Programming Strategies Reference

Michael de Raadt

Introduction

This appendix contains a number of useful strategies relevant to an introductory programming course, but also necessary to solve problems of a more complex nature. The list is not complete, but contains strategies that are well defined and malleable enough be manipulated to suit particular problems.

This appendix should be seen as a tool-kit for solving problems at a sub-algorithmic level. The plans at this scale usually do not constitute an entire algorithm (although some approach this level) but usually form part of a greater algorithm.

This reference is not meant to be a complete curriculum; it is merely a short reference guide.

Certain programming language knowledge (constructs and functions) are required before each plan can be applied. These dependencies are listed *in italics* at the beginning of each plan.

Table of Contents

Plan Integration

Before introducing the plans, it is important to discuss how plans can be integrated into a whole solution. There are three ways of combining plans.

Abutment

Abutment is placing plans or steps within plans one after the other. The sequence of these defines the necessary order that must be followed to be successful. For example, if we wish to perform calculations on user inputs, we must first get the inputs before we can perform the calculation.

Merging

Often two plans need to be achieved together. Step within the two plans may be intertwined in their order so that they can be achieved together. A processor can only achieve one instruction at a time so these steps cannot be achieved simultaneously, but the steps can be placed one after another in arbitrary order. For example, if we were wishing to calculate an average of a set of numbers we need to count the numbers and sum the numbers. Rather than inputting and processing the set of numbers twice, we can merge these two plans and achieve them together.

Nesting

Where one plan is contained within another, the inner plan is said to be nested inside the outer plan. For example, if we were summing numbers we may nest the summing plan within one of the specific looping plans. If we were to calculate an average, we may nest this within a Guarded Division plan to avoid division by zero in the average calculation.

Plan 1. Average Plan

This plan requires an understanding of the division operator.

Finding the average of a series of numbers is a common task in programming. To calculate the average we need the sum of the numbers and the count of the numbers. Assuming we have these two values we calculate the average by dividing the sum by the count.

```
average = sum / count
```

Here is an example in the context of a full program.

```
#include <stdio.h>

int main() {
      int sum = 15;   // Stores the some of some numbers
      int count = 3;  // Stores the count of those numbers
      int average;    // Will store the calculated average

      // Calculate the average
      average = sum / count;

      // Output the average
      printf("Average: %i\n", average);
}
```

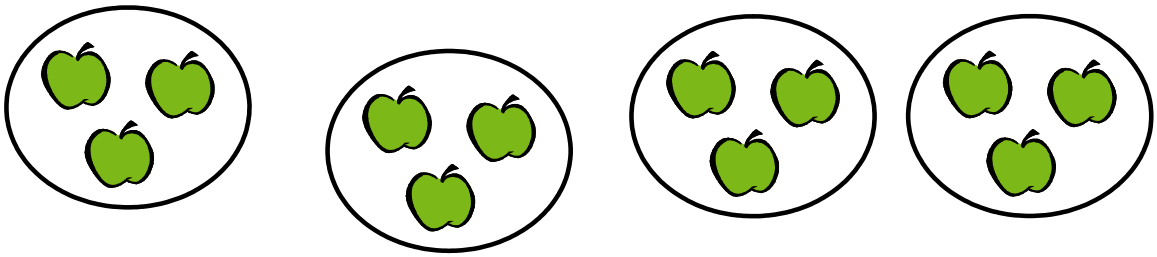
Here is the output of the above program.

```
Average: 5
```

Plan 2. Divisibility Plan

This plan requires an understanding of the mod operator and selection statements.

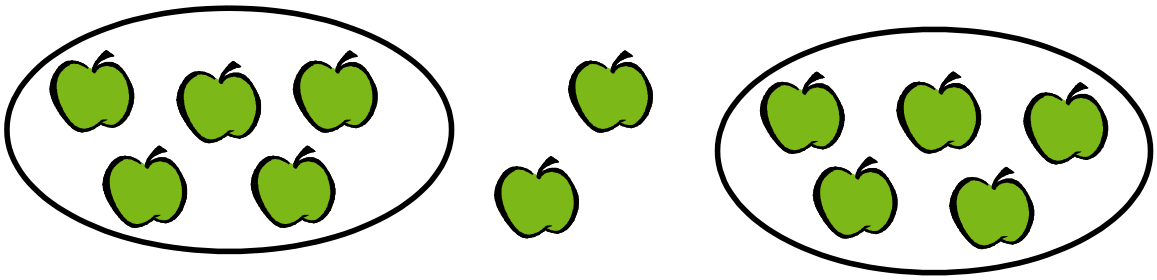
If we wish to see if one number is evenly divisible by another, we can use the mod operator. If this operator produces a result of zero we know that the first operand is divisible by the second. The mod operator gives us the remainder after division. If there is no remainder we know that the first operand is divisible by the second. In a real world application, if we were to group objects, say apples, we may wish to know if we can form complete groups from the number of apples at hand. If we have 12 apples we can divide this into 4 groups of 3 with no remainder.



