Applications of Control Flow Mechanisms

In this section we will examine some basic *algorithms* that employ the *while* and *if* control mechanisms. Important applications include combinations of:

Counting
Accumulation
Searching
Recurrences

Examples will be based upon a common design:

Initialize program state and read first value
While (not end-of file)
    update program state as needed
    read next value
Output final state

The type of *state* that must be maintained by the program that must be maintained by the program is dependent on the nature of the problem and can include:

counters
sums
previous input values
indicator (true/false) variables

Searching and Counting: Find the number of 3's in an input file.

In this program the *state* that must be maintained is the number of 3's seen so far. It is initialized to have the value 0 and incremented each time a 3 is encountered in the file.

```
/* p5.c */
#include <stdio.h>

int main()
{
   int counter;    // the number of three's we've seen
   int value;      // the value just read
   int howmany;    // howmany values were read

   counter = 0;
   howmany = fscanf(stdin, "%d", &value);

   while (howmany == 1)
   {
      if (value == 3)
         counter = counter + 1;

      howmany = fscanf(stdin, "%d", &value);
   }
   fprintf(stdout, "The number of 3's was %d \n", counter);
   return(0);
}
```

```
Sample input: 1 2 3 2 1 3 4
Sample output: 2

Sample input: 3 3 3
Sample output: 3
```

Searching and Counting: Find the smallest value in a file containing a collection of integers.

Here the *state* that must be maintained is the smallest value seen so far. It is tempting to initialize this to a *large* number. However, that approach is not a good idea. The proper way to handle problems of this sort is to initialize the *state* variable *minval* to the *first value in the file.*

```
/* p11.c */
#include <stdio.h>

int main()
{
   int howmany;    // how many integers were read
   int minval;     // the minimum value in the file
   int value;      // current value

   howmany = fscanf(stdin, "%d", &value);
   minval = value;

   while (howmany == 1)
   {
      if (minval > value)
         minval = value;

      howmany = fscanf(stdin, "%d", &value);
   }

   fprintf(stdout, "%d\n", minval);
   return(0);
}
```

```
Sample input: 4 -100 5
Sample output: -100

Sample input: 10000 100000 200000
Sample output: 10000
```

Accumulation: Find the sum of all of the single digit numbers in a file

Here the program state that must be maintained is the *sum* of all values between (-10 and 10) that have seen so far. As each new value is obtained, it must be tested to see if it is a single digit and if so its value is added to the current value of *sum.*

```
/* p6.c */
#include <stdio.h>

int main()
{
   int sum;         // the sum of the single digit numbers
   int value;       // the value just read
   int howmany;     // howmany values were read

   sum = 0;
   howmany = fscanf(stdin, "%d", &value);

   while (howmany == 1)
   {
      if ((value > -10) && (value < 10))
         sum = sum + value;
      howmany = fscanf(stdin, "%d", &value);
   }

   fprintf(stdout, "The sum was was %d \n", sum);
   return(0);
}
```

```
Sample input: -11 -12 -1 15 2 7
Sample output: 8

Sample input: 14 12 99
Sample output: 0
```

Searching: Write a program that reads a collection of integers from the standard input. If the collection contains a value of 13 print "yes" to the standard output. Otherwise print "no".

Unlike previous examples the state that must be maintained here is an indicator (*true/false)* value that defines whether or not the program has encountered a value of 13 in its input. We use the standard C language convention of representing *false* with a value of 0 and *true* with a value of 1.

```
/* p9.c */
#include <stdio.h>

int main()
{
   int found13;    // 1 => found a 13
   int value;      // the value just read
   int howmany;    // howmany values were read

   found13 = 0;    // haven't found it yet
   howmany = fscanf(stdin, "%d", &value);

   while (howmany == 1)
   {
      if (value == 13)
         found13 = 1;
      howmany = fscanf(stdin, "%d", &value);
   }

   if (found13)
      fprintf(stdout, "yes\n");
   else
      fprintf(stdout, "no\n");
   return(0);
}
```

We could also say
`if (found13 == 1)`
here.

```
Sample input:
1 2 4 77 12 13
Sample output:
yes
```

Searching: Write a program that reads a collection of integers from the standard input. If the collection contains a value of 13 followed by a 14 and then by 15 print "yes" to the standard output. Otherwise print "no".

