad switch and a loop of track. The train of "flow of control" will travel around the loop of track while the boolean expression evaluates to true.



Here is a fairly useful example: converting degrees Celsius to degrees Fahrenheit. While `celsius` is less than 101, the two statements enclosed in curly braces are repeated. The second statement updates the `celsius` variable [2] right before the boolean expression is evaluated the next time around.

There is a new feature below that has not previously been discussed – `setw(num)`. For now, know that it allows the programmer to specify the "width" of the next field given to `cout`. If the next field is a number, then it will be right-justified as well.

```
int celsius = -40;
cout << "Celsius  Fahrenheit\n";
while (celsius < 101) {
   //      set width of 7        set width of 12 for the next field
   cout << setw(7) << celsius << setw(12)
                              << celsius * 9/5 + 32 << '\n';
   celsius = celsius + 20;                        [2]
}

// Celsius  Fahrenheit
//     -40         -40
//     -20          -4
//       0          32
//      20          68
//      40         104
//      60         140
//      80         176
```

```
//     100         212
```

As we saw earlier with the tokens `cout`, `cin`, and `string`, `setw()` requires the following lines at the top of the program for the compiler to accept its use.

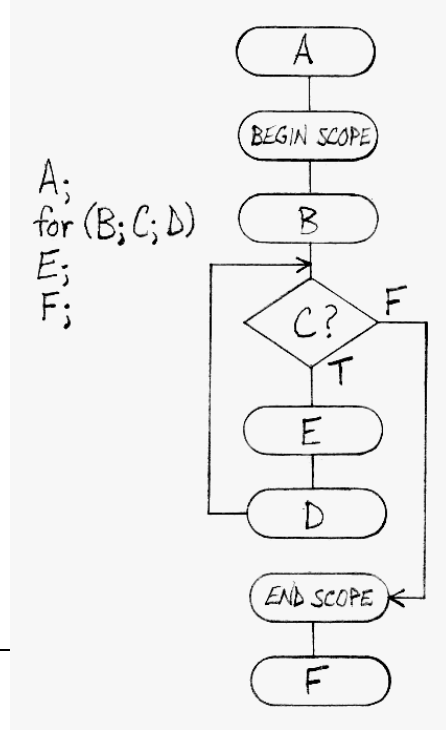
```
#include <iomanip>
using std::setw;
```

The boolean expression in the next example [3] is always true, so it seemingly results in an infinite loop. But – the `break` statement [4] will cause the enclosing loop to quit, and flow of control will immediately proceed to the next statement after the loop. Another special loop flow of control statement is `continue` [5]. It causes flow of control to jump to the next evaluation of the boolean expression.

```
int i = 0;
while (42 == 42) {                    [3]
   // If i is sufficiently large, then exit the loop
   if (i > 15)
      break;                          [4]
   // If i is even, then increment i and go to the top of the loop
   if (i % 2 == 0) {
      i = i + 1;
      continue;                       [5]
   }
   cout << i << ' ';
   i = i + 1;
}
cout << '\n';

// 1 3 5 7 9 11 13 15
```

The `for` loop provides the same functionality as the `while` loop – plus some. Inside the parentheses, there are three "clauses": initialization (B in the flow chart), evaluation (aka boolean expression, C in the flow chart), and modification (D). The initialization clause is executed only once – the first time the `for` loop is encountered. The evaluation clause is executed at the top of the loop, every time through the loop. The modification clause is executed at the bottom of the loop before program execution returns to the top of the loop.

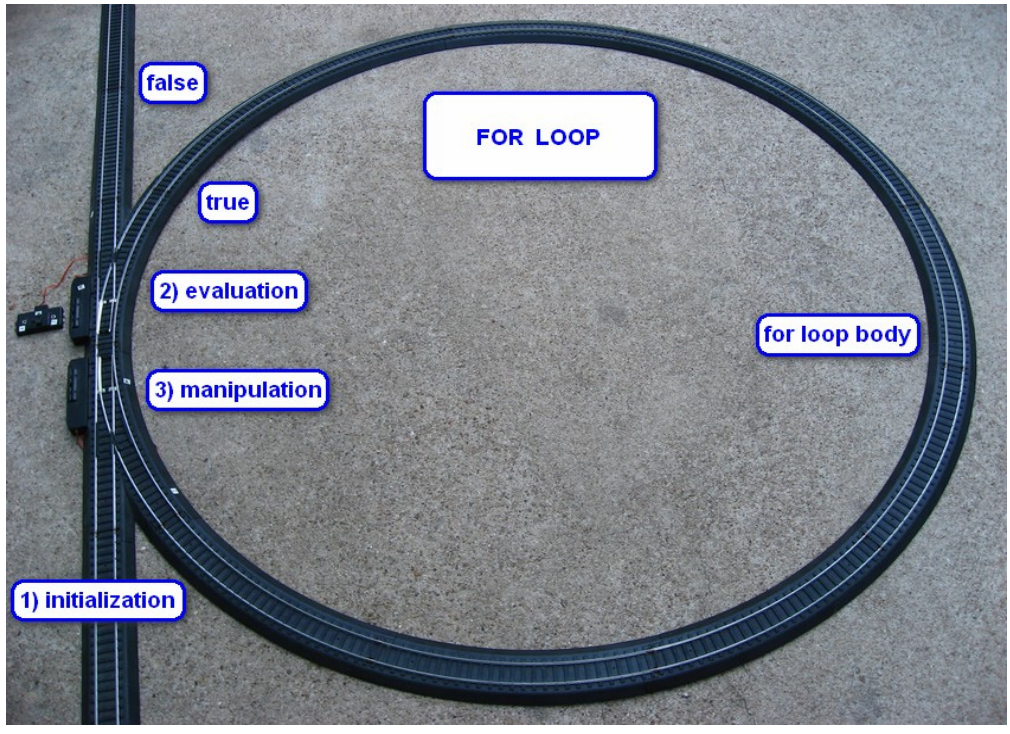
To the right, E is the `for` loop's body. B and D are new with the `for` loop. "Begin Scope" and "End Scope" seek to demonstrate that variables declared in a `for` loop have a life time that is limited to the `for` loop, and any variables declared in a `for` loop can be reused after the loop.



The example below compares comparable `for` and `while` loops.

```
for (int num = 0;      // initialization
     num < 5;          // evaluation
     num = num + 1)    // modification
{
   cout << num << ' ';
}
// int num = 0;        // initialization
// while (num < 5) {   // evaluation
//    cout << num << ' ';
//    num = num + 1;   // modification
// }
cout << '\n';

// 0 1 2 3 4
```

Trying to model the `for` loop with a railroad switch and a loop of track requires more visual elements than the picture provides. The important points below are: initialization occurs one time, evaluation occurs every time through the loop, and manipulation occurs at the tail-end of the loop right before evaluation.



Here is an example of a loop within a loop. The "outer" loop produces 3 rows, and the "inner" loop results in 6 columns. The body of the inner loop consists of a single statement. The outer loop's body is two statements: the inner `for` loop [6] (which includes the first `cout` statement), and the second `cout` statement [7].

```
for (int i=0; i < 3; i = i + 1) {
   for (int j=0; j < 6; j = j + 1)              [6]
      cout << i << j << ' ';
   cout << '\n';                                [7]
}

// 00 01 02 03 04 05
// 10 11 12 13 14 15
// 20 21 22 23 24 25
```

There is a lot more power and complexity available with the `if`, `for`, and `while` statements, but this chapter has presented the 20% of the features that afford 80% of the leverage.

Practice. Complete increment 1 (App A and App E) and increment 2 (App B and App C).

Chapter 3 ... Arrays, Input, stringstream

Arrays. In the example below, to add and use a fourth integer requires: declaring an additional `int` variable, assigning it a value, and adding it to the `cout` statement.

```
int first, second, third;
first = 1;
second = 2;
third = 3;
cout << first << ' ' << second << ' ' << third << '\n';

// 1 2 3
```

Computer languages provide a mechanism (called "array") for declaring and referencing a "list" of variables using a single name. When an array variable is declared, its size goes inside square brackets [1]. Each element in the array is read or assigned by putting its position in the list in square brackets [2]. The first element of the array is addressed with the value 0 (not 1).

```
int numbers[3];                              // [1]
numbers[0] = 1;                              // [2]
numbers[1] = 2;
numbers[2] = 3;
cout << numbers[0] << ' ' << numbers[1]   // [2]
     << ' ' << numbers[2] << '\n';

// 1 2 3
```

Arrays are like a patch panel. Each port is an element; and having them arranged in close proximity affords many benefits: elements have an implicit order, each element has neighbors that are readily "computed", and the neighborhood can be easily "canvased".

Data structures are often implemented with arrays. A data structure is data that is designed and structured to provide peculiar efficiency for a class of problems.

The real leverage of arrays is that they can be accessed by using another variable, and they can be traversed by using a loop. Below, a sequence of four integer variables is defined. Each element in the sequence is accessed iteratively by using the variable "`i`".

```
int numbers[4];
for (int i=0; i < 4; i = i + 1)
   numbers[i] = i + 1;
for (int i=0; i < 4; i = i + 1)
   cout << numbers[i] << ' ';
cout << '\n';

// 1 2 3 4
```

Specifying the size of an array, and the boundary for traversing the array can be done with an integer variable. That variable below is `SIZE`. It is declared at [3] and used at [4]. With this kind of foresight and design (aka "indirection"), the size of an array can be easily changed, and no other code needs to change. The loop demonstrates prompting for two integers, and using one as an index to select a specific "element" of the array, and using the other to give that element a new value [5].

```
int SIZE = 6;                              [3]
int array[SIZE];                           [4]
for (int i=0; i < SIZE; i = i + 1)         [4]
   array[i] = 0;
int position, value;
while (true) {
   for (int i=0; i < SIZE; i = i + 1)      [4]
      cout << array[i] << ' ';
   cout << "\nPosition value: ";
   cin >> position >> value;
   if (position == -1)
      break;
   array[position] = value;                [5]
}

// 0 0 0 0 0 0
// Position value: 0 6
// 6 0 0 0 0 0
// Position value: 5 1
// 6 0 0 0 0 1
// Position value: -1 1
```

Here is another demonstration of the usefulness of arrays. The first two elements are assigned the value one, and then each subsequent element is assigned a value computed by adding the values of its two predecessors (element "i-2" and element "i-1") [6].

```
int array[10];
array[0] = 1;
array[1] = 1;
for (int i=2; i < 10; i = i + 1)
   array[i] = array[i-2] + array[i-1];      [6]
for (int i=0; i < 10; i = i + 1)
   cout << array[i] << ' ';
cout << '\n';

// 1 1 2 3 5 8 13 21 34 55
```

Arrays can be initialized when they are defined by putting a list of values in curly braces. If you put fewer elements in the initialization list than are defined in the array, all the latter elements are automatically assigned the value "0" (or whatever makes the most sense for the type of the array).

```
double array[10] = { 2.0, 3.1, 4.2 };
for (int i=0; i < 10; i = i + 1)
   cout << array[i] << ' ';
cout << '\n';

// 2 3.1 4.2 0 0 0 0 0 0 0
```

It is possible to define an array of arrays (also known as a two dimensional array). The variable `matrix` below is an array of 3 arrays, each of which is an array of 4 integers. Notice that the first dimension is being accessed using the variable "i" (inside square brackets), and the second dimension is being accessed with the variable "j".

```
int matrix[3][4] = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 }, { 9, 10, 11, 12 } };
for (int i=0; i < 3; i = i + 1)
   for (int j=0; j < 4; j = j + 1)
      cout << matrix[i][j] << ' ';
cout << '\n';

// 1 2 3 4 5 6 7 8 9 10 11 12
```

Below, the array `values` simulates a bunch of random numbers. The array `buckets` will be used to "count" the number of occurrences of the digits 0 through 9. The first loop iterates through each element of the `values` array, and uses that element to index into the `buckets` array. Whatever value is found at the location is incremented by one [7]. The second loop outputs "column headers" [8], and the third loop outputs the computed counts [9].

```
int buckets[10] = { 0 };
int values[30]  = { 4, 7, 9, 1, 3, 2, 0, 5, 2, 6, 8, 4, 5, 3, 6,
                    1, 3, 0, 4, 8, 3, 5, 1, 9, 5, 2, 7, 8, 3, 1 };
for (int i=0; i < 30; i = i + 1)
   buckets[ values[i] ] = buckets[ values[i] ] + 1;     [7]
for (int i=0; i < 10; i = i + 1)                        [8]
   cout << i << ' ';
cout << '\n';
for (int i=0; i < 10; i = i + 1)                        [9]
   cout << buckets[i] << ' ';
cout << '\n';

// 0 1 2 3 4 5 6 7 8 9                                  [8]
// 2 4 3 5 3 4 2 2 3 2                                  [9]
```

In the **Selection** section, a multi-branch selection statement was demonstrated. This kind of code is very common. Since this is a sequence of instructions to achieve a desired outcome, it could be characterized as an algorithm.

```
if (num < 20)
   cout << "num is less than 20\n";
else if (num < 40)
   cout << "num is between 20 and 39\n";
else if (num < 60)
...
```

Just as there is an equivalence between mass and energy, algorithm and data structure are often interchangeable.

Instead of using an enormous "if ... else if ... else if ..." algorithm, the boundary conditions can be loaded in an array (a type of data structure), and the messages to output can also be loaded in an array. The inner loop [11] is used to find the correct range, and the outer loop [10] is used to step through the test values. Notice that the variable "j" is declared in the outer loop, and then initialized in the inner loop. This is because the correct value for "j" is found in the inner loop [12], but it needs to hang around so that it can be used in the outer loop [13].

```
int   boundaries[4] = { 20, 40, 60, 80 };
string  messages[5] = { " is less than 20\n",
                        " is between 20 and 39\n",
                        " is between 40 and 59\n",
                        " is between 60 and 79\n",
                        " is greater than or equal to 80\n" };
int test_values[8] = { 19, 20, 39, 40, 59, 60, 79, 80 };
for (int i=0, j; i < 8; i = i + 1) {                    [10]
   for (j=0; j < 4; j = j + 1)                          [11]
      if (test_values[i] < boundaries[j])
         break;                                         [12]
   cout << test_values[i] << messages[j];               [13]
}

// 19 is less than 20
// 20 is between 20 and 39
// 39 is between 20 and 39
// 40 is between 40 and 59
// 59 is between 40 and 59
// 60 is between 60 and 79
// 79 is between 60 and 79
// 80 is greater than or equal to 80
```

The next example is a piece of a larger example in the Introduction chapter. The goal is to convert a julian number (a number in the range 1..365) to its corresponding month/day representation. The array at [13] is a table of the number-of-days in each month. `test_data` is a list of interesting test cases. The outer loop [14] steps through the test data. The inner loop [15] steps through the days table until the size of the test data no longer exceeds the size of the current month. The "heavy lifting" appears at [16] – the test data is iteratively reduced in size by the number of days in the current month.

```
int test_data[7] = { 1, 31, 32, 59, 60, 100, 365 };
                              // Jan Feb Mar Apr May Jun
int days_per_month_table[12] = { 31, 28, 31, 30, 31, 30,        [13]
                                  31, 31, 30, 31, 30, 31 };
                              // Jul Aug Sep Oct Nov Dec
for (int i=0, j; i < 7; i = i + 1) {                            [14]
   for (j=0;                                                    [15]
        test_data[i] > days_per_month_table[j];
```

```
          j = j + 1)
        // body of inner loop
        test_data[i] = test_data[i] - days_per_month_table[j];    [16]
    cout << (j+1) << '/' << test_data[i] << "   ";
}
cout << '\n';

// 1/1   1/31   2/1   2/28   3/1   4/10   12/31
```

It would be nice if an array variable could be given to `cout` and `cout` would print the elements of the array. This is being attempted at [19] and it doesn't work. The output at [21] is seemingly not intelligible. [What is being printed is the "base address" of the array.] Sending an array of type `char` to `cout` seems to behave a little better [20] – but there are four characters of garbage at the end [22].

C++ will accept an array of type `char`, but it doesn't know where the end of the array is (that's why the garbage at the end). If you put a "null" character (a zero) at the end of a `char` array [17], C++ can now identify the end of the array [23]. Since arrays of characters are so very useful, C++ allows them to be easily initialized by using enclosing the desired characters in double quotes [18]. When you use the double quotes feature, the compiler puts the null character at the end for you automatically.

```
int  nums[]   = { 1, 2, 3 };
char chars[]  = { '1', '2', '3' };
char first[]  = { 'o', 'n', 'e', ' ', 't', 'w', 'o', 0 };   [17]
char second[] = "one two";                                  [18]
cout << "nums   is --" << nums   << "--\n";                 [19]
cout << "chars  is --" << chars  << "--\n";                 [20]
cout << "first  is --" << first  << "--\n";
cout << "second is --" << second << "--\n";

// nums   is --0x22ff60--                                   [21]
// chars  is --123w^#=--                                    [22]
// first  is --one two--                                    [23]
// second is --one two--                                    [24]
```

Here is a simple "Wheel of Fortune" example. When the user enters a character [25], it is: appended to the end of the `guesses` string [26], used to search the `answer` array [27], and used to update the `puzzle` array [28].

```
char   answer[] = "too close for comfort";
char   puzzle[] = "--- ----- --- -------";
string guesses;
char   letter;
int    SIZE = 21;
while (true) {
   cout << '\n' << puzzle
        << '\n' << "previous guesses: " << guesses
        << '\n' << "next guess:       ";
   cin >> letter;                                           [25]
   guesses = guesses + letter;                              [26]
   for (int i=0; i < SIZE; i = i + 1)
      if (answer[i] == letter)                              [27]
         puzzle[i] = letter;                                [28]
}
```

```
// --- ----- --- -------
// previous guesses:
// next guess:       f                                          [25]
//
// --- ----- f-- ---f---
// previous guesses: f                                          [26]
// next guess:       o                                          [25]
//
// -oo --o-- fo- -o-fo--
// previous guesses: fo                                         [26]
// next guess:       r                                          [25]
//
// -oo --o-- for -o-for-
// previous guesses: for                                        [26]
// next guess:
```

Practice. Complete increment 2 (App A, App D, App E) and increment 3 (App B, App C).

Input (reprise). `cin` supports all C++ variable types. The programmer should not assume that the user will provide the input that is expected, and should be careful to validate keyboard input before anything "undefined" is attempted (e.g. assigning non-number input to a number variable).

```
string first;
int    second;
char   third;
cout << "Input a word, a number, and a character: ";
cin >> first >> second >> third;
cout << "first is " << first << ", second is " << second
     << ", third is " << third << '\n';

// Input a word, a number, and a character: eleven 11 b
// first is eleven, second is 11, third is b
```

Receiving an entire line of input is demonstrated below. `getline()` is a function (functions are yet to be presented) that expects to be find in the parentheses: the `cin` object and a `string` variable [1]. The user's input is assigned to the `string` variable and returned to the code that "called" `getline()`.

```
string str;
cout << "Input: ";
getline( cin, str );                                    [1]
cout << "input was --" << str << "--\n";

// Input: one two three
// input was --one two three--
```

Validating user input starts with the `getline()` function [2], and then its up to the programmer's ingenuity. Here is a very basic algorithm. You can step through each character in the `string` [3] (we saw the `size()` function in the `string` class in a previous section). Checking that the current

character is in the range '0' to '9' (inclusive) [4] may be sufficient; but [5] and [6] demonstrate "boundary conditions" that may need to be addressed.

```
string line;
while (true) {
   cout << "Enter a number: ";
   getline( cin, line );                                  [2]
   int i;
   for (i=0; i < line.size(); i = i + 1)                  [3]
      if (line[i] < '0'  ||  line[i] > '9')               [4]
         break;
   if (i < line.size()) cout << "   NOT a number\n";
   else                 cout << "   IS a number\n";
}

// Enter a number: 123
//    IS a number
// Enter a number: 12a
//    NOT a number
// Enter a number: -12                                    [5]
//    NOT a number
// Enter a number: 1.2                                    [5]
//    NOT a number
// Enter a number: 1 2
//    NOT a number
// Enter a number: 001
//    IS a number
```

Another motivation for reading an entire line of input into a single variable is the following. The `cin` object returns a single "token" for each variable it is asked to initialize. [A "token" is a sequence of characters delimited by "white space" (space, tab, or carriage return).] At [2], a single variable is specified, but at [3], the user enters three tokens. `cin` immediately assigns the first token to the variable `str`. When flow of control returns to the `cin` statement, it does not wait for new user input, because it already has the remaining input from the previous iteration [4].

```
string str;
while (true) {
   cout << "Input: ";
   cin >> str;                                      [2]
   cout << "input was --" << str << "--\n";
}

// Input: one two three                             [3]
// input was --one--
// Input: input was -two--                          [4]
// Input: input was --three--                       [4]
// Input:
```

stringstream. In the **Variables** section, `string` was introduced as a class. `stringstream` is the second class we'll use. It is an excellent tool for converting a value from one type to another (e.g. from string to int). It is also used with `getline()` to support heterogeneous user input, and "tokenizing" user input.