We can apply the same to numbers in code, for example…

`12 % 3` results in `0` so we can say `12` is divisible by `3`

We can also see when a number is not divisible by another. If we group 12 apples in to groups of 5 we are left with 2 apples remaining.



Again we can apply the same to numbers in code, for example…

`12 % 5` results in `2` so we can say `12` is not divisible by `5`

Here is an example in the context of a full program.

```
#include <stdio.h>

int main() {
      int numberToCheck = 12;  // A number to check for divisibility
      int firstDivisor = 3;    // A sample divisor to use
      int secondDivisor = 5;   // Another sample divisor to use
      int result;              // Will store the result of mod operation

      // Check the divisibility using first divisor
      result = numberToCheck % firstDivisor;
      printf("Result using %i: %i\n", firstDivisor, result);

      // Check the divisibility using second divisor
      result = numberToCheck % secondDivisor;
      printf("Result using %i: %i\n", secondDivisor, result);
}
```

Here is the output of the above program.

```
Result using 3: 0
Result using 5: 2
```

The above results show that 12 is divisible by 3 but 12 is not divisible by 5.

Here is a program that tests if numbers are even. An even number is divisible by two.

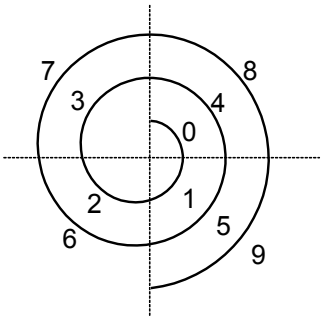
```
#include <stdio.h>

int main() {
      int firstNumberToCheck  = 4;   // Number to check divisibility by 2
      int secondNumberToCheck = 5;   // Another "

      // Check if first number is even
      if(firstNumberToCheck%2 == 0) {
            printf("%i is even\n", firstNumberToCheck);
      }
      else {
            printf("%i is not even\n", firstNumberToCheck);
      }

      // Check if second number is even
      if(secondNumberToCheck%2 == 0) {
            printf("%i is even\n", secondNumberToCheck);
      }
      else {
            printf("%i is not even\n", secondNumberToCheck);
      }
}
```

Here is the output of the above program.

```
4 is even
5 is not even
```

Plan 3. Cycle Position Plan

This plan requires an understanding of the mod operator.



It is possible to form a series of numbers into a cycle. Each number will then have a relative position within the cycle. For example we can to take a series of numbers beginning with zero and group them by fours. Each number would then have a relative position within each cycle from zero to three. In the figure above we see such a cycle. The numbers are in four groups and each group has a relative. Numbers with position 0 are { 0, 4, 8, … }, numbers with position 1 are { 1, 5, 9, … } and so on.

We can determine the position of a number in a cycle using the mod operator. As a general rule numbers can be brought into a cycle of size *n* by applying mod *n*.

x % n gives the position of *x* in a cycle of size *n*

For example if we want to create a size 3 we can apply mod 3 and we can then find positions of numbers in this cycle.

```
...
 9 % 3 gives 0
10 % 3 gives 1
11 % 3 gives 2
12 % 3 gives 0 …and so on.
```

One useful application of this idea is to bring random numbers into a range. In the C/C++ language random numbers are generated in a range from 0 to the largest possible integer value (with 4 byte integers this is 2147483647). If we want to generate a random number in a specified range, we can take the random number given by the standard library function **`rand()`** and find its position in a specified cycle.

x % n gives the a value in the range 0 to *n-1*

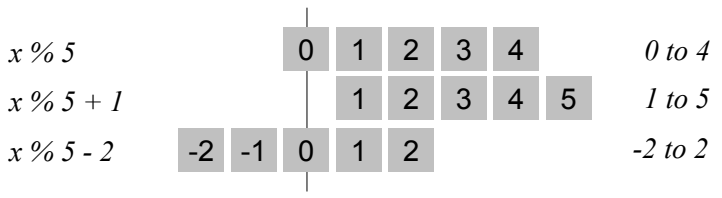
If we wanted to have a random number between 0 and 4 we can apply mod 5.

```
myRand = rand() % 5;
```

If we want a random number between 1 and 5 we can shift the previous range by adding 1 to the result.

```
myRand = rand() % 5 + 1;
```

We can also shift such a range in a negative direction. The diagram below shows a range and how it can be visualised when shifted.



We can create a function that generates a random number between 1 and 10 as follows.

```
int rand1to10() {
      return rand()%10 + 1;
}
```

We can generalise this function to apply settable upper and lower limits.

```
int myRand(int lowerLimit, int upperLimit) {
      return rand()%(upperLimit-lowerLimit+1) + lowerLimit;
}
```

Plan 4. Number Decomposition Plan

This plan requires an understanding of the mod and division operators.

We can use the division and mod operators to tear numbers apart. For example, if we want to find the last two digits of 12345 we can apply mod 100. For decimal digits the following rules apply.

`x % 10`	gives	the last digit
`x % 100`	gives	the last two digits
`x % 1000`	gives	the last three digits
`x % 10000`	gives	the last four digits …and so on.

Applying a similar idea we can discover the first digits of a number using the division operator. Using a 5 digit number, the following rules apply.