The *state* that must be maintained by this program is more complex than that of the previous one. As in the last program, an indicator of whether the subsequence {13, 14, 15} has or has not been seen yet must be maintained. However, it is clearly *not possible to determine if such is the case by looking at only the current input value.* If the current input value is 15, the program must also know whether the previous input value was 14 and the one before that was 13. Therefore, it is necessary to always remember the two previous values. The variables *old* and *older* are used for that purpose. Initialization is also more complicated as all three value holders must be initialized.

```
/* p10.c */
#include <stdio.h>

int main()
{

   int older;          // value before that
   int old;            // previous value
   int value;          // current value
   int howmany;        // howmany values were read
   int found = 0;      // found target

   howmany = fscanf(stdin, "%d", &older);
   howmany = fscanf(stdin, "%d", &old);
   howmany = fscanf(stdin, "%d", &value);
```

What happens here if the input file contains only 1 or 2 values? Should *howmany* be tested after each call to *fscanf()??*

The main loop illustrates two important programming techniques:

- the *if* statement with *compound* condtions (always use parentheses)
- a *window* mechanism for discarding the oldest value and updating *older, old,* and *value*.

```
   while (howmany == 1)
   {
      if ((older == 13) &&
          (old   == 14) &&
          (value == 15))
      {
         found = 1;
      }

      older = old;
      old = value;
      howmany = fscanf(stdin, "%d", &value);
   }

   if (found)
      fprintf(stdout, "yes\n");
   else
      fprintf(stdout, "no\n");
   return(0);
}
```

The order in which these assignments are made is critical to correctness.

```
Sample input: 11 12 13 15 12
no
Sample input: 14
no
```

Programs with no input

The last two examples that we consider in this section are those in which the program has no input at all! In both cases the set of all non-negative integers is implicitly the input and the problem that we are attempting to solve is to find a subset of the integers that have a particular property.

Searching and enumerating: Print all of the integers less than or equal to 10000 that are perfect squares.

This is a problem in which it pays to engage the brain before engaging the fingers. The first instinct of most people is to try to take the square root of all integers between 1 and 10000 and see if the value is an integer. However, a better approach is to just compute them all directly. In any problem of *enumeration* (print all of the numbers that....) it will be necessary to have *fprintf()* within the main loop (possibly guarded by an *if*. This is in contrast to previous problems in which we printed only the final state of the program at the end.

```
/* p8.c */
#include <stdio.h>

int main()
{
   int val;       // the current value in [1, 10000]
   int valsqr;    // square of the value
   val = 1;
   valsqr = val * val;

   while (valsqr <= 10000)
   {
      fprintf(stdout, "%d\n", valsqr);

   /* Compute next value and next square */

      val = val + 1;
      valsqr = val * val;
   }
}
Sample output
1
4
9
16
:
10000
```

Recurrences: Suppose the first two numbers of a sequence of numbers are {0, 1}. Suppose each subseqent number is the sum of its two immediate predecessors. We manually compute a few terms of the sequence as follows:

0 + 1 = 1 -> {0, 1, 1}
1 + 1 = 2 -> {0, 1, 1, 2}
1 + 2 = 3 -> {0, 1, 1, 2, 3}
2 + 3 = 5 -> {0, 1, 1, 2, 3, 5}
3 + 5 = 8 -> {0, 1, 1, 2, 3, 5, 8}

Problem: Print the first twenty terms of this sequence

Here we need to remember the strategy from the {13, 14, 15} problem in which we learned how to "remember" a window of values.

```
/* p7.c */
#include <stdio.h>

int main()
{
   int old;     // most recent old value;
   int older;   // less recent old value;
   int new;     // new value;
   int counter; // number of values printed so far

   older = 0;
   fprintf(stdout, "%d\n", older);

   old = 1;
   fprintf(stdout, "%d\n", old);

   counter = 2;
   while (counter < 20)
   {
      new = old + older;
      fprintf(stdout, "%d\n", new);
      counter = counter + 1;
      older = old;
      old = new;
   }
}
```

Why not use
`while (counter <= 20)`
here

1. Write a program that reads one integer value at a time from the standard input. If the collection of input integers contains at least one 7 and at least one 11, the program should write "yes" (without the "s) to the standard output. Otherwise it should write "no" to the standard output.

2. Write a program that reads one integer value at a time from the standard input. If the collection of input integers contains a 7 followed immediately by an 11, the program should write "yes" (without the "s) to the standard output. Otherwise it should write "no" to the standard output.

3. Write a program that reads PAIRS of integers from the standard input. If the sum of the two integers in a pair is 10 then the program should print the pair to the standard output.

4. Write a program that reads PAIRS of integers from the standard input. The program should print the number of pairs whose sum is 20 to the standard output

5. Write a program that reads PAIRS of integers from the standard input. The program should compute sum of each pair and print the largest sum found to the standard output.