It would be nice if ... C++ was smart enough to automatically handle the type conversion represented at [1] (int to string) and [2] (string to int). The compiler errors are presented as comments.

```
int    num = 123;
string str = num;                                    [1]
// ERROR: invalid conversion

string str = "123";
int    num = str;                                    [2]
// ERROR: cannot convert `string' to `int' in initialization
```

There are many tools for doing type conversion, and the easiest is the `stringstream` class. The programmer can "write" to a `stringstream` object just like one would `cout` [3], and "read" from a `stringstream` object just like one would `cin` [4]. After those two statements, a `string` variable has been assigned an `int` value.

The inverse is also possible. Notice at [6] that the string includes a fractional part, but after the conversion, the fractional part has been truncated [7]. To reuse a `stringstream` object it must first be "cleared" [5].

```
stringstream  converter;
int           num = -123;
string        str;
converter << num;                                    [3]
converter >> str;                                    [4]
cout << "str is **" << str << "**\n";

converter.clear();                                   [5]
str = "3.14159";                                     [6]
converter << str;
converter >> num;
cout << "num is **" << num << "**\n";

// str is **-123**
// num is **3**                                      [7]
```

`stringstream` is part of the C++ standard library, and requires the following lines at the top of the program.

```
#include <sstream>
using std::stringstream;
```

Another prominent purpose for the `stringstream` class is to divide a string into its constituent tokens. Below, the string `phrase` is "written" to the object `tokenizer` [8]. Individual "words" are then extracted until "end of string" is reached [9].

```
stringstream tokenizer;
string word, phrase = "the quick brown fox";
tokenizer << phrase;                                 [8]
while (tokenizer >> word)                            [9]
   cout << "word --" << word << "--\n";
cout << "done\n";

// word --the--
// word --quick--
```

```
// word --brown--
// word --fox--
// done
```

Suppose your application needs to iteratively accept commands like "quit" or "help" – and – multiple integers. The former are strings, and the latter are integers. `getline()` can move any and all user input to a string variable [11]. The string can then be evaluated [12]. If no "command" is found, the string is loaded in a `stringstream` object and the desired integers are extracted [13].

This technique has the added benefit of disposing of unwanted input. At [14] three numbers were requested, but four were provided. The unsolicited number was simply ignored. [Miscellaneous note: [10] demonstrates how to include the double quote character in a string literal.]

```
string line;
int    first, second, third;
while (true) {
   cout << "Enter 3 numbers or \"quit\": ";         [10]
   getline( cin, line );                            [11]
   if (line == "quit")                              [12]
      break;
   stringstream tokenizer;
   tokenizer << line;                               [13]
   tokenizer >> first >> second >> third;
   cout << first << "^^^" << second << "^^^" << third << '\n';
}

// Enter 3 numbers or "quit": -123 234 -345 456     [14]
// -123^^^234^^^-345
// Enter 3 numbers or "quit": quit
```

Practice. Complete increment 3 of Appendix A, increment 4 of Appendix B and Appendix C.

Chapter 4 ... Wrestling with Arrays

Many of the practice projects use arrays extensively. This section provides additional coverage of the subject. It discusses arrays and loops in the context of algorithms (sort and search) and data structures (accessing an array with an array).

Consider how to sort the array of `numbers`. A very simple strategy might be: step through the entire array and move the largest value to the end. This could be accomplished by comparing adjacent elements, and swap the two values if the left number is larger than the right number [1].

```
int numbers[] = { 9,8,7,6,5,4,3,2,1,3,5,7,9,2,4,6,8,1 };
for (int j=0, temp; j < 17; j = j + 1)
   if (numbers[j] > numbers[j+1]) {                          [1]
      temp         = numbers[j];
      numbers[j]   = numbers[j+1];
      numbers[j+1] = temp;
   }
for (int i=0; i < 18; i = i + 1)
   cout << numbers[i] << ' ';
cout << '\n';

// 8 7 6 5 4 3 2 1 3 5 7 9 2 4 6 8 1 9
```

That is sufficient to move the largest number to the last position [2]. We could then repeat the process all over again to move the next largest number to the next-to-last position [3]

```
int numbers[] = { 9,8,7,6,5,4,3,2,1,3,5,7,9,2,4,6,8,1 };
for (int j=0, temp; j < 17; j = j + 1)                       [2]
   if (numbers[j] > numbers[j+1]) {
      temp         = numbers[j];
      numbers[j]   = numbers[j+1];
      numbers[j+1] = temp;
   }
for (int j=0, temp; j < 16; j = j + 1)                       [3]
   if (numbers[j] > numbers[j+1]) {
      temp         = numbers[j];
      numbers[j]   = numbers[j+1];
      numbers[j+1] = temp;
   }
for (int i=0; i < 18; i = i + 1)
   cout << numbers[i] << ' ';
cout << '\n';

// 7 6 5 4 3 2 1 3 5 7 8 2 4 6 8 1 9 9
```

Notice that the upper bound on the first loop is "N-1", and on the second loop it is "N-2". Each subsequent loop will always be one less than its predecessor. Instead of enumerating a seemingly endless sequence of loops, introducing an outer loop [4] would save a lot of lines of code – and – allow any size array to be sorted. The outer loop's "i" is used to limit the inner loop's "j" [5].

```
int numbers[] = { 9,8,7,6,5,4,3,2,1,3,5,7,9,2,4,6,8,1 };
int temp;
for (int i=17; i > 0; i = i - 1)                             [4]
```

```
   for (int j=0, temp; j < i; j = j + 1)                    [5]
      if (numbers[j] > numbers[j+1]) {
         temp         = numbers[j];
         numbers[j]   = numbers[j+1];
         numbers[j+1] = temp;
      }
for (int i=0; i < 18; i = i + 1)
   cout << numbers[i] << ' ';
cout << '\n';

// 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
```

This algorithm is called bubble sort – the smallest values in the array "bubble" to the top.

Once an array is sorted, it is much easier to search. Consider the sorted array `letters` [6] and the elements of `test_data` [7] that will be sought. The output below is the goal.

```
char letters[] = "abcdefghijklmnopqrstuvwxyz";                  [6]
char test_data[] = { '5', 'a', 'e', 'j', 'o', 't', 'z', '|' };  [7]
cout << "01234567890123456789012345\n" << letters << '\n';
...

// 01234567890123456789012345
// abcdefghijklmnopqrstuvwxyz
// 5 not found
// a is at position 0
// e is at position 4
// j is at position 9
// o is at position 14
// t is at position 19
// z is at position 25
// | not found
```

In the field of gunnery, it is common to employ "bracketing". If the first shell lands short of the target, the elevation of the howitzer is adjusted so that the next shell will land long. Then the elevation is adjusted again so that the third shell "splits the difference". The enclosing "bracket" shrinks by half with each shell fired and resulting split-the-difference adjustment.

The "binary search" algorithm uses the same strategy to find its "target". The loop at [8] steps through each element of `test_data`. The variables `lower` and `upper` maintain the limits of the "bracket", and `mid` is computed to split their difference [9]. If the character in the sorted array at the `mid` position is greater than the test character [10], then `upper` is moved closer to `lower`. If the inverse is true [11], then `lower` is moved closer to `upper`. Otherwise, the two are equal and the search is complete.

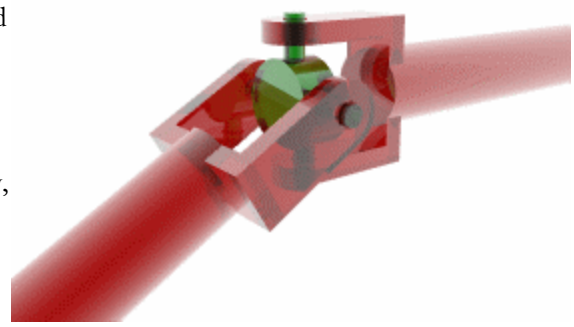
When the while loop exits, there are two possibilities: the element was found [12], or `lower` and `upper` crossed each other because the element was not found [9].

```
for (int i=0, lower, mid, upper; i < 8; i = i + 1) {       [8]
   lower = 0;
   upper = 25;
   while (lower <= upper) {                                 [9]
      mid = (lower + upper) / 2;
      if (letters[mid] > test_data[i])                     [10]
         upper = mid - 1;
```

```
      else if (letters[mid] < test_data[i])                [11]
         lower = mid + 1;
      else  // letters[mid] == test_data[i]                [12]
         break;
   }
   if (letters[mid] == test_data[i])                       [12]
      cout << test_data[i] << " is at position " << mid << '\n';
   else
      cout << test_data[i] << " not found\n";
}
```

Practice. Complete increment 5 of Appendix C, increment 3 of Appendix D and Appendix E.

This section demonstrates accessing an array with an array. Arrays offer flexibility like hinges. Instead of a fixed attach point, a hinge offers "range of motion" in one dimension. A scalar variable can provide access to a single value, but an array provides access (or range of motion) to a sequence of values. If the index into an array is another array, then range of motion is extended to two dimensions.



The universal joint to the right provides two "degrees of freedom". Using an array to access another array provides the same kind of decoupling.

An anagram takes a sentence, applies an encoding function, and produces a puzzle to solve. The encoding that will be used here is to reverse the order of the alphabet [13]. The anagram produced is at [14].

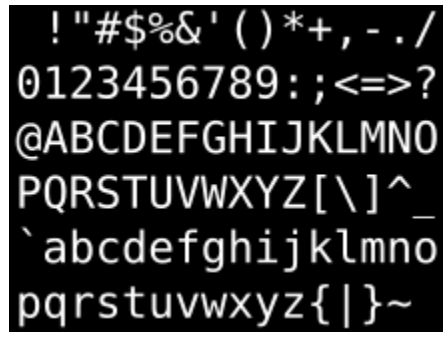
```
// abcdefghijklmnopqrstuvwxyz
// zyxwvutsrqponmlkjihgfedcba                          [13]
//
// the early worm is for the birds.
// gsv vziob dlin rh uli gsv yriwh.                    [14]
```

Here is a possible design for encoding an anagram. The `letters` array is searched for each letter in the `answer` array [16], and when its position is found [17], that position is used on the `mapping` array to initialize the `encoded` array. The code at [15] transfers any non-lower-case character from `answer` to `encoded`.

```
char answer[33] = "the early worm is for the birds.";
char encoded[33];
char letters[] = "abcdefghijklmnopqrstuvwxyz";
char mapping[] = "zyxwvutsrqponmlkjihgfedcba";
for (int i=0; i < 33; i = i + 1)
   if (answer[i] < 'a'  ||  answer[i] > 'z')           [15]
      encoded[i] = answer[i];
   else
```

```
      for (int j=0; j < 26; j = j + 1)                    [16]
         if (answer[i] == letters[j]) {                   [17]
            encoded[i] = mapping[j];                      [18]
            break;
         }
cout << letters << '\n' << mapping << "\n\n";
cout << answer << '\n' << encoded << '\n';
```

C++ allows characters to be treated very much like integers. The loop below prints the lower case character set.




```
for (char ch='a'; ch <= 'z'; ch = ch + 1)
   cout << ch;
cout << '\n';

// abcdefghijklmnopqrstuvwxyz
```

The ASCII printable characters are in the range 32 to 126 (lower case letters go from 97 to 122). Instead of having parallel `letters` and `mapping` arrays, it would be nice if each character in `answer` could be used to index into `mapping` directly. Here we have two degrees of freedom: the loop control variable 'i' gets us to the correct character in the `mapping` array, and that character points to the encoding character.

```
encoded[i] = mapping[ answer[i] ];
```

The trick then becomes: how to replace 'a'..'z' with 'z'..'a' in the ASCII encoding sequence. The `mapping` array now needs to have elements 0 to 122. Instead of an enormous initialization string, [19] employs an obscure C feature: breaking a string literal into two or more smaller string literals (the compiler will patch them together to produce a single string literal).

The loop at [20] is the same as the previous design, and [21] is the new leverage for which we were shooting.

```
char answer[33] = "the early worm is for the birds.";
char encoded[33];
               // 0123456789012345678901234567890123456789
char mapping[] = "                                !\"#$%&'"      [19]
                 "()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNO"
                 "PQRSTUVWXYZ[\\]^_`zyxwvutsrqponmlkjihgfedcba";
for (int i=0; i < 33; i = i + 1)                                [20]
   if (answer[i] < 'a'  ||  answer[i] > 'z')
      encoded[i] = answer[i];
   else
      encoded[i] = mapping[ answer[i] ];                        [21]
cout << "abcdefghijklmnopqrstuvwxyz" << '\n';
for (char ch='a'; ch <= 'z'; ch = ch + 1)
   cout << mapping[ch];
cout << "\n\n" << answer << '\n' << encoded << '\n';
```

Yet another optimization allows us to shrink the 123-element `mapping` array to 26 elements: characters can be manipulated with arithmetic.

```
int first = 'a' - 'a', second = 'z' - 'a';
cout << "first is " << first << ", second is " << second << '\n';
```

```
// first is 0, second is 25
```

The `mapping` array is a shadow of its former self [22], and the array-on-array statement [23] has been adapted.

```
char answer[33] = "the early worm is for the birds.";
char encoded[33];
char mapping[] = "zyxwvutsrqponmlkjihgfedcba";                    [22]
for (int i=0; i < 33; i = i + 1)
   if (answer[i] < 'a'  ||  answer[i] > 'z')
      encoded[i] = answer[i];
   else
      encoded[i] = mapping[ answer[i] - 'a' ];                   [23]
cout << "abcdefghijklmnopqrstuvwxyz" << '\n' << mapping;
cout << "\n\n" << answer << '\n' << encoded << '\n';
```

Practice. Complete increment 4 of Appendix E.

Chapter 5 ... Functions

In previous sections, examples of "functions" were introduced. One of those early previews is replayed below. `getline()` is a function that accepts the `cin` object and a `string` variable [1], and returns `true` if more input remains and `false` if "end of file" is reached [2].


```
string line;
while (getline( cin, line ))                [1]
   cout << "   " << line << '\n';
cout << "   done\n";

// first line
//    first line
// second line
//    second line
// ^Z                                       [2]
//    done
```

The function is a mechanism for encapsulating functionality so that it can be used as a "black box" – when the user supplies one or more inputs, the box provides one or more outputs. It is like a change machine: the user inserts a dollar, and the machine returns coins.

Consider the code below. Let's move the second `cout` statement to a function.

```
int main() {
   cout << "header ...\n";
   cout << "the good stuff in the middle\n";
   cout << "... trailer\n";
}

// header ...
// the good stuff in the middle
// ... trailer
```

The motivation for this activity is [3] below. `encapsulation()` is a function, and the syntax at [3] is referred to as a "function invocation".

```
int main() {
   cout << "header ...\n";
   encapsulation();                                  [3]
   cout << "... trailer\n";
}
```

At a minimum, each function is composed of: a return type [4], the function's name [5], and a body [6]. The `encapsulation` function below has a body that consists of the second `cout` statement from the previous `main()` [6]. It does not need to return anything to the code that called (or invoked) it, so its return type is specified as `void` [4].

```
void                                                 [4]
encapsulation() {                                    [5]
```

```
   cout << "the interesting stuff in the middle\n";    [6]
}
```

A function can accept information supplied by the caller of the function. That information can be packaged as one or more arguments (or parameters). Below, the text that had previously been "hard wired" in the function's `cout` statement, is now being specified by the calling code [10], and "passed" as an argument. Each argument has a type [7] and a name [8], and the argument's name can then be referenced in the function's body [9].

```
//                    [7]     [8]
void encapsulation( string argument ) {
   //         [9]
   cout << argument << '\n';
}

int main() {
   cout << "header ...\n";
   //              [.................10.................]
   encapsulation( "the interesting stuff in the middle" );
   cout << "... trailer\n";
}
```

If a function needs to return information to it's caller, the function can define a return type [11], and then explicitly return a literal or variable in the body that agrees with the return type [12]. The function `min()` is then invoked with two `int` variables [13], and its return value is immediately "forwarded" to the `cout` statement.

```
int                             // return type           [11]
min( int one, int two ) {       // 2 input arguments
   if (one < two)
      return one;               // return value          [12]
   return two;                  // return value          [12]
}

int main() {
   int first  = 48;
   int second = 24;                          [....13.....]
   cout << "the smaller value is " << min( first, second ) << '\n';
}

// the smaller value is 24
```

Here's another example of a function with a defined return type. The function `to_upper()` accepts an argument of type `string` and returns a value of type `string`. [14]. The body of the function makes a copy of the argument [15] and iterates through the copy [16]. When a lower case letter is found, it is converted to upper case [17] (e.g. the encoded value of 'a' is 97 and 'A' is 65). The `copy` variable is then returned [18], and subsequently handed to `cout` [19].

```
string to_upper( string argument ) {                    [14]
   string copy = argument;                               [15]
   for (int i=0; i < copy.size(); i = i + 1)             [16]
      if (copy[i] >= 'a'  &&  copy[i] <= 'z')
         copy[i] = copy[i] - 32;                         [17]
   return copy;                                          [18]
```

```
}

int main() {
   string phrase  = "Here is Camel Case";
   cout << to_upper( phrase ) << '\n';                [19]
}

// HERE IS CAMEL CASE
```

Instead of the `to_upper()` function returning a `string` value, what about having it modify the `string` argument it was passed [20]? The variable `phrase` is passed in (and ostensibly passed out) at [21]. When it is subsequently output [22], the value has not changed [23].

```
void to_upper( string argument ) {
   for (int i=0; i < argument.size(); i = i + 1)
      if (argument[i] >= 'a'  &&  argument[i] <= 'z')
         argument[i] = argument[i] - 32;              [20]
}

int main() {
   string phrase  = "Here is Camel Case";
   to_upper( phrase );                                [21]
   cout << phrase << '\n';                            [22]
}

// Here is Camel Case                                 [23]
```

The reason the previous example did not work is because parameters passed to functions are "passed by value" by default. The `argument` parameter above does not represent the same piece of computer memory as the `phrase` variable. It is instead a **copy** of the `phrase` variable, and "goes away" when the function returns.

To tell the compiler that you don't want "pass by value", but instead want the variable that is passed and the parameter that is defined to refer to the same piece of computer memory – you supply an '&' character immediately after the type of the function's parameter [24]. This results in what is called "pass by reference" – the function works on the original variable rather than a copy of the original.

```
//                  [24]
void to_upper( string& argument ) {
   for (int i=0; i < argument.size(); i = i + 1)
      if (argument[i] >= 'a'  &&  argument[i] <= 'z')
         argument[i] = argument[i] - 32;
}

int main() {
   string phrase  = "Here is Camel Case";
   to_upper( phrase );
   cout << phrase << '\n';
}

// HERE IS CAMEL CASE
```

"Pass by value" could be characterized as: what happens in Vegas, stays in Vegas. "Pass by reference" could lamely be summarized as: you put your left foot in, you pull your left foot out.