Other control mechanisms

The *for* loop

for (init-expression; continue-condition; update-expression)
{
loop-body
}

The *init-expression* is executed one time
The *continue-condition* is evaluated each iteration of the loop *before* the *loop-body* is executed.
The *update-expression* is executed after the *loop-body* is executed.
The *loop-body* is executed if and only if the *continue* condition is true.

```
#include <stdio.h>

int main()
{
   int i;

   for (i = 5; i < 5; i = i + 1)
      printf("in body with i = %d \n", i);

   printf("at end with i = %d \n", i);

   for (i = 3; i < 5; i = i + 1)
      printf("in body with i = %d \n", i);

   printf("at end with i = %d \n", i);
}
```

```
acad/cs101/examples/notes ==> a.out
at end with i = 5

in body with i = 3
in body with i = 4
at end with i = 5
```

The *do ... while()* loop

do
{
loop-body;
} while (continue-condition);

This structure is similar the the *while ()* loop, but unlike, the while loop,

- the *loop-body* will always be executed at least one time.
- a semicolon *must* follow *while (continue-condition);*

```
#include <stdio.h>

int main()
{
   int i = 15;

   do
   {
      printf("in body with i = %d \n", i);
      i  = i + 1;
   }  while (i < 10);
}
```

```
acad/cs101/examples/notes ==> a.out
in body with i = 15
```

Altering control flow within a loop

Two single word statements may be used to alter control flow within a loop:

The *break* statement

- exit the loop containing the break *immediately*
- execution continues at the line immediately following the break

#include <stdio.h>

```
int main()
{
   int i;
   int j;

   for (i = 0; i < 3; i = i + 1)
   {
      for (j = 0; j < 3; j = j + 1)
      {
         printf("in body with (i, j) = (%d, %d)\n",
                                                i, j);
         if (i == j)
            break;
      }
   }
}
```

```
acad/cs101/examples/notes ==> a.out
in body with (i, j) = (0, 0)
in body with (i, j) = (1, 0)
in body with (i, j) = (1, 1)
in body with (i, j) = (2, 0)
in body with (i, j) = (2, 1)
in body with (i, j) = (2, 2)
```

Use of *break* is discouraged as proper design can almost always produce an equally simple implementation with requiring *break*.

Here is a an example of an equivalent program that does not contain the break;

```
int main()
{
   int i;
   int j;

   for (i = 0; i < 3; i = i + 1)
   {
      for (j = 0; j <= i; j = j + 1)
      {
         printf("in body with (i, j) = (%d, %d)\n",
                                                i, j);

      }
   }
}
```

The *continue* statement

- causes a transfer of control to the end of the loop bypassing the remainder of the statements in the body of the loop.

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

int main()
{
   float val;
   float sqrtval;
   float sum;

   while (fscanf(stdin, "%f", &val) == 1)
   {
      printf("\n%8.2f ", val);

      if (val < 0)
         continue;

      sqrtval = sqrt(val);
      sum = sum + sqrtval;
      printf(" %8.2f %8.2f ", sqrtval, sum);
   }
}
```

```
    1.00      1.00     1.00
    2.00      1.41     2.41
    4.00      2.00     4.41
   -4.00
    5.00      2.24     6.65
```

As with the *break* statement, it is almost always the case that use of the *continue* statement can always be avoided by careful design.

The *switch* statement

Similar in function to the *if , else if, else if* construct.

```
switch (expression)
{
        case const1:
                statement1;
                statement2;
                break;
        case const2:
                statement3;
                break;
        default:
                statement4;
}
```

This is equivalent to:

```
if (expression == const1)
{
        statement1;
        statement2:
}
else if (expression == const2)
{
        statement3;
}
else
{
        statement4;
}
```

Differences

with if/else *const1 and const2* may be *expr1* and *expr2*
the break statement may be removed in the switch to allow *fall-through.*

```c
#include <stdio.h>

int main()
{
   int area;
   int sum864 = 0;
   int sum803 = 0;
   int other  = 0;

   while (scanf("%d", &area) == 1)
   {
      switch (area)
      {
      case 803:
         sum803 += 1;
         break;
      case 864:
         sum864 += 1;
         break;
      default:
         other += 1;
      }
   }
}
```

The *question mark* operator

condition ? true-expression: false-expression

The condition is evaluated and if true *true-expression* will be executed and the value of the entire expression will be the value of the *true-expression.*
Otherwise the *false-expression will be executed* and the value of the entire expression will be the value of the *false-expression.*

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

int main()
{
   float val;
   float sqrtval;
   float sum;

   while (fscanf(stdin, "%f", &val) == 1)
   {
      printf("\n%8.2f %8.2f ",
                  val, val > 0 ? sqrt(val): 0.0);

   }
}
```

```
    1.00     1.00
    2.00     1.41
    4.00     2.00
   -4.00     0.00
    5.00     2.24
```

The *comma* operator

e1, e2

This operator can be used when it is desired to place two (or more) expressions where one is normally used. The value of the expression is the value of *e2*.

```
for (i = 0, j = n - 1; i < n; i++, j--)
{
    out[j] = in[i];
}
```

```
#include <stdio.h>

int in[10];
int out[10];

int main()
{
   int i, j;
   int k;
   int m;

   for (i = 0, j = 9; i < 10; m++, k++, i++, j--)
      out[j] = in[i];
}
```