When an array is passed to a function, C++ effectively provides "pass by reference" by default. At [25], no special syntax was used; yet the changes made at [26] actually come through at [27].

```
void double_array( int array[5] ) {                    [25]
   for (int i=0; i < 5; i = i + 1)
      array[i] = array[i] * 2;                         [26]
}

int main() {
   int array[] = { 1, 2, 3, 4, 5 };
   double_array( array );
   for (int i=0; i < 5; i = i + 1)
      cout << array[i] << ' ';                         [27]
   cout << '\n';
}

// 2 4 6 8 10                                          [27]
```

C++ does not require the size of the array to be specified in a function's argument list [28]. But – passing that size allows the function to be more general.

```
                     [...28...]
void double_array( int array[], int length ) {
   for (int i=0; i < length; i = i + 1)
      array[i] = array[i] * 2;
}

int main() {
   int array[] = { 1, 2, 3, 4, 5 };
   double_array( array, 5 );
   ...
}
```

Passing a multidimensional array to a function undoes the previous discussion. At [29], leaving out the size of both dimensions causes a compiler error. The response that limitation is at [30]. Since the first dimension can be left out, it has been, and passed as a parameter instead. The second dimension is hard-wired in the function's argument list, and in the inner loop [31].

```
// void print_matrix( int matrix[][] ) {                    [29]
// ERROR: declaration of 'matrix' as multidimensional array
// must have bounds for all dimensions except the first
void print_matrix( int matrix[][6], int first_dim ) {       [30]
   for (int i=0; i < first_dim; i = i + 1) {
      for (int j=0; j < 6; j = j + 1)                       [31]
         cout << setw(3) << matrix[i][j];
      cout << '\n';
}  }

int main() {
   int matrix[3][6] = { {1,2,3,4,5,6}, {7,8,9,10,11,12},
                        {13,14,15,16,17,18} };
   print_matrix( matrix, 3 );
}

//  1  2  3  4  5  6
```

```
//  7  8  9 10 11 12
// 13 14 15 16 17 18
```

As programs get larger, how the functions are laid out physically (how they are grouped in separate files and how they are ordered) becomes important. As a very small example, maybe you want the main() function to appear first in your C++ file. When the compiler reaches [32], it does not know what to make of the token "do_something".

```
int main() {
   // ERROR: 'do_something' undeclared
   do_something( "some input" );                    [32]
}

void do_something( string arg ) {
   cout << "do_something: " << arg << '\n';
}
```

At a minimum, the compiler needs a function's "declaration" (aka its "signature") before it encounters an invocation of the function. A function declaration consists of the function's: name, return type, and the number and type of all arguments [33]. The function definition [34] can then appear anywhere – as long as it can be found when the program's executable is constructed.

```
// function declaration
void do_something( string );                        [33]

int main() {
   // function invocation
   do_something( "some input" );
}

// function definition
void do_something( string arg ) {                   [34]
   cout << "do_something: " << arg << '\n';
}

// do_something: some input
```

In the **Variables** section, the notion of "scope" was introduced – every variable is defined in some "block" or "scope" that defines the life-time of that variable. Here is a demonstration that variables can be defined at "global" scope [35]. This variable can be accessed any where in the program [36].

```
string global_variable;                                 [35]

void do_something() {
   cout << "do_something: " << global_variable << '\n';   [36]
}

int main() {
   global_variable = "some value";                      [36]
   do_something();
}

// do_something: some value
```

Global variables allow programmers to be lazy. Instead of exercising the discipline of defining and passing function arguments – the programmer defines a few global variables, and then any and all functions can write to or read from those variables. This practice leads to programs that are very difficult to debug and maintain. Coupling is a serious design issue, and global variables invite chaotic coupling that is impossible to keep up with.

Practice. Complete increment 5 of App B and App E, increments 4/5/6 of Appendix D.

Let's consider some standard functions supplied by C++. Several projects will need random numbers. The output of `rand()` can be "scaled" by using remainder division [37]. `rand()` needs a random "seed" to produce its randomness. Without this seed, it produces the same sequence every time its enclosing program is executed [38].

```
for (int i=0; i < 20; i = i + 1)
   cout <<  rand() % 20  << ' ';                        [37]
cout << '\n';

// > demo.exe
// 1 7 14 0 9 4 18 18 2 4 5 5 1 7 1 11 15 2 7 16      [38]
// > demo.exe
// 1 7 14 0 9 4 18 18 2 4 5 5 1 7 1 11 15 2 7 16      [38]
// > demo.exe
// 1 7 14 0 9 4 18 18 2 4 5 5 1 7 1 11 15 2 7 16      [38]
```

The "seed" for `rand()` is an integer supplied to the standard `srand()` function [40]. In the following example, the integer is specified by the user [39].

```
int seed;
cout << "Enter seed: ";
cin >> seed;                                                    [39]
srand( seed );                                                  [40]

for (int i=0; i < 20; i = i + 1)
   cout << setw(3) << rand() % 20;
cout << '\n';

// > demo.exe
// Enter seed: 123                                              [39]
//   0 13 15  4  3  5  9 18  0  4  9 10  0  3  9  6 18 15  7 10
// > demo.exe
// Enter seed: 231                                              [39]
//  12 13 13 10  5  7  5  5 19 12 15  7 15 14 11  4  9 12 17 11
// > demo.exe
// Enter seed: 321                                              [39]
//   6 18 17  0  5 13  2 14 14  2 11 11 13 16 17  8  4 14 13 19
// > demo.exe
// Enter seed: 321                                              [39]
//   6 18 17  0  5 13  2 14 14  2 11 11 13 16 17  8  4 14 13 19
```

A more convenient way of supplying the requisite seed is the standard `time()` function. `time()` returns the time since midnight GMT, 1 January 1970 (measured in seconds). It expects a "pointer"

(introduced later), but the value null (aka zero) can be passed instead [41]. Every time the enclosing program is executed, the number of seconds returned by `time()` is different – so – the subsequent sequence of numbers generated will be random.

```
srand( time(0) );                                                    [41]

for (int i=0; i < 20; i = i + 1)
   cout << setw(3) << rand() % 20;
cout << '\n';

// > demo.exe
//  12  7 11  8 10 10 11 10 19 11  7  9  9 15 19 10  7 11 11  8
// > demo.exe
//   2  4  7  2  6 10  9  9  5 15  3  1 10 18 11  8 17 19 15  6
// > demo.exe
//   8  7  9  1  7 16  0 18  6 18  0 14 13 16  9  1 13  0  1 15
```

Practice. Complete increment 4 of Appendix A and increment 6 of App B and App C.

Chapter 6 ... struct, vector, deque, map

`struct`. In the Arrays section, creating a sequence of variables under a single name was discussed [1]. This section introduces a new way to define a "composite" type. `struct` allows the programmer to define a new "type" (something that behaves like `int` or `string` – you can subsequently define and manipulate instances of it). The syntax for `struct` is at [2]. Each variable in the `struct` is known as a "member". Using the new type is at [3] (notice the ';' after the right brace), and using the instance of that new type is at [4] (a '.' separates the `struct`'s instance name and the member name).

```
// three integers, each with their own name
int first = 1, second = 2, third = 3;

// three integers under a single name
int numbers[3] = { 1, 2, 3, };                          [1]

// define a new type
struct Trio {                                           [2]
   int first, second, third;
};

// create and initialize an instance of that new type
Trio group = { 1, 2, 3 };                               [3]

// access the "members" of this instance
cout << group.first << ' ' << group.second << ' '       [4]
     << group.third << '\n';

// 1 2 3
```

The real power of the `struct` feature is its ability to aggregate a heterogeneous collection of variables [5]. The programmer can treat this new "compound" type just like the "simple" types previously discussed. Instances of this type can be: put in an array, passed to a function, returned from a function, etc.

```
struct Heterogeneous {
   int    first;                                        [5]
   string second;
   char   third;
};

Heterogeneous bunch = { 42, "some words", 'z' };

cout << bunch.first << "--" << bunch.second << "--"
     << bunch.third << '\n';

// 42--some words--z
```

The following example demonstrates a "system" of cooperating data structure and algorithms. The programmer can create and initialize an instance of the data structure [8], and then pass that data structure to the algorithms as it requests "services" [9].

In the **Functions** section, "pass by value" and "pass by reference" were introduced. The former is what you get by default when you define a parameter to a function [7]. The latter can be requested (by specifying the '&' character) [6]. The former results in a parameter that is effectively "read only". The latter yields a "read/write" parameter (in the `add()` function both `total` and `count` are updated).

```
struct RunningTotal {                    // DATA STRUCTURE
   double total;
   int    count;
};
              [...6...]
void add( RunningTotal& rt, int value ) {     // ALGORITHM
   rt.total = rt.total + value;
   rt.count++; }
                  [...7...]
double average( RunningTotal rt ) {           // ALGORITHM
   return rt.total / rt.count; }

int main() {
   RunningTotal rt = { 0, 0 };                          [8]

   for (int i=1; i < 7; i++)
      add( rt, i );                                     [9]
   cout << "average is " << average( rt ) << '\n';     [9]
}

// average is 3.5
```

Previously, the notion of objects was introduced. Objects (like `word` and `converter` below) can respond to requests (aka function invocations).

```
string  word;
... word.size()
stringstream  converter;
... converter.clear()
```

If we move the `add()` and `average()` functions into the `RunningTotal` definition [10], then they become "siblings" with the data members of the `struct`. These "function members" no longer need to have the `struct` passed to them, since they are now part of the `struct`. Instead – each function is now invoked on the "object" [11].

```
struct RunningTotal {
   double total;
   int    count;

   void add( int value ) {                          [10]
      total = total + value;
      count++; }

   double average() { return total / count; }       [10]
};

int main() {
   RunningTotal rt = { 0, 0 };
```

```
   for (int i=1; i < 7; i++)
      // formerly: add( rt, i );
      rt.add( i );                                          [11]
                // formerly: average(rt)
   cout << "average is " << rt.average() << '\n';         [11]
}
```

The `struct` above is almost a complete class. To take that final step: you change `struct` keyword to `class` [12], add the keyword "`public`" to make the function's accessible [13] (adding "`private`" is optional), change the initialization syntax [15], and add a "constructor" to handle initialization [14]. This is covered in more detail later.

```
class RunningTotal {                                        [12]
  private:
    double total;
    int    count;

  public:                                                   [13]
    RunningTotal( int t, int c ) {                          [14]
      total = t;
      count = c; }
    ...
};

int main() {
   RunningTotal rt( 0, 0 );                                 [15]
   ...
```

Practice. Complete increment 6 of Appendix E.

`vector`. In the **Functions** section, passing arrays as an argument routinely required explicitly passing the size of the array. `vector` (a class in C++'s standard library) is a "smart array", and does not suffer from this limitation. Additionally, it is a nice stepping stone on the path to classes.

But, in order to talk about `vector`, the subject of "templates" must be introduced. Just like functions can be "configured" with an argument list, they can also be configured with the `template` language feature [1]. Inside the angle brackets, function-like parameters [2] may be specified, but primarily you will find the new keyword `typename` [3]. Entries of the latter kind allow place-holders for some future "type" to be specified.

The function `search()` below has been "parameterized" to allow the caller of the function to specify `SIZE` and `TYPE` inside angle brackets at "compile time" [4], and `value` and `array` inside parentheses at "invocation time" [5]. Angle brackets are the syntax for specifying "template parameters" [4], and parentheses are required for "function parameters" [5].

```
 [..1..]   [..2..]   [..3..]
template <int SIZE, typename TYPE>
int search( TYPE value, TYPE array[] ) {
   for (int i=0; i < SIZE; i = i + 1)
      if (value == array[i])
```

```
            return i;
      return -1;
   }

   ...
   string array[] = { "tom", "dick", "harry" };
                         [..4..]      [...5...]
   int position = search<3,string>( "dick", array )
```

Templates allow code (functions or classes) to be written that are independent of one or more types. Instead of having to write many versions of the `search()` function, each seemingly identical; the programmer has written a "template function", and the compiler will "expand" that template into as many versions as programmers desire. Below, the compiler will create a `string` version of `search()` because of the request at [6], and an `int` version because of the call at [7].

```
   string array[] = { "tom", "dick", "harry" };
   cout << "dick is at " << search<3,string>( "dick", array ) << '\n';    [6]
   int nums[] = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };
   cout << "5 is at " << search<9,int>( 5, nums ) << '\n';                [7]

   // dick is at 1
   // 5 is at 4
```

In addition to template functions, C++ supports template classes. `vector` is one of these. When you define an instance of `vector` (aka an object), you specify the "type" of the elements it will hold in angle brackets [9]. Because the class `vector` is not part of the C++ standard language (it is part of its standard library), the two statements at [8] are required in order for the compiler to not generate an error when it encounters the token "vector".

After a `vector` object is created, elements can be inserted by invoking the `push_back()` function [10]. The argument supplied to `push_back()` must be of the type specified in angle brackets at [9]. Any attempt to insert a value of the wrong type will cause a compiler error [11].

```
   #include <vector>                                                      [8]
   using std::vector;                                                     [8]

   int main() {
      vector<int> array;                                                  [9]
      array.push_back( 123 );                                            [10]
      array.push_back( "234" );                                          [11]
      // ERROR:  invalid conversion from 'string' to 'int'
   }
```

`vector` objects are "smart": they can grow to any size desired, and can report their `size()` on demand [12]. As new elements are added, they are appended to the end of the `vector` and its size is incremented [13]. Once one or more elements have been added to a `vector` object, it behaves just like an array. Elements can be accessed or modified by using the array square bracket notation [14].

```
   vector<int> array;
   cout << "initial size is " << array.size() << '\n';                   [12]
   for (int i=10; i > 0; i = i - 1) {
      array.push_back( i );
      cout << array.size() << ' ';                                       [13]
   }
```

```
cout << '\n';
for (int i=0; i < array.size(); i = i + 1)
   cout << array[i] << ' ';                                          [14]
cout << '\n';
}

// initial size is 0
// 1 2 3 4 5 6 7 8 9 10                                             [13]
// 10 9 8 7 6 5 4 3 2 1                                             [14]
```

Unlike arrays, a `vector` object can have its contents assigned to another `vector` object with a single assignment statement [15].

```
vector<int> first, second;
for (int i=1; i < 10; i = i + 1)
   first.push_back( i );

second = first;                                                     [15]

for (int i=0; i < second.size(); i = i + 1)
   cout << second[i] << ' ';
cout << '\n';

// 1 2 3 4 5 6 7 8 9
```

`vector` objects can be compared like variables of "primitive" type (e.g. `int` or `char`). [16] causes the three `vector` objects to have the same contents. The `pop_back()` function removes the last element from a `vector` [17]. The equality expression at [18] succeeds; but the one at [19] fails because the two objects are of different size. Next – the second object's first element is changed [20], and it is no longer equal to the first [21].

```
vector<int> first, second, third;
for (int i=1; i < 10; i = i + 1)
   first.push_back( i );

third = second = first;                                             [16]
third.pop_back();                                                   [17]

cout << "first.size " << first.size() << ", second.size "
     << second.size() << ", third.size " << third.size() << '\n';

if (first == second) cout << "first == second\n";                   [18]
else                 cout << "first != second\n";
if (first == third)  cout << "first == third\n";                    [19]
else                 cout << "first != third\n";

second[0] = 99;                                                     [20]
if (first == second) cout << "first == second\n";                   [21]
else                 cout << "first != second\n";

// first.size 9, second.size 9, third.size 8
// first == second                                                  [18]
// first != third                                                   [19]
// first != second                                                  [21]
```

A limitation of arrays is the requirement to pass the length of the array (along with the array) when a function is invoked. This is not needed when a `vector` object is passed [22]. The object knows its size, and the function can request it [23].

An additional feature of the `vector` class is shown at [24]. A `vector` object can be initialized when it is created by supplying a "sequence". A sequence is essentially an array – but it is "represented" by specifying its "begin" (inclusive) and "end" (exclusive).

```
void print_vector( vector<int> array_object ) {                [22]
   for (int i=0; i < array_object.size(); i = i + 1)           [23]
      cout << array_object[i] << ' ';
   cout << '\n';
}

int main() {
   int array[] = { 2,4,6,8,10,12,14,16,18,20 };
                 // begin    end
   vector<int> vec( array, array+10 );                         [24]
   print_vector( vec );
}

// 2 4 6 8 10 12 14 16 18 20
```

The `vector` class is wonderful – but – it was designed to grow (and shrink) only at the rear. The `deque` class (double-ended queue, rhymes with check) can grow and shrink at the front **and** the rear. As with other classes, the statements at [25] are required for the compiler to understand the token "deque". A deque object can contain any type of element – but only one type per object. The object created at [26] will hold `int` values.

In addition to `push_back()` [28], this class provides `push_front()` [27]. Odd numbers are being inserted at the front, and even numbers at the back. `deque` supports standard array syntax; but you can also retrieve the first element with `front()` [29] and the last element with `back()`. To remove the first element you call `pop_front()` [30], and `pop_back()` will remove the last.

```
...
#include <deque>                                                [25]
using std::deque;                                               [25]
...
deque<int> double_ended;                                        [26]
for (int i=0; i < 10; i = i + 1)
   if (i % 2)
      double_ended.push_front( i );                             [27]
   else
      double_ended.push_back( i );                              [28]
while (double_ended.size()) {
   cout << double_ended.front() << ' ';                         [29]
   double_ended.pop_front();                                    [30]
}

// 9 7 5 3 1 0 2 4 6 8
```

Accessing the elements of an array requires the use of an integer literal or variable [31]. If they could be accessed with a "string" that would be a powerful tool. Implementing a dictionary application would be easy.

The `map` class is a container for "key-value" pairs. The "key" is used for storing and retrieving the "value". The types for the "key" and the "value" are specified inside angle brackets when the `map` object is created [32]. Then values that are of the "key" type are specified inside the array's square brackets [33].

```
string normal_array[5];
normal_array[0] = "first";                                          [31]

map<string,string> map_array;                                       [32]
map_array["first"] = "A sentence for a value.";
cout << "first element is " << map_array["first"] <<'\n';           [33]

// first element is A sentence for a value.
```

Here is a very typical use of `map`. The "key" type is `string` and the "value" type is `int` [37]. The elements of the `input` array are used as the index into the `map_array` object. [38] represents a "look up". If no instance of the "key" exists, then one is created, and it is assigned an "empty value". [39] takes the value returned by [38] and increments it. And [40] writes the arithmetic result back to the original "key".

Listing the elements of a `map` is more complicated than a `vector`. The easiest way to accomplish this is the `for_each()` function [41]. It expects three arguments: the "begin" and "end" of the sequence of elements contained by the `map` object, and the name of a function that will be called for each element. The "signature" of that function should specify a single argument: a "pair" template object that declares the same two types (in angle brackets) as the `map` object itself [35]

`pair` is actually a `struct` with two members [36]: `first` (the "key" for an element), and `second` (the "value" for the same element).

This example counts and reports the number of occurrences of each "word" added to the map object. Like `vector` and `deque`, `map` requires the lines at [34] for the sake of the compiler.

```
...
#include <map>                                                      [34]
using std::map;
using std::pair;

void print( pair<string,int> element ) {                            [35]
   cout << element.first << "--" << element.second << "  ";         [36]
}

int main() {
   map<string,int> map_array;                                       [37]
   string input[] = { "one", "two", "three", "four", "five", "four",
                      "three", "two", "one", "one", "two", "one" };
   for (int i=0; i <12; i = i + 1)
      // [40] write          [38] read            [39] increment
      map_array[ input[i] ] = map_array[ input[i] ] + 1;
```

```
    for_each( map_array.begin(), map_array.end(), print );              [41]
    cout << '\n';
}

// five--1  four--2  one--4  three--2  two--3                           [42]
```

Practice. Increment 7 of Appendix C and increments 7/8/9/10/11 of Appendix E do not depend on anything discussed hereafter. Complete them anytime you desire.

Appendix A ... Grid Game project

Increments

1. draw the board
2. iteratively prompt for input and plot it
3. implement the game with a fixed target
4. implement a randomly assigned target

The "target" is hiding somewhere in the 10 x 10 grid, and the player iteratively inputs a row and a column trying to find it. The first guess below is 4 and 8. The game computes the distance between the target and the guess (7), and plots that value at the row 4 and column 8 grid location. For the second guess, the player "steps off" 7 paces (horizontally and vertically), and inputs row 3 and column 2. The game computes a "delta X plus delta Y" of 4 and plots the answer at row 3 and column 2.

The user then triangulates to produce her third guess: row 6 and column 3 is 7 steps from the first guess and 4 steps from the second guess. When the game displays 0 at that location, the target has been "destroyed".

```
   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -  -  -
2  -  -  -  -  -  -  -  -  -  -
3  -  -  -  -  -  -  -  -  -  -
4  -  -  -  -  -  -  -  -  -  -
5  -  -  -  -  -  -  -  -  -  -
6  -  -  -  -  -  -  -  -  -  -
7  -  -  -  -  -  -  -  -  -  -
8  -  -  -  -  -  -  -  -  -  -
9  -  -  -  -  -  -  -  -  -  -
row col ... 4 8

   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -  -  -
2  -  -  -  -  -  -  -  -  -  -
3  -  -  -  -  -  -  -  -  -  -
4  -  -  -  -  -  -  -  -  7  -
5  -  -  -  -  -  -  -  -  -  -
6  -  -  -  -  -  -  -  -  -  -
7  -  -  -  -  -  -  -  -  -  -
8  -  -  -  -  -  -  -  -  -  -
9  -  -  -  -  -  -  -  -  -  -
row col ... 3 2

   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -  -  -
2  -  -  -  -  -  -  -  -  -  -
3  -  -  4  -  -  -  -  -  -  -
4  -  -  -  -  -  -  -  -  7  -
```

```
5  -  -  -  -  -  -  -  -  -  -
6  -  -  -  -  -  -  -  -  -  -
7  -  -  -  -  -  -  -  -  -  -
8  -  -  -  -  -  -  -  -  -  -
9  -  -  -  -  -  -  -  -  -  -
row col ... 6 3

   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -  -  -
2  -  -  -  -  -  -  -  -  -  -
3  -  -  4  -  -  -  -  -  -  -
4  -  -  -  -  -  -  -  -  7  -
5  -  -  -  -  -  -  -  -  -  -
6  -  -  -  0  -  -  -  -  -  -
7  -  -  -  -  -  -  -  -  -  -
8  -  -  -  -  -  -  -  -  -  -
```

Increment 1. Use two or more loops to output the board, and then exit.

```
   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -  -  -
2  -  -  -  -  -  -  -  -  -  -
3  -  -  -  -  -  -  -  -  -  -
4  -  -  -  -  -  -  -  -  -  -
5  -  -  -  -  -  -  -  -  -  -
6  -  -  -  -  -  -  -  -  -  -
7  -  -  -  -  -  -  -  -  -  -
8  -  -  -  -  -  -  -  -  -  -
9  -  -  -  -  -  -  -  -  -  -
```

Consider attacking this in stages: get the column labels working, then the row labels, then the dash characters in the "body".

Increment 2. In order to redraw the board and all previous guesses after each new guess, all guesses must be "remembered". You could choose to maintain a list of all guesses (and the answer received), and then compare each guess with each grid location as the board is redrawn. But that seems like an enormous amount of work.

What about maintaining a two-dimensional array that holds the "state" of each grid location? The type of that array would need to be able to hold dash characters – and – integer values.

In this increment, iteratively prompt for: a row, a column, and a value; and plot the value at the specified row and column. Notice that the value can be two digits, and it is right justified.

```
   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -  -  -
2  -  -  -  -  -  -  -  -  -  -
3  -  -  -  -  -  -  -  -  -  -
4  -  -  -  -  -  -  -  -  -  -
5  -  -  -  -  -  -  -  -  -  -
6  -  -  -  -  -  -  -  -  -  -
7  -  -  -  -  -  -  -  -  -  -
8  -  -  -  -  -  -  -  -  -  -
```

```
9  -  -  -  -  -  -  -  -  -  -
row col val ... 2 1 21

   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -  -  -
2  - 21  -  -  -  -  -  -  -  -
3  -  -  -  -  -  -  -  -  -  -
4  -  -  -  -  -  -  -  -  -  -
5  -  -  -  -  -  -  -  -  -  -
6  -  -  -  -  -  -  -  -  -  -
7  -  -  -  -  -  -  -  -  -  -
8  -  -  -  -  -  -  -  -  -  -
9  -  -  -  -  -  -  -  -  -  -
row col val ... 7 8 78

   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -  -  -
2  - 21  -  -  -  -  -  -  -  -
3  -  -  -  -  -  -  -  -  -  -
4  -  -  -  -  -  -  -  -  -  -
5  -  -  -  -  -  -  -  -  -  -
6  -  -  -  -  -  -  -  -  -  -
7  -  -  -  -  -  -  -  - 78  -
8  -  -  -  -  -  -  -  -  -  -
9  -  -  -  -  -  -  -  -  -  -
row col val ...
```

Increment 3. Add a "hard-wired" target position (i.e. the target row and column will be the same every time the game is run). Remove the "value" prompt and input. Instead – compute and plot "delta X plus delta Y".

```
   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -  -  -
2  -  -  -  -  -  -  -  -  -  -
3  -  -  -  -  -  -  -  -  -  -
4  -  -  -  -  -  -  -  -  -  -
5  -  -  -  -  -  -  -  -  -  -
6  -  -  -  -  -  -  -  -  -  -
7  -  -  -  -  -  -  -  -  -  -
8  -  -  -  -  -  -  -  -  -  -
9  -  -  -  -  -  -  -  -  -  -
row col ... 3 5

   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -  -  -
2  -  -  -  -  -  -  -  -  -  -
3  -  -  -  -  -  3  -  -  -  -
4  -  -  -  -  -  -  -  -  -  -
5  -  -  -  -  -  -  -  -  -  -
6  -  -  -  -  -  -  -  -  -  -
7  -  -  -  -  -  -  -  -  -  -
```

```
8  -  -  -  -  -  -  -  -  -  -
9  -  -  -  -  -  -  -  -  -  -
row col ... 5 4

   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -  -  -
2  -  -  -  -  -  -  -  -  -  -
3  -  -  -  -  -  3  -  -  -  -
4  -  -  -  -  -  -  -  -  -  -
5  -  -  -  -  4  -  -  -  -  -
6  -  -  -  -  -  -  -  -  -  -
7  -  -  -  -  -  -  -  -  -  -
8  -  -  -  -  -  -  -  -  -  -
9  -  -  -  -  -  -  -  -  -  -
row col ... 4 7

   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -  -  -
2  -  -  -  -  -  -  -  -  -  -
3  -  -  -  -  -  3  -  -  -  -
4  -  -  -  -  -  -  -  0  -  -
5  -  -  -  -  4  -  -  -  -  -
6  -  -  -  -  -  -  -  -  -  -
7  -  -  -  -  -  -  -  -  -  -
8  -  -  -  -  -  -  -  -  -  -
9  -  -  -  -  -  -  -  -  -  -
```

Increment 4. Use the `rand()` function to compute a random target position.

Appendix AA ... Grid Game implementations

Increment 1. Printing the column labels is a simple exercise in iteration.

```
      cout << ' ';
      for (int i=0; i < 10; i = i + 1)
         cout << "  " << i;

      //    0  1  2  3  4  5  6  7  8  9
```

```
Printing the row labels is an additional simple exercise in iteration [0].
```

```
      cout << ' ';
      for (int i=0; i < 10; i = i + 1)
         cout << "  " << i;
      cout << '\n';
      for (int i=0; i < 10; i = i + 1)       [0]
         cout << i << '\n';

      //    0  1  2  3  4  5  6  7  8  9
      // 0
      // 1
      // 2
      // ...
```

```
Adding the "body" to this grid is accomplished by adding an inner loop [2] to
the second loop [1].
```

```
      cout << ' ';
      for (int i=0; i < 10; i = i + 1)
         cout << "  " << i;
      cout << '\n';
      for (int i=0; i < 10; i = i + 1) {     [1]
         cout << i;
         for (int j=0; j < 10; j = j + 1)    [2]
            cout << "  -";
         cout << '\n';
      }

      //    0  1  2  3  4  5  6  7  8  9
      // 0  -  -  -  -  -  -  -  -  -  -
      // 1  -  -  -  -  -  -  -  -  -  -
      // 2  -  -  -  -  -  -  -  -  -  -
      // ...
```

Increment 2. The goal here is to iteratively: draw the board, prompt for row-column-value, receive that input from the player, and repeat.

```
         0  1  2  3  4  5  6  7  8  9
      0  -  -  -  -  -  -  -  -  -  -
      1  -  -  -  -  -  -  -  -  -  -
      2  -  -  -  -  -  -  -  -  -  -
      ...
      row col val ... 2 1 21
```

```
   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -  -  -
2  - 21  -  -  -  -  -  -  -  -
...
row col val ...
```

The playing surface (or board) resembles a matrix. Therefore, modeling it with a two-dimensional array is a natural fit. This choice allows us to: assign an initial value to every position [3], modify any position specified by the player [4], and retrieve the value of each position in order to draw the board [5]. Since the contents of the matrix are heterogeneous (characters and numbers), let's declare the 2D array to be of type `string` [6]. [`string` is the "lowest common denominator" between all other data types.] The variable that receives the "value" input is also of type `string` [7].

Because the "value" input by the player can be one or two characters wide, displaying them right-justified is desirable. In the early `cout` discussion, right-justification was demonstrated using the `setw()` function; and has been exercised in this increment [8].

```
string grid[10][10];                          [6]
for (int i=0; i < 10; ++i)
   for (int j=0; j < 10; ++j)
      grid[i][j] = "-";                       [3]

int row, col;
string val;                                   [7]
while (1) {
   cout << "\n ";
   for (int i=0; i < 10; ++i)
      cout << setw(3) << i;                   [8]
   cout << '\n';
   for (int i=0; i < 10; ++i) {
      cout << i;
      for (int j=0; j < 10; ++j)
         cout << setw(3) << grid[i][j];       [5]
      cout << '\n';
   }
   cout << "row col val ... ";
   cin >> row >> col >> val;
   grid[row][col] = val;                      [4]
}
```

Increment 3. This increment introduces the notion of a target position and wrestles with the algorithm for computing "delta X plus delta Y". The "value" user input has been removed, and replaced with an integer computed by the application.

```
   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
1  -  -  -  -  -  -  -  -  -  -
2  -  -  -  -  -  -  -  -  -  -
...
row col ... 2 1

   0  1  2  3  4  5  6  7  8  9
0  -  -  -  -  -  -  -  -  -  -
```

```
1  -  -  -  -  -  -  -  -  -  -
2  -  8  -  -  -  -  -  -  -  -
...
row col val ...
```

The next increment will pick a random target location; but the target location is constant in this increment [9]. The algorithm for computing the response to player input is at [10] (development of that algorithm follows). Converting that value from integer to string is at [10] and [11].

```
string grid[10][10];
... (initialize grid)

int row, col, target_row = 4, target_col = 7;                [9]
while (1) {
   ... (draw the board)
   cout << "row col ... ";
   cin >> row >> col;
   stringstream ss;
   ss << abs(target_row - row) + abs(target_col - col);     [10]
   grid[row][col] = ss.str();                               [11]
}
```

Every location on the board is identified by a (row,column) pair. What would happen in the following design if the `target_col` is greater than the `guess_col`?

```
int guess_row, guess_col, target_row = 4, target_col = 7;
int delta_x   = guess_col - target_col;
int delta_y   = guess_row - target_row;
int delta_x_y = delta_x + delta_y;
```

Do you need to accommodate this possibility with code like the following?

```
int delta_x;
if (guess_col > target_col
   delta_x = guess_col - target_col;
else
   delta_x = target_col - guess_col;
```

What about this strategy?

```
int delta_x = guess_col - target_col;
if (delta_x < 0)
   delta_x = delta_x * -1;
```

The field of mathematics defines a function called "absolute value" that represents this logic. Computer languages routinely provide this functionality, and it is called `abs()` in C++.

```
delta_x_y = abs(target_col - guess_col) + abs(target_row - guess_row);
```

Increment 4. Picking random numbers is the domain of the `rand()` function. It was introduced earlier. The integer produced by rand() can be very large, but it can be scaled to the range 0 to 9 by applying the the modulus operator [12]. Everything else remains the same.

```
string grid[10][10];
... (initialize grid)

srand( time( 0 ) );
int target_row = rand() % 10;             [12]
```

```
int target_col = rand() % 10;
int row, col;
while (1) {
   ... (draw the board)
   cout << "row col ... ";
   cin >> row >> col;
   stringstream ss;
   ss << abs(target_row - row) + abs(target_col - col);
   grid[row][col] = ss.str();
}
```

Appendix B ... Fifteen Puzzle project

Increments

1. draw the board
2. draw the board with loops
3. add an array to maintain a mapping of location and number
4. add cin, search, update
5. draw_board(), move(), swap()
6. mix up the numbers

The goal is to: implement a "board" consisting of the numbers 1 through 15, and a space (an empty position), mix up the numbers, and respond to requests from the player to move numbers (swap the number specified with the adjacent space).

```
     6  4 14
  2  1  7  8
  9  3 13 12
 11  5 10 15

Number: 6

  6     4 14
  2  1  7  8
  9  3 13 12
 11  5 10 15

Number: 1

  6  1  4 14
  2     7  8
  9  3 13 12
 11  5 10 15

Number: 2

  6  1  4 14
     2  7  8
  9  3 13 12
 11  5 10 15
```

Increment 1. Print the numbers 1 through 16 in a 4 x 4 grid using one or more loops. Right justify the numbers.

```
 1  2  3  4
 5  6  7  8
 9 10 11 12
13 14 15 16
```

Increment 2. As the project evolves, we will need to maintain a mapping between positions and numbers. The former are fixed: positions 0 through 3 are the first row, 4 through 7 are the second row, etc. The latter are dynamic – they move around as the player interacts.

```
In this increment, put the numbers in an array, and print the elements of that
array.
```

Increment 3. Add an infinite loop that: displays the board, prompts for a number, finds the specified number and replaces it with a space. These are all pieces that will be needed in the final application. Consider using the number '0' to represent the concept of "space".

```
  1  2  3  4
  5  6  7  8
  9 10 11 12
 13 14 15 16

Number: 2

  1     3  4
  5  6  7  8
  9 10 11 12
 13 14 15 16

Number: 13

  1     3  4
  5  6  7  8
  9 10 11 12
    14 15 16

Number: 11

  1     3  4
  5  6  7  8
  9 10    12
    14 15 16
```

Increment 4. Move the "draw board" functionality to a `draw_board()` function. Replace the number 16 with a space. When the user specifies a number attempt to "move" it (if the space is adjacent to that number, then swap the two).

How much effort is necessary to evaluate this adjacency criteria? The number 1 has 2 neighbors, number 2 has 3 neighbors, and number 6 has 4 neighbors. Do all these numbers have to be treated special – or – is there some "secret family recipe" that yields a solution in a fraction of the code?

```
  1  2  3  4
  5  6  7  8
  9 10 11 12
 13 14 15

Number: 15

  1  2  3  4
  5  6  7  8
  9 10 11 12
 13 14    15

Number: 11
```

```
 1  2  3  4
 5  6  7  8
 9 10    12
13 14 11 15

Number: 10

 1  2  3  4
 5  6  7  8
 9    10 12
13 14 11 15
```

Increment 5. In the real puzzle, the numbers must first be mixed up. Design and implement a "randomizing" algorithm.

Would it make sense to randomly place the numbers on the board?

What about swapping pairs of random number?

How are the numbers randomized in the real puzzle?

Is it possible to put the puzzle in a state that is impossible to solve?

Appendix BB ... Fifteen Puzzle implementations

Increments

1. draw the board
2. draw the board with loops
3. add an array to maintain a mapping of location and number
4. add cin, search, update
5. draw_board(), move(), swap()
6. mix up the numbers

Increment 1. Each line is output with its own `cout` statement. Notice the "\n" carriage returns embedded in each string literal.

```
cout << " 1  2  3  4\n";
cout << " 5  6  7  8\n";
cout << " 9 10 11 12\n";
cout << "13 14 15 16\n";

//  1  2  3  4
//  5  6  7  8
//  9 10 11 12
// 13 14 15 16
```

Increment 2. How about using an outer loop to control the output of each row [1], and an inner loop to output each "column" [2]?

```
int number = 1;
for (int i=0; i < 4; i = i + 1) {            [1]
   for (int j=0; j < 4; j = j + 1) {         [2]
      cout << setw(3) << number;
      number = number + 1;
   }
   cout << '\n';
}

//  1  2  3  4
//  5  6  7  8
//  9 10 11 12
  // 13 14 15 16
```

I don't give a damn for a man that can only spell a word one way. [Mark Twain]

Let's consider two alternative "spellings". Instead of using a separate variable to count and output each number, you could compute the number from the two loop control variables [3].

```
for (int i=0; i < 4; i = i + 1) {
   for (int j=0; j < 4; j = j + 1)
      cout << setw(3) << i * 4 + j + 1;      [3]
   cout << '\n';
}
```

Or – a single loop will work, if you embed some logic to decide when it's time to start a new row [4]. When "i" equals 4, 8, or 12, then the modulo (i.e. "remainder") operator returns 0.

```
for (int i=1; i <= 16; i = i + 1) {
   cout << setw(3) << i;
   if (i % 4 == 0)                          [4]
      cout << '\n';
}
```

Increment 3. In anticipation of dealing with stationary positions and moving numbers, this increment adds an array to provide the indirection that can manage this mapping. The loop index variable is no longer output; it is used to access each element of the array [5].

```
int numbers[] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 };
for (int i=0; i < 16; i = i + 1) {
   cout << setw(3) << numbers[i];           [5]
   if ((i+1) % 4 == 0)
      cout << '\n';
}
```

Increment 4. The goal here is it iteratively: display the board [6], receive a number from the player [7], and convert the specified number [8] to a space. The convention chosen for representing a space is the number 0. It is assigned at [8], and then used at [9]. A simple sequential search is used to identify the location of the specified number [10].

```
int numbers[] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16 };
int number;
while (true) {
   // Draw the board                        [6]
   cout << '\n';
   for (int i=0; i < 16; i = i + 1) {
      if (numbers[i] == 0)                  [9]
         cout << "   ";
      else
         cout << setw(3) << numbers[i];
      if ((i+1) % 4 == 0)
         cout << '\n';
   }
   // Prompt for user input
   cout << "\nNumber: ";
   cin >> number;                           [7]
   // Search for the specified number
   for (int i=0; i < 16; i = i + 1)        [10]
      if (numbers[i] == number) {
         numbers[i] = 0;                    [8]
         break;
}     }
```

Increment 5. The intent of this increment is to implement the entire game, except for the randomizing of the numbers. A `draw_board()` abstraction has been introduced [11], and everything associated with a "move"has been encapsulated [12].

```
int main() {
   int numbers[] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0 };
   int number;
```

```
      while (true) {
         draw_board( numbers );                [11]
         cin >> number;
         move( numbers, number );              [12]
   }  }
```

`draw_board()` packages implementation that was previously "inline".

```
   void draw_board( int numbers[] ) {
      cout << '\n';
      for (int i=0; i < 16; i = i + 1) {
         if (numbers[i] == 0)
            cout << "   ";
         else
            cout << setw(3) << numbers[i];
         if ((i+1) % 4 == 0)
            cout << '\n';
      }
      cout << "\nNumber: ";
   }
```

How should `move()` be implemented? Consider the geometry and the numbering convention of our 16-position grid.

```
 0  1  2  3
 4  5  6  7
 8  9 10 11
12 13 14 15
```

What about searching for the position of the number specified, and then testing all its neighbors for the location of the space.

```
   void move( int numbers[], int number ) {
      // Search for the number's position
      int i;
      for (i=0; i < 16; i = i + 1)
         if (numbers[i] == number)
            break;
      // If the number was not found - return
      if (i == 16)
         return;
      // Test the 2 neighbors of position 15
      if (i == 15 && numbers[11] == 0)   // neighbor above
         swap( numbers, i, 11 );
      if (i == 15 && numbers[14] == 0)   // neighbor to the left
         swap( numbers, i, 14 );
      // Test the 3 neighbors of position 14
      if (i == 14 && numbers[10] == 0)   // neighbor above
         swap( numbers, i, 10 );
      if (i == 14 && numbers[13] == 0)   // neighbor to the left
         swap( numbers, i, 13 );
      if (i == 14 && numbers[15] == 0)   // neighbor to the right
         swap( numbers, i, 15 );
      ...
```

This choice seems like an enormously bloated approach.

Looking at position 6, its neighbor above is 4 less, its neighbor below is 4 greater, its neighbor to the left is 1 less, and its neighbor to the right is 1 greater. Does that pattern always hold?

```
 0  1  2  3
 4  5  6  7
 8  9 10 11
12 13 14 15
```

Is position 8 to the right of position 7? Are positions 11 and 12 adjacent? Do the positions on the "edge" present nothing but exceptions, or are there some rules that can be exercised?

What about testing for a problematic boundary condition and then applying a general rule?

```
// "i" is the position of the number specified by the player
// If "i" is not on the top row – and – the space is above "i",
//    then swap the contents of the two positions
if (i is not on the top row && numbers[i-4] == 0)
   swap( numbers, i, i-4 );
// If "i" is not on the top row – and – the space is below "i"
if (i is not on the bottom row && numbers[i+4] == 0)
   swap( numbers, i, i+4 );
// If "i" is not in the left column – and – the space is left of "i"
if (i is not in the left column && numbers[i-1] == 0)
   swap( numbers, i, i-1 );
// If "i" is not in the right column – and – the space is right of "i"
if (i is not in the right column && numbers[i+1] == 0)
   swap( numbers, i, i+1 );
```

Is there some "general" way to test for these boundary conditions (i.e. top row, bottom row, left column, right column)? Consider the following example of integer division and modulus division.

```
for (int i=0; i < 16; i = i + 1) {
   cout << (i / 4) << ' ';
   //cout << (i % 4) << ' ';
   if ((i+1) % 4 == 0)
      cout << '\n';
}

//     i / 4       i       i % 4
//    0 0 0 0     0-3     0 1 2 3
//    1 1 1 1     4-7     0 1 2 3
//    2 2 2 2     8-11    0 1 2 3
//    3 3 3 3    12-15    0 1 2 3
```

It looks like "`i / 4 == 0`" can tell us if "i" is on the top row. And "`i % 4 == 0`" can tell us if "i" is in the left column. Given our accumulated insight, the following implementation is suggested.

```
void move( int numbers[], int number ) {
   int i;
   for (i=0; i < 16; i = i + 1)
      if (numbers[i] == number)
         break;
   if (i == 16)
      return;
   // if i is not on the top row AND the space is above
   if      (i / 4 > 0  &&  numbers[i-4] == 0)
```

```
      swap( numbers, i, i-4 );
   // if i is not on the bottom row AND the space is below
   else if (i / 4 < 3  &&  numbers[i+4] == 0)
      swap( numbers, i, i+4 );
   // if i is not in the left column AND the space is to the left
   else if (i % 4 > 0  &&  numbers[i-1] == 0)
      swap( numbers, i, i-1 );
   // if i is not in the right column AND the space is to the right
   else if (i % 4 < 3  &&  numbers[i+1] == 0)
      swap( numbers, i, i+1 );
}
```

The final piece is a `swap()` function. You can see a "diagonal" quality to the assignments below. First, save the value at position_one. Second, move the value from position_two to position_one. Finally, move the saved value to position_two.

```
void swap( int numbers[], int position_one, int position_two ) {
   int temp             = numbers[position_one];
   numbers[position_one] = numbers[position_two];
   numbers[position_two] = temp;
}
```

Increment 6. When I first implemented this project, I created a randomizing algorithm that was a work of art. The testing proceeded without a hitch. Later, I showed the puzzle to my daughter. After working for several minutes, she insisted the puzzle could not be solved. I tried to finish her game and was forced to agree with her.

Searching the Internet, I found the puzzle was invented by a traveling showman in the 1800s. He would start with the numbers in ascending order, manually switch the 14 and the 15, slide all the numbers until they were sufficiently mixed up, and then wager that no one in the community could slide the numbers back into ascending order.

I decided that my randomizing algorithm effectively reproduced the "switched 14 and 15" configuration on occasion.

The point of this story is to suggest that the only reasonable way to randomize the numbers is to perform dozens (or hundreds) of moves. Do you want to focus on strictly "legal" moves (i.e. numbers that are adjacent to the space)? Or – do you want to just generate random numbers in the range 1 to 16, and pass each to `move()`?

I chose the latter option ...

```
int main() {
   int numbers[] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,0 };
   srand( time( 0 ) );
   for (int i=0; i < 600; i = i + 1)
      move( numbers, rand() % 15 + 1 );

   int number;
   while (true) {
      draw_board( numbers );
      cin >> number;
      move( numbers, number );
}  }
```

```
//  14  9 11  8
//  10  1  3  6
//   5 12  4
//   7 13 15  2
//
// Number:
```

Appendix C ... Mastermind project

Increments

1. prompt, receive input, report input
2. prompt, receive input, report input in a loop
3. replace 4 variables with an array of 4 elements
4. convert 4 integer inputs to 1 string input
5. add hard-wired answer array and compute "totally correct" response
6. populate answer array with random values
7. compute "partially correct" response

The Mastermind game starts with the selection of four random numbers in the range 1 to 6, and the player tries to guess the selected numbers. After each guess, the game provides two numbers: the count of guess numbers that are totally correct (correct number in the correct position), and the count of guess numbers that are partially correct (correct number in the wrong position).

Below, the player typed "1 1 4 4". The program responded "0 0". The first number represents "totally correct", and the second number is "partially correct". The second response says there are 3 numbers that are correct – but – in the wrong position. After rearranging three of those numbers, and adding a fourth; the response seems to suggest a step backward: two of the numbers are "totally correct". The player assumed the middle two numbers were correct, and added two other numbers that were consistent with all previous feedback. The fourth response says: the four numbers have been identified, but two need to be swapped. The fifth guess solved the puzzle.

```
Input: 1 1 4 4
                    0  0
Input: 2 2 3 3
                    0  3
Input: 3 3 2 5
                    2  0
Input: 6 3 2 2
                    2  2
Input: 3 6 2 2
                    4  0
```

Increment 1. Start with really basic functionality: prompt for input, accept four strings, report the strings back to the player, and exit.

```
Input: 4 x 2 one
                    4 x 2 one
```

Increment 2. Replace the `string` variables with `int` variables. Add a loop and repeat indefinitely. Notice what happens when too many numbers are entered, and then too few numbers.

```
Input: 4 3 2 1
                    4 3 2 1
Input: 1 2 3 4 5
                    1 2 3 4
Input: 6 7 8
                    5 6 7 8
```

Increment 3. Replace the four integer variables with an integer array of four elements. Not much work in this increment – don't make too much of it.

Increment 4. In the final product, we want to allow the player to enter "answer" and have the selected numbers output. To that end, use `getline()` to capture player input in a `string` variable, and test if "quit" is input. If not, use the `string` variable to initialize a `stringstream` object that can then be used to "tokenize"the player input and initialize the legacy integer array.

```
Input: 1 2 3 4
                    1 2 3 4
Input: 5 6 7
                    5 6 7 4
Input: 8 9 1 2 3
                    8 9 1 2
Input: quit
```

Increment 5. Add a hard-wired "answer" array. Evaluate player input against the answer array and report the count of "correct number in the correct position". When the count equals 4 – exit. In addition to the "quit" command, add an "answer" command that allows the player to request that the answer array be displayed.

```
int answer[] = { 2, 6, 4, 1 };

// Input: 1 3 2 4
//                    0
// Input: answer
//                    2 6 4 1
// Input: 2 6 1 4
//                    2
// Input: 2 6 4 1
//                    4
```

Increment 6. Populate the answer array with random numbers.

Increment 7. Complete the Mastermind project by computing the "correct number in the wrong position", and displaying it after the "correct number in the correct position" count.

```
// Input: 1 1 2 2
//                    1  1
// Input: 3 3 4 4
//                    0  1
// Input: 1 4 1 5
//                    1  1
// Input: answer
//                    4 2 2 5
// Input: 2 4 2 5
//                    2  2
// Input: 4 2 2 5
//                    4  0
```

When a number in the answer contributes to the "totally correct" count, it should not also participate in the "partially correct" count. When a number in the guess is matched to a number in the answer, both numbers should be set aside so that they are not matched to any other number. The sum of "totally correct" plus "partially correct" can never be more than four.

Should the "partially correct" count be computed before or after the "totally correct" count?

Is the computation of each independent of the other, or are they dependent?

Are there data structures that offer leverage to the algorithm?

Appendix CC ... Mastermind implementations

Increments

1. prompt, receive input, report input
2. prompt, receive input, report input in a loop
3. replace 4 variables with an array of 4 elements
4. convert 4 integer inputs to 1 string input
5. add hard-wired answer array and compute "totally correct" response
6. populate answer array with random values
7. compute "partially correct" response

Increment 1. The goal of this increment is very limited: iteratively [1] prompt for input [2], receive four integers [3], and report the user's input [4]. Notice that the user supplied five integers at [5], but only the first four seem to have been returned by `cin`. The fifth value is actually returned the next time `cin` is called [6]. This demonstrates that `cin` stores up user input, and then dispenses that input to the number of variables supplied by the programmer.

```
string  first, second, third, fourth;
cout << "Input: ";                                [2]
cin >> first >> second >> third >> fourth;    [3]
cout << "                  ";
cout << first << ' ' << second << ' ' << third << ' ' << fourth << '\n';

// Input: 4 x 2 one
//                  4 x 2 one
```

Increment 2. The goal of this increment is very limited: iteratively [1] prompt for input [2], receive four integers [3], and report the user's input [4]. Notice that the user supplied five integers at [5], but only the first four seem to have been returned by `cin`. The fifth value is actually returned the next time `cin` is called [6]. This demonstrates that `cin` stores up user input, and then dispenses that input to the number of variables supplied by the programmer.

```
int first, second, third, fourth;
while (true) {                                        [1]
   cout << "Input: ";                                 [2]
   cin >> first >> second >> third >> fourth;    [3]
   cout << "                    ";
   cout << first << ' ' << second << ' '          [4]
   cout << third << ' ' << fourth << '\n';
}

// Input: 4 3 2 1
//                  4 3 2 1
// Input: 1 2 3 4 5                                   [5]
//                  1 2 3 4
// Input: 6 7 8
//                  5 6 7 8                           [6]
```

Increment 3. Instead of juggling four autonomous integers, this increment replaces them with an integer array composed of four elements [7]. An array lends itself to the leverage of loops. This is

seen at as input is received [8] and reported [9]. The laborious enumeration of four distinct variables has been replaced by the shorthand of repetition.

```
int input[4];                                    [7]
while (true) {
   cout << "Input: ";
   for (int i=0; i < 4; i = i + 1)               [8]
      cin >> input[i];
   cout << "                    ";
   for (int i=0; i < 4; i = i + 1)               [9]
      cout << input[i] << ' ';
   cout << '\n';
}
```

Increment 4. This increment reworks the scheme for user input. In addition to integers, we would the user to be able to input commands like "quit" and "answer". To this end, let's accept input as a string [10]. Since we don't know if the user will enter four integers or one word, we need to "read" an entire line of input with `getline()` [11].

If "quit" is entered [12], then we break out of the while loop and exit the program. Otherwise, the string received from the user is loaded into a `stringstream` object [13], and that object is used to break the string into individual integers [14].

```
string input_string;                             [10]
int input[4];
while (true) {
   cout << "Input: ";
   getline( cin, input_string );                 [11]
   if (input_string == "quit")                   [12]
      break;
   stringstream ss;
   ss << input_string;                           [13]
   for (int i=0; i < 4; i = i + 1)
      ss >> input[i];                            [14]
   ...
}
```

Increment 5. We are ready to add the beginning of the actual puzzle. A hard-wired "answer" array has been introduced at [15]. The user can now ask for the answer to be displayed [16].

The most interesting feature of this increment is designing an algorithm for computing the count of "correct numbers in the correct position". How hard (or simple) is this challenge?

How about this approach – step through the input array and the answer array, compare each element of one to its counterpart in the other, and increment a `num_correct` variable when they are found to be equal [17].

```
string input_string;
int input[4];
int answer[] = { 2, 6, 4, 1 };                   [15]
int num_correct = 0;
while (true) {
   cout << "Input: ";
   getline( cin, input_string );
   if (input_string == "quit")
```

```
      break;
   if (input_string == "answer") {                [16]
      cout << "                    ";
      for (int i=0; i < 4; i = i + 1)
         cout << answer[i] << ' ';
      cout << '\n';
      continue;
   }
   stringstream ss;
   ss << input_string;
   for (int i=0; i < 4; i = i + 1)
      ss >> input[i];
   for (int i=0; i < 4; i = i + 1)
      if (input[i] == answer[i])                  [17]
         num_correct = num_correct + 1;
   cout << "                   " << num_correct << '\n';
   if (num_correct == 4)
      break;
   num_correct = 0;
}
```

Increment 6. To replace the hard-wired array with a random array, we can use the same functions as we did with the Grid Game and the Fifteen Puzzle. `srand()` [18] seeds the random number generator, and `time()` gives `srand()` a different starting point each time the application is run. rand() [19] returns a random number of unknown size, and the % operator expression scales the number to the range 0 through 5.

```
string input_string;
int input[4];
int answer[4];
srand( time( 0 ) );                              [18]
for (int i=0; i < 4; i = i + 1)
   answer[i] = rand() % 6 + 1;                   [19]
...
```

Increment 7. The final piece is to compute the count of "correct numbers in the wrong position". Let's encapsulate the computation of both values (totally correct and partially correct) in a new function `evaluate_guess()`. It will need the answer array and the input array to do its work [20]. Since it must return two values, the return value of the function won't be sufficient. We could use the return value for one of the numbers, and a function argument for the other. But – it would be more symmetric to use a function argument for each [21].

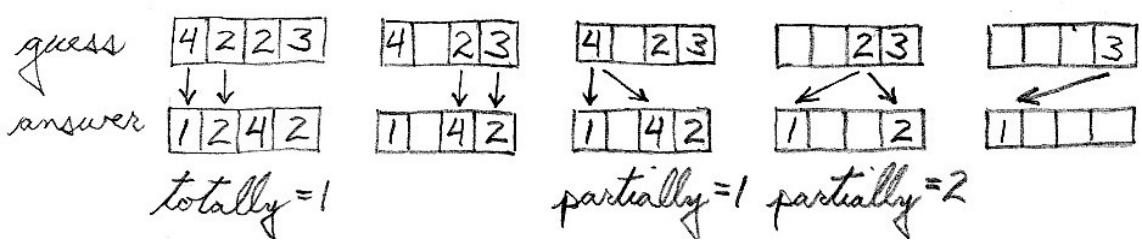
```
...
int num_correct, num_partial_correct;
while (true) {
   ...
   evaluate_guess( answer, guess,                          [20]
                   num_correct, num_partial_correct );     [21]
   cout << "                   " << num_correct
        << "  " << num_partial_correct << '\n';
   if (num_correct == 4)
      break;
}
```

The first thing to notice about `evaluate_guess()` is that the `totally_correct` and `partially_correct` arguments are being passed by reference (the '&' syntax after the type of the argument) [22]. This means the function can assign a value to each argument, and expect that value to be returned to the caller of the function.

The algorithm `evaluate_guess()` operates on the foundation of the two data structures at [23]. As the "totally" and "partially" counts are computed, these two arrays maintain the state of the evaluation of the guess versus the answer.

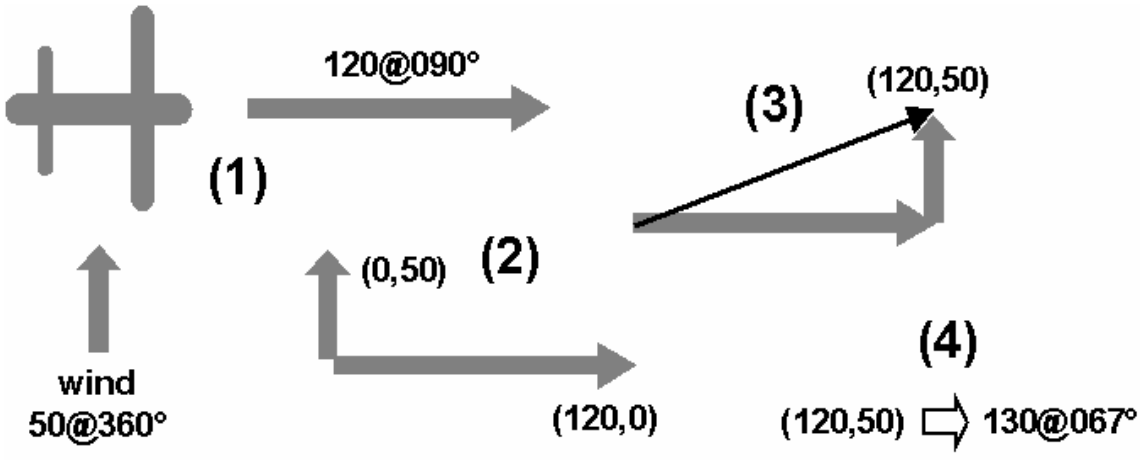
```
void evaluate_guess( int answer[], int guess[],
                     int& totally_correct,                  [22]
                     int& partially_correct ) {
   bool guess_used[4]  = { false, false, false, false };    [23]
   bool answer_used[4] = { false, false, false, false };
   totally_correct = partially_correct = 0;
   // Compute the totally correct value (from the previous increment)
   for (int i=0; i < 4; i = i + 1)
      if (guess[i] == answer[i]) {                          [24]
         totally_correct = totally_correct + 1;
         guess_used[i] = answer_used[i] = true;
      }
   // Compute the partially correct value                   [25]
   for (int i=0; i < 4; i = i + 1) {       // step thru the guess
      if (guess_used[i])
         continue;
      for (int j=0; j < 4; j = j + 1) {    // step thru the answer
         // If this slot in answer has been previously used, skip it
         if (answer_used[j])
            continue;
         if (guess[i] == answer[j]) {
            partially_correct = partially_correct + 1;
            guess_used[i] = answer_used[j] = true;
            // Now that this guess slot has been "bound" to an answer
            // slot, don't try to match it to another answer slot
            break;
}  }  }  }
```

The code at [24] above came from the previous increment, and is modeled in the first frame below. When the first '2' of the guess matches the first '2' of the answer, the `totally_correct` count is incremented, and both '2's are removed from future consideration. The code at [25] is new, and is modeled by the next four frames below. Each remaining number in the guess is compared to those that remain in the answer. In the third frame below: '4' matches '4', the `partially_correct` count is incremented, and the '4's are removed. The fourth frame does the same with '2's. The final frame does not produce a match.



This algorithm is fairly straight-forward (in hind-sight). Is it the simplest algorithm possible? Is it the only algorithm possible?

Oftentimes, transforming a problem from its default "domain" to a different domain makes the complexity go away. Consider rectangular coordinates and polar coordinates. Addition and subtraction are easy in the former; multiplication and division are easy in the latter.



What if we moved from a "position" focus to a "value summary" focus. Both the guess and the answer could be refactored as a count of the digits from '1' to '6'. The two new arrays could then be combined by performing a "min" operation. The effect of this manipulation could be described as: if the guess has one '2' and the answer has two '2's (or vice versa) then the overlap is one and that represents the number of totally correct and partially correct '2's.

```
guess: 4 2 2 3   answer: 1 2 4 2   min(guess,answer)
    1s  0            1s  1             1s  0
    2s  2            2s  2             2s  2
    3s  1            3s  0             3s  0
    4s  1            4s  1             4s  1
    5s  0            5s  0             5s  0
    6s  0            6s  0             6s  0
```

The total of the third column above is 3. This represents the sum of totally correct and partially correct numbers in the guess – that is to say: the sum does not distinguish between the two values we are interested in computing. But -- we could subtract the count of totally correct numbers (developed in increment 4), and the amount remaining is just the partially correct figure.

Here is a model of the algorithm just proposed. The arrays that "count" the digits '1' to '6' are at [26]. Their size is one larger than necessary just for the convenience of being able to ignore the "zero" element. The first loop [27] uses each element of the `guess` or `answer` array as the index into the corresponding "counts" array. The second loop [28] compares the last six elements of the two "counts" array, and computes the sum of the "min"s. This is an excellent example of the indirection afforded by arrays – using an array to access an array.

```
int answer[4] = { ... };
int guess[4]  = { ... };
int answer_counts[7] = { 0,0,0,0,0,0,0 };                              [26]
int guess_counts[7]  = { 0,0,0,0,0,0,0 };                              [26]
// Count the number of occurrences of each digit
for (int i=0; i < 4; i = i + 1) {                                      [27]
   // 3) write                    1) read                 2) increment
   answer_counts[ answer[i] ] = answer_counts[ answer[i] ] + 1;
   guess_counts[  guess[i]  ] = guess_counts[  guess[i]  ] + 1;
}
int partially_correct = 0;
for (int i=1, min_value; i < 7; i = i + 1) {                           [28]
   min_value = min( guess_counts[i], answer_counts[i] );
   partially_correct = partially_correct + min_value;
}
```

Integrating this new algorithm into `evaluate_guess()` appears below. Because the `answer_counts` array only needs to be computed once per game, that is being done in `main()` (like `guess_counts` is computed at [29]), and the array is supplied as an argument to `evaluate_guess()`.

```
void evaluate_guess( int answer[], int answer_counts[], int guess[],
                     int& totally_correct, int& partially_correct ) {
   int guess_counts[7] = { 0,0,0,0,0,0,0 };
   totally_correct = partially_correct = 0;
   // Count the number of occurrences of each digit in the guess
   //    [answer_counts was computed in main()]
   for (int i=0; i < 4; i = i + 1)
      guess_counts[ guess[i] ] = guess_counts[ guess[i] ] + 1;      [29]
   // Over-compute the partially_correct value
   for (int i=1; i < 7; i = i + 1)
      partially_correct = partially_correct +
                          min( guess_counts[i], answer_counts[i] );
   // Compute the totally_correct value
   for (int i=0; i < 4; i = i + 1)
      if (guess[i] == answer[i])
         totally_correct = totally_correct + 1;
   // Remove the totally_correct value from the partially_correct value
   partially_correct = partially_correct - totally_correct;
}
```

The new `min()` function, and the changes to `main()` are collected below.

```
int min( int one, int two ) {
   if (one < two) return one;
   else           return two;
}
```

```
int main() {
   ...
   int answer_counts[7] = { 0,0,0,0,0,0,0 };
   for (int i=0; i < 4; i = i + 1)
      answer_counts[ answer[i] ] = answer_counts[ answer[i] ] + 1;
   ...
      evaluate_guess( answer, answer_counts, guess,
                      num_correct, num_partial_correct );
      ...
```

Appendix D ... Peg Game project

Increments

1. draw the "board" with no formatting and no loop(s)
2. draw the "board" with no formatting
3. draw the "board" with formatting
4. prompt for a position, toggle the position
5. implement everything except "jump" logic
6. design and implement the "jump" logic

The goal is to create a character-based user interface for the Peg Game. Missing pegs are represented by a single lower-case character; pegs that are present are upper case and enclosed in angle brackets. Moves are specified by giving the character of the desired peg, followed by the character representing the desired destination. Below – the 'F' peg jumps to the 'a' position, and the 'C' peg is removed.

```
                a

            b    <C>

        <D>   <E>   <F>

    <G>   <H>   <I>   <J>

<K>   <L>   <M>   <N>   <O>

Move (from to): f a

            <A>

            b     c

        <D>   <E>    f

    <G>   <H>   <I>   <J>

<K>   <L>   <M>   <N>   <O>
```

Increment 1. Output the starting point shown below.

```
A
B C
D E F
G H I J
K L M N O
```

Increment 2. Use one or more loops to produce the same output as the previous increment. Consider using an array of characters that holds the characters 'A' through 'O'. This will allow the application to remember upper and lower case characters in subsequent increments.

```
A
B C
D E F
```

```
G H I J
K L M N O
```

Increment 3. Adapt your application to produce the target format for the game.

```
            <A>

         <B>   <C>

      <D>   <E>   <F>

   <G>   <H>   <I>   <J>

<K>   <L>   <M>   <N>   <O>
```

Increment 4. Prompt the user for a position. Toggle the "state" of the position entered: if the peg is present, then remove it, and vice versa. Consider creating a `draw_board()` function and moving the array of position characters to a global variable. Consider adding a global array of "pegs present" booleans that can make it easier to evaluate the current state of each position.

```
            <A>

         <B>   <C>

      <D>   <E>   <F>

   <G>   <H>   <I>   <J>

<K>   <L>   <M>   <N>   <O>

Input: e

            <A>

         <B>   <C>

      <D>    e    <F>

   <G>   <H>   <I>   <J>

<K>   <L>   <M>   <N>   <O>

Input: e

            <A>

         <B>   <C>

      <D>   <E>   <F>

   <G>   <H>   <I>   <J>

<K>   <L>   <M>   <N>   <O>
```

Increment 5. Implement everything except the actual "jump" logic. Prompt the user for the location of the first peg to remove. Then iteratively ask for 2 positions, and toggle the state of both locations. Again – this input does not represent a jump, it is just two arbitrary positions.

```
              <A>

          <B>   <C>

      <D>   <E>   <F>

   <G>   <H>   <I>   <J>

<K>   <L>   <M>   <N>   <O>

Select initial hole: m

              <A>

          <B>   <C>

      <D>   <E>   <F>

   <G>   <H>   <I>   <J>

<K>   <L>    m    <N>   <O>

Move (from to): o m

              <A>

          <B>   <C>

      <D>   <E>   <F>

   <G>   <H>   <I>   <J>

<K>   <L>   <M>   <N>    o

Move (from to): n j

              <A>

          <B>   <C>

      <D>   <E>   <F>

   <G>   <H>   <I>    j

<K>   <L>   <M>    n     o

Move (from to):
```

Increment 6. How do you want to implement the notion of a "jump"? Can you compute all possible jumps? Or do you have to create an exhaustive list of all jumps? Consider the board

portrayal below with positions 1 through 15. All the possible jumps are listed below that. Is there some pattern that can be identified and exploited?

```
          1
        2   3
      4   5   6
    7   8   9   0
  1   2   3   4   5

 1--4  1--6  2-7  2-9  3--8  3-10  4-1  4-11 4-6 4-13
 5-12  5-14  6-1  6-4  6-13  6-15  7-2  7--9 8-3 8-10
 9--2  9--7 10-3 10-8 11--4 11-13 12-5 12-14
13-11 13-15 13-6 13-4 14-12 14--5 15-6 15-13
```

Below, is it possible to jump from A to D? Is it legal to jump from I to B? Instead of focusing on possible jumps, is it better to think in terms of legal jumps? What is a legal jump? Do you need to consider position – and – state (i.e. filled or empty)? Are "from" position and "to" position sufficient for evaluating and processing a jump?

```
                <A>

             b    <C>

          <D>    e    <F>

       <G>    h    <I>   <J>

    <K>   <L>   <M>   <N>   <O>
```

Complete this project by implementing the following interaction. When the number of pegs reaches one, the application should exit.

```
                <A>

             <B>   <C>

          <D>   <E>   <F>

       <G>   <H>   <I>   <J>

    <K>   <L>   <M>   <N>   <O>

    Select initial hole: d

                <A>

             <B>   <C>

           d    <E>   <F>

       <G>   <H>   <I>   <J>

    <K>   <L>   <M>   <N>   <O>

    Move (from to): a d
```

```
             a

          b    <C>

      <D>   <E>   <F>

   <G>   <H>   <I>   <J>

<K>   <L>   <M>   <N>   <O>

Move (from to): d a

NOT A LEGAL MOVE

             a

          b    <C>

      <D>   <E>   <F>

   <G>   <H>   <I>   <J>

<K>   <L>   <M>   <N>   <O>

Move (from to): p j

INVALID INPUT

             a

          b    <C>

      <D>   <E>   <F>

   <G>   <H>   <I>   <J>

<K>   <L>   <M>   <N>   <O>

Move (from to): f a

             <A>

          b     c

      <D>   <E>    f

   <G>   <H>   <I>   <J>

<K>   <L>   <M>   <N>   <O>

Move (from to):
```

Here is an excerpt of `main()` that suggests one possible design.

```
peg_count = 14;   // a global variable
string message = "VALID MOVE";
while (true) {
   if (message != "VALID MOVE")
      cout << '\n' << message << '\n';
   draw_board();
   if (peg_count == 1)
      break;
   cout << "Move (from to): ";
   cin >> from >> to;
   message = attempt_move( from, to );
}
```

`attempt_move()` above does many things: tests if the input is valid, tests if the specified move is possible and legal, performs the jump by toggling the state of all affected positions, and returns a error message if necessary.

Appendix DD ... Peg Game implementations

Increments

1. draw the "board" with no formatting and no loop
2. draw the "board" with no formatting
3. draw the "board" with formatting
4. prompt for a position, toggle the position
5. implement everything except "jump" logic
6. design and implement the "jump" logic

Increment 1. Each line is output with its own `cout` statement. Notice the "\n" carriage returns embedded in each string literal.

```
cout << "A\n";
cout << "B C\n";
cout << "D E F\n";
cout << "G H I J\n";
cout << "K L M N O\n";
```

Increment 2. A very useful property of the triangular arrangement of characters is: the number of characters on each row is equal to that row's number. Testing the column number against the row number [1] is all that is needed to correctly place carriage returns.

The `positions` array [2] is not necessary in this increment. But – it will come in handy later.

```
char positions[15] = { 'A','B','C','D','E','F','G','H',   // [2]
                       'I','J','K','L','M','N','O' };
int i = 0, row = 0, col = 0;
while (row < 5) {
   while (col <= row) {             // [1]
      cout << positions[i] << ' ';
      col = col + 1;
      i = i + 1;
   }
   col = 0;
   row = row + 1;
   cout << '\n';
}

// A
// B C
// D E F
// G H I J
// K L M N O
```

Increment 3. The target output is …

```
              <A>               // row 0

          <B>   <C>             // row 1

      <D>   <E>   <F>           // row 2
```

```
      <G>   <H>   <I>   <J>       // row 3

<K>   <L>   <M>   <N>   <O>    // row 4
```

The indentation of each row is inversely proportional to the number of the row – the lower the row, the more the indentation. Row 0 needs an indentation of 12 spaces, row 1 needs 9 spaces, row two 6 spaces, row three 3 spaces, and row four 0 spaces. The loop at [3] implements this algorithm. The only other change is the "double spacing" at [4].

```
char positions[15] = { 'A','B','C','D','E','F','G','H',
                       'I','J','K','L','M','N','O' };
int i = 0, row = 0, col = 0;
while (row < 5) {
   for (int j=0; j < 4 - row; j = j + 1)       // [3]
      cout << "   ";
   while (col <= row) {
      cout << '<' << positions[i] << ">   ";
      col = col + 1;
      i = i + 1;
   }
   col = 0;
   row = row + 1;
   cout << "\n\n";                              // [4]
}
```

Increment 4. The goal of this increment is the following scenario. When the user enters a position, the program toggles the state (i.e. the case of the letter) of that position.

```
            <A>

         <B>   <C>

      <D>   <E>   <F>

   <G>   <H>   <I>   <J>

<K>   <L>   <M>   <N>   <O>

Position: e

            <A>

         <B>   <C>

      <D>   <e>   <F>

   <G>   <H>   <I>   <J>

<K>   <L>   <M>   <N>   <O>

Position: e

            <A>

         <B>   <C>
```

```
        <D>   <E>   <F>

     <G>   <H>   <I>   <J>

  <K>   <L>   <M>   <N>   <O>
```

Every time the user supplies a position, the "board" is redrawn. Let's encapsulate the "draw board" functionality in its own function [5]. The positions array is needed in `draw_board()` and `main()`. Rather than pass that array (and any other data that may be identified later) in the function invocation, it was decided to make it a global variable [6].

The prompt, user input, and input validation have been added at [7]. [8] demonstrates how a character can be converted to an array index (i.e. an integer in a range from 0 to whatever).

The primary goal of this increment is toggling the value (or state, or "fullness") of a specified position on the board. It is possible to identify if an element of the `positions` array is present or not by testing its case, but it is easier to test a boolean variable. For this reason, a "pegs present" array has been added at [9], referenced at [10], and toggled at [11]. Converting the case of a character is a simple matter of adding or removing an offset of 32 [12].

```
char positions[15] = { 'A','B','C','D','E','F','G','H',     //  [6]
                       'I','J','K','L','M','N','O' };
bool pegs_present[15] = { true,true,true,true,true,true,    //  [9]
          true,true,true,true,true,true,true,true,true };

void draw_board() {                                         //  [5]
   int i = 0, row = 0, col = 0;
   cout << '\n';
   while (row < 5) {
      for (int j=0; j < 4 - row; j = j + 1)
         cout << "   ";
      while (col <= row) {
         if (pegs_present[i])                                   // [10]
            cout << '<' << positions[i] << ">   ";
         else
            cout << ' ' << positions[i] << "    ";
         col = col + 1;
         i = i + 1;
      }
      col = 0;
      row = row + 1;
      cout << "\n\n";
}  }

int main() {
   char letter;
   int posit;
   while (true) {
      draw_board();                                         //  [5]
      cout << "Position: ";                                 //  [7]
      cin >> letter;
      if (letter < 'a'  ||  letter > 'o')
         continue;
```

```
      posit = letter - 'a';                                    //  [8]
      if (pegs_present[posit])                                 // [10]
         positions[posit] = positions[posit] + 32;             // [12]
      else
         positions[posit] = positions[posit] - 32;             // [12]
      pegs_present[posit] = ! pegs_present[posit];             // [11]
} }
```

Increment 5. The goal here is to implement everything except the actual "jump" logic. Prompting for the initial hole has been added at [13]. Prompting for "from" and "to" positions has been changed at [14].

Since there are now two positions whose state need to be toggled, that logic has been encapsulated in a new function `toggle_peg(posit)` at [15] and invoked at [16].

```
char positions[15] = { ... };
bool pegs_present[15] = { ... };

void draw_board() { ... }

void toggle_peg( int i ) {                 // [15]
   if (pegs_present[i])
      positions[i] = positions[i] + 32;
   else
      positions[i] = positions[i] - 32;
   pegs_present[i] = ! pegs_present[i];
}

int main() {
   char from, to;
   draw_board();
   cout << "Select initial hole: ";        // [13]
   cin >> from;
   toggle_peg( from - 'a' );               // [16]
   while (true) {
      draw_board();
      cout << "Move (from to): ";          // [14]
      cin >> from >> to;
      if (from < 'a' || from > 'o' || to < 'a' || to > 'o')
         continue;
      toggle_peg( from - 'a' );            // [16]
      toggle_peg( to - 'a' );              // [16]
} }
```

Increment 6. There does not seem to be any insight or cunning that would allow us to compute or evaluate a proposed move at run-time. So, a brute force enumeration of all possible moves seems necessary.

The notion of "jump" needs to include three things: the "from" position, the "to" position (destination), and the "jumped over" position. An array of three integers can be used to represent the abstraction of a jump. The total number of possible jumps is 36. Taken together, a two-dimensional "36 by 3" array of integers is the starting point for the jump logic [17].

The `attempt_move()` [18] function needs to do many things: test if the input is valid [19], test if the specified move is possible [20] and valid [21], perform the jump by toggling the state of all affected positions [22], and return a error message if necessary [23].

To identify if the requested move is possible, a low-tech sequential search is used at [20]. For a jump to be valid: the "from" position must be occupied [21a], the "jumped over" position must be occupied [21b], and the "to" position must be empty [21c].

A new global variable (`peg_count`) has been added at [24], updated at [25], and used to determine when the game has been won [26].

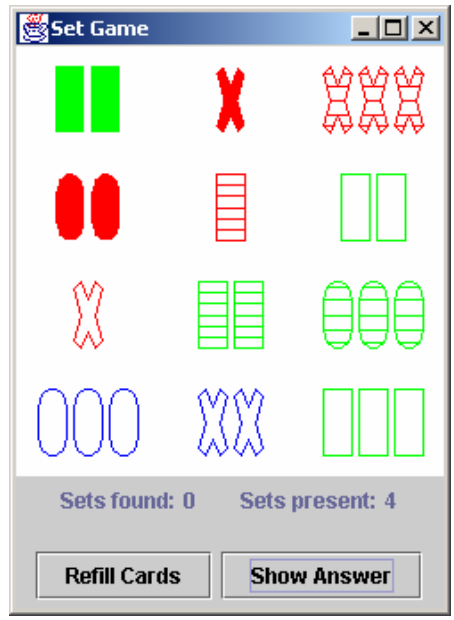
```
char positions[15] = { ... };
bool pegs_present[15] = { ... };
                        // {1,4,2} = { from, to, over }
int jump_table[36][3] = { {1,4,2}, {1,6,3}, {2,7,4}, {2,9,5},    // [17]
                          {3,8,5}, {3,10,6},
                          {4,6,5}, {4,1,2}, {4,11,7}, {4,13,8},
                          {5,14,9}, {5,12,8},
                          {6,4,5}, {6,13,9}, {6,15,10}, {6,1,3},
                          {7,2,4}, {7,9,8}, {8,3,5}, {8,10,9},
                          {9,2,5}, {9,7,8},
                          {10,8,9}, {10,3,6}, {11,13,12}, {11,4,7},
                          {12,5,8}, {12,14,13},
                          {13,11,12}, {13,15,14}, {13,6,9}, {13,4,8},
                          {14,12,13}, {14,5,9}, {15,13,14}, {15,6,10} };
int peg_count = 15;                                              // [24]

void toggle_peg( int i ) { ... }

string attempt_move( char from, char to ) {                      // [18]
   if (from < 'a' || from > 'o' || to < 'a' || to > 'o')          // [19]
      return "INVALID INPUT";                                    // [23]
   int one = from - 'a' + 1, two = to - 'a' + 1;
   for (int i=0; i < 36; i = i + 1)
      if (one == jump_table[i][0]  &&  two == jump_table[i][1])  // [20]
         if (pegs_present[ jump_table[i][0]-1 ]                  // [21a]
               &&  pegs_present[ jump_table[i][2]-1 ]            // [21b]
               &&  ( ! pegs_present[ jump_table[i][1]-1 ])) {    // [21c]
            toggle_peg( jump_table[i][0]-1 );   // source_peg    // [22]
            toggle_peg( jump_table[i][2]-1 );   // jumped_peg    // [22]
            toggle_peg( jump_table[i][1]-1 );   // destination   // [22]
            peg_count = peg_count - 1;                           // [25]
            return "VALID MOVE";
         }
   return "NOT A LEGAL MOVE";                                    // [23]
}

void draw_board() { ... }

int main() {
   char from, to;
   draw_board();
   cout << "Select initial hole: ";
   cin >> from;
```

```
   toggle_peg( from - 'a' );
   peg_count = 14;
   string message = "VALID MOVE";
   while (true) {
      if (message != "VALID MOVE")
         cout << '\n' << message << '\n';
      draw_board();
      if (peg_count == 1)                                          // [26]
         break;
      cout << "Move (from to): ";
      cin >> from >> to;
      message = attempt_move( from, to );
}  }
```

Appendix E ... Set Game project

Increments

1. print all combinations of three three-value variables
2. print all combinations of ./+/^ X/O/S 1/2/3
3. initialize and output a "int deck_of_cards[27][3]" data structure
4. replace "X3" format with "XXX" implementation
5. design initialize_cards() and draw_card() functions
6. replace state array with struct
7. add draw_board(), solicit card number (1-12), remove card
8. add a "fill" command
9. modify initialize_cards() to shuffle the deck of cards
10. design is_a_set()
11. design compute_sets_present()



The rules for the Set Game were introduced in the Introduction. The application evaluates all combinations of the displayed cards and counts the total number of sets present. The user may ask for the "answer" to be displayed. When three cards are selected, the game evaluates whether the cards represent a set. If so: the cards are removed, and the "Sets present" count is incremented. If not: an error message is displayed.

Since a character-based user interface is very limited in its ability to display dense information content, this project will limit itself to three of the game's four dimensions: fill, shape, and number (color has been dropped). The normal game has 81 cards (3 to the 4th power). Our scaled back game will have 27 cards (3 to the 3rd power). Here is the entire deck of cards (the characters ./+/^ represent the three possible "fill" values.

```
. X     . XX    . XXX   . O     . OO    . OOO   . S     . SS    . SSS
+ X     + XX    + XXX   + O     + OO    + OOO   + S     + SS    + SSS
^ X     ^ XX    ^ XXX   ^ O     ^ OO    ^ OOO   ^ S     ^ SS    ^ SSS
```

The user interface that will be developed in this chapter appears below. The user's first proposed set is ". SSS", ". X", and ". OO". The application identifies that all three fill characteristics are the same, the three shape characteristics are different, the three number characteristics are different, and the cards therefore represent a set. The second proposed set ("^ XX", ". O", "+ SSS") has different fills, different shapes, and different numbers. The third set ("+ SS", "+ OO", "+ XX") has same fill, different shapes, same number. The "fill" command then replaces all missing cards and computes the new number of sets present.

```
. SSS   ^ XX    ^ XXX   + XXX
. O     + SS    + OO    . X
^ O     + SSS   + XX    . OO

sets present 8, sets found 0
```

```
Input: answer

answer 1 8 12   1 9 11   2 4 8   2 5 10
       2 6 12   3 5 6   3 8 11   6 7 11

. SSS    ^ XX    ^ XXX   + XXX
. O      + SS    + OO    . X
^ O      + SSS   + XX    . OO

sets present 8, sets found 0
Input: 1 8 12

         ^ XX    ^ XXX   + XXX
. O      + SS    + OO
^ O      + SSS   + XX

sets present 3, sets found 1
Input: answer

answer 2 5 10   3 5 6   6 7 11

         ^ XX    ^ XXX   + XXX
. O      + SS    + OO
^ O      + SSS   + XX

sets present 3, sets found 1
Input: 2 5 10

                 ^ XXX   + XXX
         + SS    + OO
^ O              + XX

sets present 1, sets found 2
Input: 6 7 11

                 ^ XXX   + XXX

^ O

sets present 0, sets found 3
Input: fill

+ S      + O     ^ XXX   + XXX
^ OOO    . XXX   . XX    ^ SSS
^ O      + OOO   . S     ^ SS

sets present 7, sets found 3
```

Increment 1. Let's consider how to represent the Set Game deck of cards. Each card has three attributes, and each attribute can take on one of three values – 3 fill values times 3 shape values times 3 number values equals 27 cards.

```
. X      . XX     . XXX    . O      . OO     . OOO    . S      . SS     . SSS
+ X      + XX     + XXX    + O      + OO     + OOO    + S      + SS     + SSS
```

```
^ X     ^ XX    ^ XXX   ^ O     ^ OO    ^ OOO   ^ S     ^ SS    ^ SSS
```

To get started, model this "three to the third power" domain with one or more loops and produce the following output.

```
000 001 002 010 011 012 020 021 022
100 101 102 110 111 112 120 121 122
200 201 202 210 211 212 220 221 222
```

Increment 2. Add an array of "fill" characters, an array of "shape" characters, and an array of "number" characters, and use your loop control variables to index into these arrays. The target output follows.

```
.X1    .X2    .X3    .O1    .O2    .O3    .S1    .S2    .S3
+X1    +X2    +X3    +O1    +O2    +O3    +S1    +S2    +S3
^X1    ^X2    ^X3    ^O1    ^O2    ^O3    ^S1    ^S2    ^S3
```

Increment 3. Down the road, designing a "Card" abstraction is our goal. To get to that goal, let's consider the two-dimensional array data structure below.

```
int deck_of_cards[27][3];
// deck_of_cards[i] ----- the i-th card
// deck_of_cards[i][0] -- the fill value of the i-th card
// deck_of_cards[i][1] -- the shape value of the i-th card
// deck_of_cards[i][2] -- the number value of the i-th card
```

To initialize this two-dimensional array, we could use brute assignment …

```
int deck_of_cards[27][3] = { {0,0,0}, {0,0,1}, {0,0,2}, {0,1,0} ...
```

Is there some way to use one or more loops instead of "brute force"?

```
for (int i=0; i < 3; i = i + 1)
   for (int j=0; j < 3; j = j + 1)
      for (int k=0; k < 3; k = k + 1)
         deck_of_cards[ ?? ][ ?? ] = ??;
```

How about …

```
for (int i=0; i < 3; i = i + 1)
   for (int j=0; j < 3; j = j + 1)
      for (int k=0; k < 3; k = k + 1) {
         deck_of_cards[ XX ][0] = i;
         deck_of_cards[ XX ][1] = j;
         deck_of_cards[ XX ][2] = k;
      }
```

How can we make XX go from 0 to 26?

```
  0   1   2   3   4   5   6   7   8
000 001 002 010 011 012 020 021 022

  9  10  11  12  13  14  15  16  17
100 101 102 110 111 112 120 121 122

 18  19  20  21  22  23  24  25  26
200 201 202 210 211 212 220 221 222
```

Is there some way to "compute" XX by using arithmetic on the variables i, j, and k? Here is an alternate representation.

```
i  j  k  XX    possible arithmetic
0  0  0   0 -> 0+0+0
      1   1 -> 0+0+1
      2   2 -> 0+0+2
   1  0   3 -> 0+3+0
      1   4 -> 0+3+1
      2   5 -> 0+3+2
   2  0   6 -> 0+6+0
      1   7 -> 0+6+1
      2   8 -> 0+6+2
1  0  0   9 -> 9+0+0
      1  10 -> 9+0+1
      2  11 -> 9+0+2
   1  0  12 -> 9+3+0
   ...
```

Add to your program the declaration and initialization of the two-dimensional array deck_of_cards, and then use deck_of_cards to output the following ...

```
.X1    .X2    .X3    .O1    .O2    .O3    .S1    .S2    .S3
+X1    +X2    +X3    +O1    +O2    +O3    +S1    +S2    +S3
^X1    ^X2    ^X3    ^O1    ^O2    ^O3    ^S1    ^S2    ^S3
```

Increment 4. Let's refactor the "shape" and "number" implementations, and produce the following output …

```
. X     . XX    . XXX   . O     . OO    . OOO   . S     . SS    . SSS
+ X     + XX    + XXX   + O     + OO    + OOO   + S     + SS    + SSS
^ X     ^ XX    ^ XXX   ^ O     ^ OO    ^ OOO   ^ S     ^ SS    ^ SSS
```

We could use brute force and do something like ...

```
int main( void ) {
   int deck_of_cards[27][3];
   ...
   char fills[] = { '.', '+', '^' };
   for (int i=0; i < 27; i = i + 1) {
      cout << fills[ deck_of_cards[i][0] ] << ' ';
[1]   if      (deck_of_cards[i][1] == 0  &&  deck_of_cards[i][2] == 0)
         cout << "X     ";
[2]   else if (deck_of_cards[i][1] == 0  &&  deck_of_cards[i][2] == 1)
         cout << "XX    ";
[3]   else if (deck_of_cards[i][1] == 0  &&  deck_of_cards[i][2] == 2)
         cout << "XXX   ";
[4]   else if (deck_of_cards[i][1] == 1  &&  deck_of_cards[i][2] == 0)
         cout << "O     ";
[5]   else if (deck_of_cards[i][1] == 1  &&  deck_of_cards[i][2] == 1)
         cout << "OO    ";
[6]   else if (deck_of_cards[i][1] == 1  &&  deck_of_cards[i][2] == 2)
         cout << "OOO   ";
[7]   else if (deck_of_cards[i][1] == 2  &&  deck_of_cards[i][2] == 0)
         cout << "S     ";
[8]   else if (deck_of_cards[i][1] == 2  &&  deck_of_cards[i][2] == 1)
```

```
            cout << "SS    ";
[9]   else if (deck_of_cards[i][1] == 2  &&  deck_of_cards[i][2] == 2)
            cout << "SSS   ";
         if ((i+1) % 9 == 0)
            cout << '\n';
}  }
```

Another alternative could be to use a 2-dimensional array to collapse "algorithm" (i.e. code) into "data structure". Frequently, a large multiple branch if-then-else block can be refactored into an array data structure – if – an indexing scheme can be conceived that allows the correct array element to be accessed. In this case, [1] through [9] above correspond to [1] through [9] below. The elements of the array below correspond to the "boundary conditions" being tested in the `if` statements above. `deck_of_cards[i][1]` below points to the desired shape array, and `deck_of_cards[i][2]` points to the desired number element within that shape array.

```
int main( void ) {
   int deck_of_cards[27][3];
   ...
   char fills[] = { '.', '+', '^' };   [1]    [2]    [3]
   string shaps_numbs[3][3] =      { {"X  ", "XX ", "XXX"},
                                       [4]    [5]    [6]
                                     {"O  ", "OO ", "OOO"},
                                       [7]    [8]    [9]
                                     {"S  ", "SS ", "SSS"} };
   for (int i=0; i < 27; i = i + 1) {
      cout << fills[ deck_of_cards[i][0] ] << ' ';
      cout << shaps_numbs[ deck_of_cards[i][1] ][ deck_of_cards[i][2] ]
           << "   ";
      if ((i+1) % 9 == 0)
         cout << '\n';
}  }
```

But – let's do this instead: use a loop to print each card with the correct number of shape characters.

```
for (int i=0; i < 27; i = i + 1) {
   cout << fills[ /* the fill value of the i-th card */ ] << ' ';
   for (int j=0; j < /* the numb value of the i-th card */; j = j + 1)
      cout << shaps[ /* the shap value of the i-th card */ ]
```

Notice that the shap characters are left-justified and right-filled (with spaces). How do you want to handle right-filling with spaces?

Increment 5. Define `initialize_cards()` and `draw_card()` functions by refactoring your current implementation. Then, draw the first 12 cards in three rows.

```
. X     . XX    . XXX   . O
. OO    . OOO   . S     . SS
. SSS   + X     + XX    + XXX
```

Increment 6. In future increments, we will want to be able to assign "cards" from one array to another. But that won't work with a data structure implemented as an array.

```
int deck_of_cards[27][3] = { ... };
int displayed_cards[12][3] = { ... };
// compiler error: ISO C++ forbids assignment of arrays
displayed_cards[0] = deck_of_cards[0];
```

The "`struct`" data structure does not manifest this limitation.

```
struct Card { int fill, shap, numb; };
Card deck_of_cards[27] = { ... };
Card displayed_cards[12] = { ... };
// no compiler error
displayed_cards[0] = deck_of_cards[0];
```

In this increment, convert the "deck of cards" abstraction from a two-dimensional array to an array of structs. The output will remain the same ...

```
. X     . XX    . XXX   . O
. OO    . OOO   . S     . SS
. SSS   + X     + XX    + XXX
```

Increment 7. In Set, the user specifies three cards, and the cards are removed if they constitute a set. In this increment, add a `draw_board()` function, add and initialize a "displayed cards" array, prompt the user for a card number (use the range 1-12), and remove the specified card. How do you want to keep up with cards that have been removed? You could add an additional array that maintains the present/absent memory. You could "hijack" a member of the Card struct and use a "special value" to represent a missing card.

```
. X     . XX    . XXX   . O
. OO    . OOO   . S     . SS
. SSS   + X     + XX    + XXX

Input: 1

        . XX    . XXX   . O
. OO    . OOO   . S     . SS
. SSS   + X     + XX    + XXX

Input: 6

        . XX    . XXX   . O
. OO            . S     . SS
. SSS   + X     + XX    + XXX

Input: 11

        . XX    . XXX   . O
. OO            . S     . SS
. SSS   + X             + XXX
```

Increment 8. Add a "fill" command to your user interface that "deals" cards to replace missing cards. The application must now be able to accept string or numeric input.

```
. X     . XX    . XXX   . O
. OO    . OOO   . S     . SS
. SSS   + X     + XX    + XXX

Input: 1

        . XX    . XXX   . O
. OO    . OOO   . S     . SS
. SSS   + X     + XX    + XXX
```

```
Input: 2

                  . XXX    . O
. OO     . OOO    . S      . SS
. SSS    + X      + XX     + XXX

Input: 3

                           . O
. OO     . OOO    . S      . SS
. SSS    + X      + XX     + XXX

Input: fill

+ O      + OO     + OOO    . O
. OO     . OOO    . S      . SS
. SSS    + X      + XX     + XXX
```

Increment 9. Modify `initialize_cards()` to "shuffle" the deck of cards. Should you try to replicate human shuffling, or is there a simple brute-force-like algorithm that is possible?

```
+ X      ^ OO     + XX     + SS
. XX     + O      . X      + OOO
^ SSS    . O      + SSS    ^ S
```

Increment 10. Design an `is_a_set(card,card,card)` function. We want to advance the application's user interface to accept three numbers, and evaluate whether the three cards form a set. Below, the user specifies "1 2 3" and the program responds "NOT a set". The user then inputs "2 8 10" and the program: identified those cards represent a set, removed the cards from the "board", and incremented the "sets found" count. "6 7 12" was also evaluated to be a set. Finally, "1 2 3" resulted in the response "BAD INPUT" because card 2 doesn't currently exist.

```
+ X      ^ OO     + XX     + SS
. XX     + O      . X      + OOO
^ SSS    . O      + SSS    ^ S

sets found - 0
Input: 1 2 3

NOT a set

+ X      ^ OO     + XX     + SS
. XX     + O      . X      + OOO
^ SSS    . O      + SSS    ^ S

sets found - 0
Input: 2 8 10

+ X              + XX     + SS
. XX     + O      . X
^ SSS            + SSS    ^ S

sets found - 1
Input: 6 7 12
```

```
+ X             + XX    + SS
. XX
^ SSS           + SSS

sets found - 2
Input: 1 2 3

BAD INPUT

+ X             + XX    + SS
. XX
^ SSS           + SSS

sets found - 2
Input:
```

A brute force `is_a_set()` design might look like the following ...

```
bool is_a_set( Card one, Card two, Card three ) {
   bool shap_is_good = false, fill_is_good = false, numb_is_good = false;
   if ((one.numb == two.numb && two.numb == three.numb) ||
         (one.numb != two.numb && one.numb != three.numb
                               && two.numb != three.numb))
      numb_is_good = true;
   if ((one.fill == two.fill && two.fill == three.fill) ||
         (one.fill != two.fill && one.fill != three.fill
                               && two.fill != three.fill))
      fill_is_good = true;
   if ((one.shap == two.shap && two.shap == three.shap) ||
         (one.shap != two.shap && one.shap != three.shap
                               && two.shap != three.shap))
      shap_is_good = true;
   return shap_is_good && fill_is_good && numb_is_good;
}
```

Is there some algorithm that can be designed to "compute" a true/false response? Consider the table below. Most of the permutations of 0, 1, and 2 are listed. Is there a "pattern" to the permutations that is capable of deterministically identifying if the specified cards represent a set?

```
             0   0   1   1   2   2   0   0   1   2
             0   1   1   2   2   0   1   0   1   2
             1   1   2   2   0   0   2   0   1   2
Is a set?  no  no  no  no  no  no  yes yes yes yes
```

Increment 11. Design a `compute_sets_present()` function. One of the frustrations of playing Set is: staring at the cards and trying to identify if any sets exist in the remaining cars. It sure would be nice if: the game could tell you how many sets exist in the displayed cards.

How to proceed? What do you think about a strategy of testing every 3-card combination of the 12 displayed cards? Is that an outrageously large number of tests, or a reasonable number of tests? Is there a mathematical formula for computing "all combinations of 12 things taken 3 at a time"?

The answer is, "Yes."

```
12 * 11 * 10 / 3!
```

```
12 * 11 * 10 / 3 * 2 * 1
12 * 110 / 6
2 * 110 = 220
```

I think 220 tests is completely reasonable! How can these 220 combinations be enumerated? Would the odometer be a useful inspiration? What about imagining 3 odometer-like digit wheels, each with some configuration of the numbers 1 through 12?

Here are all 220 combinations. How could you implement this enumeration?

```
1.2.3 1.2.4 1.2.5 1.2.6 1.2.7 1.2.8 1.2.9 1.2.10 1.2.11 1.2.12
1.3.4 1.3.5 1.3.6 1.3.7 1.3.8 1.3.9 1.3.10 1.3.11 1.3.12
1.4.5 1.4.6 1.4.7 1.4.8 1.4.9 1.4.10 1.4.11 1.4.12
1.5.6 1.5.7 1.5.8 1.5.9 1.5.10 1.5.11 1.5.12
1.6.7 1.6.8 1.6.9 1.6.10 1.6.11 1.6.12
1.7.8 1.7.9 1.7.10 1.7.11 1.7.12
1.8.9 1.8.10 1.8.11 1.8.12
1.9.10 1.9.11 1.9.12
1.10.11 1.10.12
1.11.12
2.3.4 2.3.5 2.3.6 2.3.7 2.3.8 2.3.9 2.3.10 2.3.11 2.3.12
2.4.5 2.4.6 2.4.7 2.4.8 2.4.9 2.4.10 2.4.11 2.4.12
2.5.6 2.5.7 2.5.8 2.5.9 2.5.10 2.5.11 2.5.12
2.6.7 2.6.8 2.6.9 2.6.10 2.6.11 2.6.12
2.7.8 2.7.9 2.7.10 2.7.11 2.7.12
2.8.9 2.8.10 2.8.11 2.8.12
2.9.10 2.9.11 2.9.12
2.10.11 2.10.12
2.11.12
3.4.5 3.4.6 3.4.7 3.4.8 3.4.9 3.4.10 3.4.11 3.4.12
3.5.6 3.5.7 3.5.8 3.5.9 3.5.10 3.5.11 3.5.12
3.6.7 3.6.8 3.6.9 3.6.10 3.6.11 3.6.12
3.7.8 3.7.9 3.7.10 3.7.11 3.7.12
3.8.9 3.8.10 3.8.11 3.8.12
3.9.10 3.9.11 3.9.12
3.10.11 3.10.12
3.11.12
4.5.6 4.5.7 4.5.8 4.5.9 4.5.10 4.5.11 4.5.12
4.6.7 4.6.8 4.6.9 4.6.10 4.6.11 4.6.12
4.7.8 4.7.9 4.7.10 4.7.11 4.7.12
4.8.9 4.8.10 4.8.11 4.8.12
4.9.10 4.9.11 4.9.12
4.10.11 4.10.12
4.11.12
5.6.7 5.6.8 5.6.9 5.6.10 5.6.11 5.6.12
5.7.8 5.7.9 5.7.10 5.7.11 5.7.12
5.8.9 5.8.10 5.8.11 5.8.12
5.9.10 5.9.11 5.9.12
5.10.11 5.10.12
5.11.12
6.7.8 6.7.9 6.7.10 6.7.11 6.7.12
6.8.9 6.8.10 6.8.11 6.8.12
6.9.10 6.9.11 6.9.12
6.10.11 6.10.12
6.11.12
```

```
7.8.9 7.8.10 7.8.11 7.8.12
7.9.10 7.9.11 7.9.12
7.10.11 7.10.12
7.11.12
8.9.10 8.9.11 8.9.12
8.10.11 8.10.12
8.11.12
9.10.11 9.10.12
9.11.12
10.11.12
count is 220
```

`compute_sets_present()` should concatenate all possible sets into a string. Add a new "answer" command to the application's user interface. The following dialog is the target ...

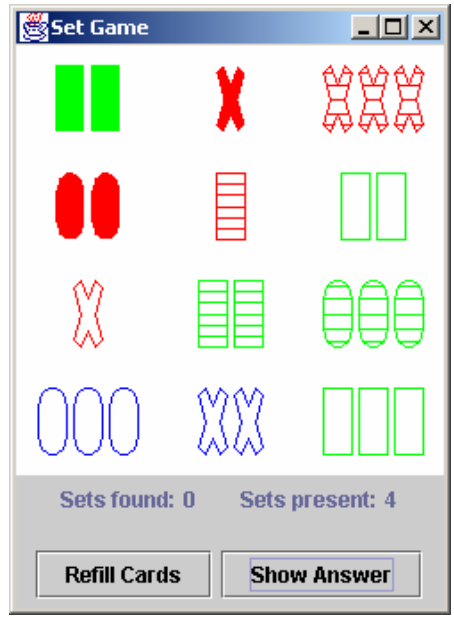
```
. SSS    ^ XX     ^ XXX    + XXX
. O      + SS     + OO     . X
^ O      + SSS    + XX     . OO

sets present 8, sets found 0
Input: answer

answer 1 8 12   1 9 11   2 4 8   2 5 10   2 6 12   3 5 6
       3 8 11   6 7 11

. SSS    ^ XX     ^ XXX    + XXX
. O      + SS     + OO     . X
^ O      + SSS    + XX     . OO

sets present 8, sets found 0
Input: 1 8 12

         ^ XX     ^ XXX    + XXX
. O      + SS     + OO
^ O      + SSS    + XX

sets present 3, sets found 1
Input: answer

answer 2 5 10   3 5 6   6 7 11

         ^ XX     ^ XXX    + XXX
. O      + SS     + OO
^ O      + SSS    + XX

sets present 3, sets found 1
Input: 2 5 10

                  ^ XXX    + XXX
         + SS     + OO
^ O               + XX

sets present 1, sets found 2
Input: 6 7 11
```

```
                ^ XXX   + XXX

^ O

sets present 0, sets found 3
Input: fill

+ S     + O     ^ XXX   + XXX
^ OOO   . XXX   . XX    ^ SSS
^ O     + OOO   . S     ^ SS

sets present 7, sets found 3
Input:
```

Appendix EE ... Set Game implementations

Increments

1. print all combinations of three three-value variables
2. print all combinations of ./+/^ X/O/S 1/2/3
3. initialize and output a "int deck_of_cards[27][3]" data structure
4. replace "X3" format with "XXX" implementation
5. design initialize_cards() and draw_card() functions
6. replace state array with struct
7. add draw_board(), solicit card number (1-12), remove card
8. add a "fill" command
9. modify initialize_cards() to shuffle the deck of cards
10. design is_a_set()
11. design compute_sets_present()



Increment 1. The target output appears below. Notice that the rightmost digit of each triple changes fastest, and the leftmost digit changes slowest.

```
// 000 001 002 010 011 012 020 021 022
// 100 101 102 110 111 112 120 121 122
// 200 201 202 210 211 212 220 221 222
```

Let's use a loop to drive the rightmost digit [1], and put that inside a loop to drive the center digit [2], and put that inside a loop to drive the leftmost digit [3].

```
for (int i=0; i < 3; i = i + 1) {        // [3]
   for (int j=0; j < 3; j = j + 1)       // [2]
      for (int k=0; k < 3; k = k + 1)    // [1]
         cout << i << j << k << ' ';
   cout << '\n';
}
```

Increment 2. The next step is to produce a representation that more closely resembles the actual look of the Set game …

```
// .X1    .X2    .X3    .O1    .O2    .O3    .S1    .S2    .S3
// +X1    +X2    +X3    +O1    +O2    +O3    +S1    +S2    +S3
// ^X1    ^X2    ^X3    ^O1    ^O2    ^O3    ^S1    ^S2    ^S3
```

Instead of outputting i, j, and k, let's use them to index into three new arrays [4] that contain the "card attributes" we wish to convey.

```
char fills[] = { '.', '+', '^' };     // [4]
char shaps[] = { 'X', 'O', 'S' };     // [4]
char numbs[] = { '1', '2', '3' };     // [4]
for (int i=0; i < 3; i = i + 1) {
   for (int j=0; j < 3; j = j + 1)
```

```
      for (int k=0; k < 3; k = k + 1)
         cout << fills[i] << shaps[j] << numbs[k] << "    ";
   cout << '\n';
}
```

Increment 3. The goal here is to model the deck of cards with a two-dimensional array, and initialize the 2D array as effortlessly as possible.

```
int deck_of_cards[27][3];
for (int i=0; i < 3; i = i + 1)
   for (int j=0; j < 3; j = j + 1)
      for (int k=0; k < 3; k = k + 1) {
         deck_of_cards[ XX ][0] = i;
         deck_of_cards[ XX ][1] = j;
         deck_of_cards[ XX ][2] = k;
      }
```

Let's assign an ordinal number to each of the 27 cards. Notice that the "fill" attribute is constant for each row, and changes when the counter advances first to 9 and second to 18.

```
00     01     02     03     04     05     06     07     08
.X1    .X2    .X3    .O1    .O2    .O3    .S1    .S2    .S3

09     10     11     12     13     14     15     16     17
+X1    +X2    +X3    +O1    +O2    +O3    +S1    +S2    +S3

18     19     20     21     22     23     24     25     26
^X1    ^X2    ^X3    ^O1    ^O2    ^O3    ^S1    ^S2    ^S3
```

That insight could be implemented by using an outer loop and multiplying its loop control variable by 9.

```
for (int i=0; i < 3; i = i + 1)
   for (int j=0; j < 3; j = j + 1)
      for (int k=0; k < 3; k = k + 1) {
         deck_of_cards[ i*9 + ... ][0] = i;
```

Notice that the "shape" attribute changes every third card, and the "number" attribute changes **every** card. These two insights taken together can complete the computation of the desired sequence …

```
for (int i=0; i < 3; i = i + 1)
   for (int j=0; j < 3; j = j + 1)
      for (int k=0; k < 3; k = k + 1) {
         deck_of_cards[ i*9 + j*3 + k ][0] = i;
```

Outputting the deck of cards can now be collapsed from three embedded loops to a single loop that steps through cards 0 to 26. The "fill" attribute (i.e. the "i" variable) is captured at [5] and then recalled and used at [6].

```
for (int i=0; i < 3; i = i + 1)
   for (int j=0; j < 3; j = j + 1)
      for (int k=0; k < 3; k = k + 1) {
         deck_of_cards[i*9 + j*3 + k][0] = i;       // [5]
         deck_of_cards[i*9 + j*3 + k][1] = j;
         deck_of_cards[i*9 + j*3 + k][2] = k;
      }
char fills[] = { '.', '+', '^' };
```

```
char shaps[] = { 'X', 'O', 'S' };
char numbs[] = { '1', '2', '3' };
for (int i=0; i < 27; i = i + 1) {
   cout << fills[ deck_of_cards[i][0] ]              // [6]
        << shaps[ deck_of_cards[i][1] ]
        << numbs[ deck_of_cards[i][2] ] << "    ";
   if ((i+1) % 9 == 0)
      cout << '\n';
}
```

Increment 4. Next – replace the temporary "number" implementation with the target implementation.

```
temporary implementation ...
.X1    .X2    .X3    .O1    .O2    .O3    .S1    .S2    .S3
+X1    +X2    +X3    +O1    +O2    +O3    +S1    +S2    +S3
^X1    ^X2    ^X3    ^O1    ^O2    ^O3    ^S1    ^S2    ^S3

target implementation ...
. X     . XX    . XXX   . O     . OO    . OOO   . S     . SS    . SSS
+ X     + XX    + XXX   + O     + OO    + OOO   + S     + SS    + SSS
^ X     ^ XX    ^ XXX   ^ O     ^ OO    ^ OOO   ^ S     ^ SS    ^ SSS
```

We can retire the "`numbs[]`" array, and replace it with a loop [7]. If the limit of the "number" attribute has not been reached [8], then print an instance of the "shape" attribute [9]; else print a blank [10].

```
int deck_of_cards[27][3];
...
char fills[] = { '.', '+', '^' };
char shaps[] = { 'X', 'O', 'S' };

for (int i=0; i < 27; i = i + 1) {
   cout << fills[ deck_of_cards[i][0] ] << ' ';
   for (int j=0; j < 3; j = j + 1)                  //  [7]
      if (j <= deck_of_cards[i][2])                 //  [8]
         cout << shaps[ deck_of_cards[i][1] ];      //  [9]
      else
         cout << ' ';                               // [10]
   cout << "   ";
   if ((i+1) % 9 == 0)
      cout << '\n';
}
```

Increment 5. Let's start packaging existing implementation into logical units. `initialize_cards()` and `draw_card()` seem like good units of functionality.

```
void initialize_cards( int deck_of_cards[27][3] ) {
   for (int i=0; i < 3; i = i + 1)
      for (int j=0; j < 3; j = j + 1)
         for (int k=0; k < 3; k = k + 1) {
            deck_of_cards[i*9 + j*3 + k][0] = i;
            deck_of_cards[i*9 + j*3 + k][1] = j;
            deck_of_cards[i*9 + j*3 + k][2] = k;
}        }
```

```
void draw_card( int card[3] ) {
   char fills[] = { '.', '+', '^' };
   char shaps[] = { 'X', 'O', 'S' };

   cout << fills[ card[0] ] << ' ';
   for (int j=0; j < 3; j = j + 1)
      if (j <= card[2])
         cout << shaps[ card[1] ];
      else
         cout << ' ';
   cout << "   ";
}
```

Given these building blocks, printing the first 12 cards in `main()` becomes easy.

```
int main( void ) {
   int deck_of_cards[27][3];
   initialize_cards( deck_of_cards );
   for (int i=0; i < 12; i = i + 1) {
      draw_card( deck_of_cards[i] );
      if ((i+1) % 4 == 0)
         cout << '\n';
}  }

// . X     . XX    . XXX   . O
// . OO    . OOO   . S     . SS
// . SSS   + X     + XX    + XXX
```

Increment 6. In anticipation of more sophisticated manipulation of cards (maintaining a "deck of cards" array and a "displayed cards" array, and moving cards from the former to the latter), convert the second dimension of `deck_of_cards` from a 3-element array to a 3-member `struct`. The definition of this new `Card` abstraction is at [11], and the repackaging of `deck_of_cards` is at [12].

```
struct Card { int fill, shap, numb; };     // [11]

int main( void ) {
   Card deck_of_cards[27];                 // [12]
   initialize_cards( deck_of_cards );
   for (int i=0; i < 12; i = i + 1) {
      draw_card( deck_of_cards[i] );
   ...
```

Using a card is now more "self documenting". What was `deck_of_cards[1][0]` is now `deck_of_cards[1].fill`, and `card[1]` is now `card.shap`.

```
void initialize_cards( Card deck_of_cards[] ) {
   for (int i=0; i < 3; i = i + 1)
      for (int j=0; j < 3; j = j + 1)
         for (int k=0; k < 3; k = k + 1) {
            deck_of_cards[i*9 + j*3 + k].fill = i;  // formerly "[i][0]"
            deck_of_cards[i*9 + j*3 + k].shap = j;  // formerly "[i][1]"
            deck_of_cards[i*9 + j*3 + k].numb = k;  // formerly "[i][2]"
}        }

void draw_card( Card card ) {
   char fills[] = { '.', '+', '^' };
```

```
   char shaps[] = { 'X', 'O', 'S' };

   cout << fills[ card.fill ] << ' ';
   for (int j=0; j < 3; j = j + 1)
      if (j <= card.numb)
         cout << shaps[ card.shap ];
      else
         cout << ' ';
   cout << "   ";
}
```

Increment 7. Significant user interaction is being added in this step, so the drawing of the 12 "displayed cards" has been encapsulated in `draw_board()`. Notice that the "4 cards per line" logic has migrated to this new function [13], and an "Input:" prompt has been added [14].

```
struct Card { ... };

void initialize_cards( Card deck_of_cards[] ) { ... }

void draw_board( Card displayed_cards[] ) {
   cout << '\n';
   for (int i=1; i <= 12; i = i + 1) {
      draw_card( displayed_cards[i-1] );
      if (i % 4 == 0)                              // [13]
         cout << "\n";
      else
         cout << "   ";
   }
   cout << "\nInput: ";                            // [14]
}
```

A new "displayed cards" array has been added [15] and initialized [16]. In an infinite loop, user input of a card number (in the range 1-12) is received [17]. To designate missing cards, the "numb" member of `Card` has been hijacked and assigned a special value (i.e. "-1") at [18], and the use of that value appears in `draw_card()` [19].

```
void draw_card( Card card ) {
   if (card.numb == -1) {                          // [19]
      cout << "     ";
      return;
   }
   ...
}

int main( void ) {
   Card deck_of_cards[27];
   Card displayed_cards[12];                       // [15]
   initialize_cards( deck_of_cards );
   for (int i=0; i < 12; i = i + 1)
      displayed_cards[i] = deck_of_cards[i];       // [16]
   int input;
   while (true) {
      draw_board( displayed_cards );
      cin >> input;                                // [17]
      if (input > 0  &&  input < 13)
```

```
            displayed_cards[input-1].numb = -1;    // [18]
    }  }
```

Increment 8. Let's add a "fill" command that can move cards from the `deck_of_cards` array to the `displayed_cards` array until no more cards remain.

```
. X     . XX    . XXX   . O
. OO    . OOO   . S     . SS
. SSS   + X     + XX    + XXX

Input: 1

        . XX    . XXX   . O
. OO    . OOO   . S     . SS
. SSS   + X     + XX    + XXX

Input: 2

                . XXX   . O
. OO    . OOO   . S     . SS
. SSS   + X     + XX    + XXX

Input: 3

                        . O
. OO    . OOO   . S     . SS
. SSS   + X     + XX    + XXX

Input: fill

+ O     + OO    + OOO   . O
. OO    . OOO   . S     . SS
. SSS   + X     + XX    + XXX
```

The application now needs to accept heterogeneous input from the user: the string "fill", or a number between 1 and 12. The variable `input` has been changed from an `int` to a `string` [19]. If the string "fill" is received [20], then each element of the `displayed_cards` array is evaluated [21], and each missing card is replaced by the "next" card in `deck_of_cards`. The `next_card` variable was added at [22].

If the input from the user was not the "fill" command, then the input must represent the number of a card. The `string` variable `input`, is converted to an `int` variable at [23].

```
struct Card { ... };

void initialize_cards( Card deck_of_cards[] ) { ... }

void draw_card( Card card ) { ... }

void draw_board( Card displayed_cards[] ) { ... }

int main( void ) {
   Card deck_of_cards[27];
   Card displayed_cards[12];
   initialize_cards( deck_of_cards );
```

```
   for (int i=0; i < 12; i = i + 1)
      displayed_cards[i] = deck_of_cards[i];
   int next_card = 12, card;                                  // [22]
   string input;                                              // [19]
   while (true) {
      draw_board( displayed_cards );
      cin >> input;
      if (input == "fill") {                                  // [20]
         if (next_card == 27) {
            cout << "\nNO MORE CARDS\n";
            continue;
         }
         for (int i=0; i < 12; i = i + 1)                     // [21]
            if (displayed_cards[i].numb == -1) {
               displayed_cards[i] = deck_of_cards[next_card];
               next_card = next_card + 1;
            }
         continue;
      }
      stringstream ss;
      ss << input;                                            // [23]
      ss >> card;
      if (card > 0  &&  card < 13)
         displayed_cards[card-1].numb = -1;
} }
```

Increment 9. How shall we "shuffle" or randomize the deck of cards? One strategy might be: divide the deck of cards into 2 decks, and interleave the cards back into a single deck (like humans shuffle a deck of cards). Another strategy might be: pull a random card out of the deck and place it in a new stack until no cards remain in the original stack. A third strategy might be: pick two random cards from the deck and swap them, and repeat 50/100/200 times.

The third strategy has been implemented below. The first random card is "saved" [24], the second random card is moved to the first card's position [25], and the saved card is moved to the second card's position [26].

```
void initialize_cards( Card deck_of_cards[27] ) {
   // Initialize the deck of cards
   ...
   // Shuffle the deck of cards
   srand( time( 0 ) );
   Card temp;
   for (int i=0,j,k; i < 100; i = i + 1) {
      j = rand() % 27;
      k = rand() % 27;
      temp             = deck_of_cards[j];       // [24]
      deck_of_cards[j] = deck_of_cards[k];       // [25]
      deck_of_cards[k] = temp;                   // [26]
} }
```

Increment 10. We are now ready for the application to accept three card numbers from the user, and evaluate if they represent a set. Earlier, we saw the following attribute value combinations. If you add up each column, is there something useful that can be gleamed about the "no" columns versus the "yes" columns?

```
            0   0   1   1   2   2   0   0   1   2
            0   1   1   2   2   0   1   0   1   2
            1   1   2   2   0   0   2   0   1   2
Is a set?  no  no  no  no  no  no  yes yes yes yes
```

All the "not a set" columns are not evenly divisible by 3, and the "is a set" columns **are** evenly divisible. If the sum of the number attributes – and – the sum of the shape attributes – and – the sum of the fill attributes, are all evenly divisible by 3; then the 3 cards represent a set.

```
bool is_a_set( Card one, Card two, Card three ) {
   if ((one.numb + two.numb + three.numb) % 3 != 0) return false;
   if ((one.shap + two.shap + three.shap) % 3 != 0) return false;
   if ((one.fill + two.fill + three.fill) % 3 != 0) return false;
   return true;
}
```

The goal for the user interface is to accept three integers in the range 1 to 12 [27], remove the corresponding cards if they represent a set [28], and report the running total of sets found [29].

```
+ X     ^ OO    + XX    + SS
. XX    + O     . X     + OOO
^ SSS   . O     + SSS   ^ S

sets found - 0
Input: 2 8 10                               // [27]

+ X             + XX    + SS                // [28]
. XX    + O     . X
^ SSS           + SSS   ^ S

sets found - 1                              // [29]
Input:
```

In the previous increment, user input was read into a string variable, and subsequently converted to an integer. In this increment, the conversion is to three integers [30]. Validation of the user input has been expanded to deal with 3 cards [31]. The 3 cards are then passed to `is_a_set()`, and its return value evaluated [32]. If the cards represent a set, then all 3 are "removed" [33] and a new `sets_found` variable is incremented. `sets_found` is now passed to `draw_board()` so that it can be reported to the user.

```
struct Card { ... };

void initialize_cards( Card deck_of_cards[] ) { ... }

void draw_card( Card card ) { ... }

void draw_board( Card displayed_cards[], int sets_found ) {     // [34]
   ...
   cout << "\nsets found - " << sets_found << "\nInput: ";
}

int main( void ) {
   ...
   int next_card = 12, sets_found = 0, first, second, third;
   string input;
```

```
   while (true) {
      draw_board( displayed_cards, sets_found );                // [34]
      getline( cin, input );
      if (input == "fill") {
         ...
      }
      stringstream ss;
      ss << input;
      // Parse and validate the input
      ss >> first >> second >> third;                           // [30]
      if (first < 1 || first > 12 || second < 1 || second > 12
            || third < 1 || third > 12
            || displayed_cards[first-1].numb == -1              // [31]
            || displayed_cards[second-1].numb == -1
            || displayed_cards[third-1].numb == -1) {
         cout << "\nBAD INPUT\n";
         continue;
      }
      if ( ! is_a_set( displayed_cards[first-1],
                       displayed_cards[second-1],
                       displayed_cards[third-1] )) {            // [32]
         cout << "\nNOT a set\n";
         continue;
      }
      displayed_cards[first-1].numb  = -1;                      // [33]
      displayed_cards[second-1].numb = -1;
      displayed_cards[third-1].numb  = -1;
      sets_found = sets_found + 1;
} }
```

Increment 11. It would be nice if the Set Game application could report to the user how many sets are present in the currently displayed cards, and then list those sets on demand.

```
. SSS    ^ XX     ^ XXX    + XXX
. O      + SS     + OO     . X
^ O      + SSS    + XX     . OO

sets present 8, sets found 0
Input: answer

answer 1 8 12   1 9 11   2 4 8   2 5 10   2 6 12   3 5 6
       3 8 11   6 7 11
```

In the previous chapter, it was argued that the first step to this end would be to enumerate all combinations of 12 cards taken three at a time. Here are the first and last several lines of such an enumeration.

```
1.2.3 1.2.4 1.2.5 1.2.6 1.2.7 1.2.8 1.2.9 1.2.10 1.2.11 1.2.12
1.3.4 1.3.5 1.3.6 1.3.7 1.3.8 1.3.9 1.3.10 1.3.11 1.3.12
1.4.5 1.4.6 1.4.7 1.4.8 1.4.9 1.4.10 1.4.11 1.4.12
1.5.6 1.5.7 1.5.8 1.5.9 1.5.10 1.5.11 1.5.12
1.6.7 1.6.8 1.6.9 1.6.10 1.6.11 1.6.12
1.7.8 1.7.9 1.7.10 1.7.11 1.7.12
1.8.9 1.8.10 1.8.11 1.8.12
1.9.10 1.9.11 1.9.12
```

```
1.10.11 1.10.12
1.11.12
...
8.9.10 8.9.11 8.9.12
8.10.11 8.10.12
8.11.12
9.10.11 9.10.12
9.11.12
10.11.12
```

What patterns can you identify in the listing? What do you think about using three loops: an outer loop, a middle loop, and an inner loop? If all three loops went from 1 to 12, then combinations like 1.1.1, 4.5.4, and 8.9.9 would be generated; and having a card appear more than once is not desirable.

What about having: the outer loop go from 1 to 10, the middle loop go from i+1 to 11, and the inner loop go from j+1 to 12? The following demo generates the correct number of combinations that was computed in the previous chapter.

```
int count = 0;
for (int i=0;          i < 10;  i = i + 1)
   for (int j=i+1;     j < 11;  j = j + 1)
      for (int k=j+1;  k < 12;  k = k + 1)
         count = count + 1;
cout << "count is " << count << '\n';

// count is 220
```

Our new function `compute_sets_present()` needs to accept the displayed cards array, and a string in which to store the list of sets present; and return the count of those sets. When a 3-card combination is generated, if one of the cards is missing, then skip that combination [35]. The `is_a_set()` function developed in the previous increment can then be invoked, and the `number_of_sets` and `answer` variables updated as appropriate [36].

```
int compute_sets_present( Card displayed_cards[], string& answer_str ) {
   int number_of_sets = 0;
   stringstream answer;
   for (int i=0;   i < 12 - 2; i = i + 1)
      for (int j=i+1; j < 12 - 1; j = j + 1)
         for (int k=j+1; k < 12;     k = k + 1) {
            if (displayed_cards[i].numb == -1
                  || displayed_cards[j].numb == -1
                  || displayed_cards[k].numb == -1)
               continue;                                          // [35]
            if (is_a_set( displayed_cards[i], displayed_cards[j],
                          displayed_cards[k] )) {
               number_of_sets = number_of_sets + 1;               // [36]
               answer << (i+1) << ' ';
               answer << (j+1) << ' ';
               answer << (k+1) << "   ";
         }  }
  answer_str = answer.str();
  return number_of_sets;
}
```

Let's report "sets present" in `draw_board()`. Since `compute_sets_present()` generates that value, it will be invoked in `draw_board()` [37].

```
//              in argument             in argument     out argument
void draw_board(Card displayed_cards[], int sets_found, string& answer) {
   ...
   cout << "\nsets present "
        << compute_sets_present( displayed_cards, answer )        // [37]
        << ", sets found " << sets_found << "\nInput: ";
}
```

The user can now ask for the "answer" to be displayed [38]. The "answer" string is reported in `main()` [39], but it is generated in `compute_sets_present()`. How to get that string from where it's generated to where its needed? The return value for `compute_sets_present() is` already spoken for. So – an "output" argument needs to be added to `draw_board()` [40] and then to `compute_sets_present()` as well [37]. The `answer` variable originates in `main()`, gets passed down to where it is assigned, and returned to where it is needed.

```
int main( void ) {
   ...
   string input, answer;
   while (true) {
      draw_board( displayed_cards, sets_found, answer );          // [40]
      getline( cin, input );
      if (input == "answer") {                                    // [38]
         cout << "\nanswer " << answer << '\n';                    // [39]
         continue;
      }
      ...
}  }
```

Appendix F ... Algorithms

Grid Game

How to compute "delta X plus delta Y" (increment 3)

Fifteen Puzzle

How to "move" a number (increment 4)

How to mix up the numbers to start the puzzle (increment 5)

Mastermind

How to compute the "totally correct" score

How to compute the "partially correct" score

Peg Game

How to output/format a board that looks like a bowling pin lay-out (increment 2)

How to validate and implement the notion of "jump" (increment 5)

Set Game

How to initialize 27 Cards (increment 3)

How to shuffle (randomize) the deck of Cards (increment 9)

How to identify if three Cards form a set (increment 10)

How to compute all possible sets present in a collection of 12 Cards (increment 11)

Appendix G ... Data Structures

Grid Game

How to model and maintain the "state" of the game (increment 2)

Fifteen Puzzle

How to maintain a mapping between "number" and "position" (increment 2)

Mastermind

Are there auxiliary data structures needed to compute the "partially correct" count? (increment 6)

Peg Game

How to maintain the state of the game's user interface (increment 3)

How to validate/implement a jump request. Is it algorithm or data structure? Int[36][3] (incr 5)

Set Game

How to model/implement the abstraction of "Card". int[27][3] then struct[27] (increment 3)

How to designate missing cards (increment 7)

Appendix H ... Precedence of Operators

Operators	Associativity
`()` `[]` `->` `.`	left to right
(unary operators) `+` `-` `!` `~` `++` `*` `&` (cast) sizeof	right to left
(binary operators) `*` `/` `%`	left to right
(binary operators) `+` `-`	left to right
`<<` `>>`	left to right
`<` `<=` `>` `>=`	left to right
`==` `!=`	left to right
`&`	left to right
`^`	left to right
`|`	left to right
`&&`	left to right
`||`	left to right
`?:`	right to left
`=` `+=` `-=` `*=` `/=` `%=` `&=` `^=` `|=` `<<=` `>>=`	right to left
`,`	left to right

Alphabetical Index


Set Game
Sets found: 0
Sets present: 4
Refill Cards
Show Answer
uzzle
1 2 14 3
12 4 8
9 11 6 7
15 5 10 13
Grid Game
8
16
0
8
2
SCRABBLE