`x / 10000`	gives	the first digit
`x / 1000`	gives	the first two digits
`x / 100`	gives	the first three digits
`x / 10`	gives	the first four digits.

To find the third last digit of a decimal number we can apply the following operation.

```
thirdLastDigit = x % 1000 / 100;
```

Plan 5. Initialisation Plan

This plan requires an understanding of variables and the assignment operator.

Initialisation is commonly applied within other plans.

Failing to initialise variables before they are used can lead to errors.

It is recommended that you initialise all variables when you declare them.

In the following example **`sum`** is initialised to 0 as this is an appropriate sum before summing commences.

```
int sum = 0;
```

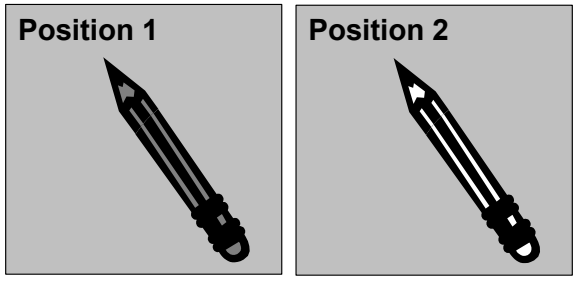
In some plans it may be necessary to initialise an array of items. For instance, here we are initialised an array used to tally letters in a message.

```
#include <stdio.h>

int main() {
      int letterCount[26]; // Array to store count of letters
      int i;               // Iterative counter

      // Initialise array of counts
      for(i=0; i<26; i++) {
            letterCount[i] = 0;
      }

      ...
}
```

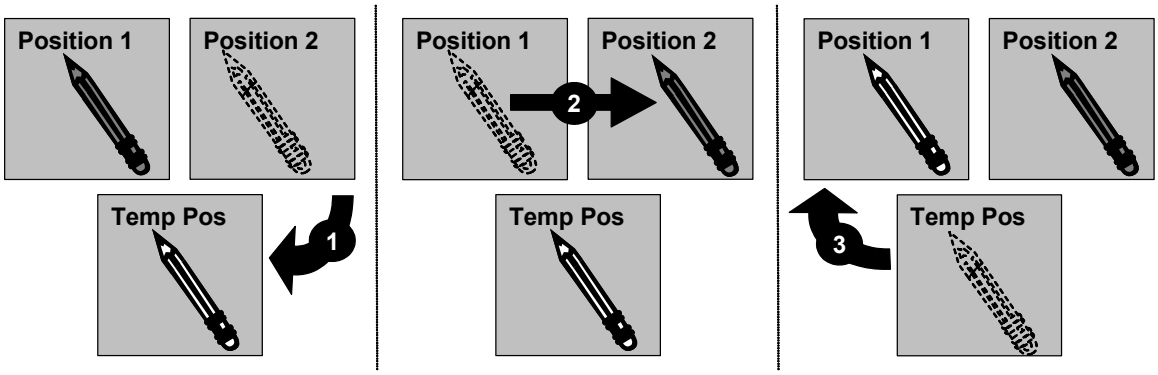
Plan 6. Triangular Swap Plan

This plan requires an understanding of variables and the assignment operator.

Consider how you swap two items. Imagine two pencils in front of you. To swap their positions you would pick up one with one hand, the second with your other hand and then place each in their new positions.



A computer can only perform one action at a time. Now imagine that you only have one hand; how would you swap the positions of the two pencils now? Keep in mind also that when a variable is assigned a new value, the old value is replaced and cannot be accessed later. Attempting to swap using the above method will result in two copies of the same value.



To achieve a swap a temporary position is needed. One of the pencils could be moved to the temporary position; the second pencil could be moved to its new location; finally the first pencil could be moved from the temporary position to its new position.

Here is an example in the context of a full program.

```
#include <stdio.h>

int main() {
      int firstPosition  = 5; // First position containing value to swap
      int secondPosition = 6; // Second position containing value to swap
      int tempPosition;       // Temporary position for swap

      // Output the numbers after the swap
      printf("Before Swap...\n");
      printf("First: %i, Second: %i\n", firstPosition, secondPosition);

      // Swap the two numbers in a triangular swap
      // 1. Copy the value from the second position to temp
      tempPosition = secondPosition;

      // 2. Copy the value from the first position to the second
      secondPosition = firstPosition;

      // 3. Copy the value from the temp position to the first
      firstPosition = tempPosition;

      // Output the numbers after the swap
      printf("After Swap...\n");
      printf("First: %i, Second: %i\n", firstPosition, secondPosition);
}
```

Here is the output of the above program.

```
Before Swap...
First: 5, Second: 6
After Swap...
First: 6, Second: 5
```

The above results show the values are swapped and not duplicated.

Plan 7. Guarded Exception Plans (including Guarded Division Plan)

This plan requires an understanding of the ***`if`*** *statement.*

When a program compiles and runs, there are still opportunities for things to go wrong. Usually such *logic errors* occur around or outside *boundaries* of the data being worked on. Such boundaries include:

- Absence of data where some is expected,
- Negatives or zero where positives are expected,
- Too much data where a finite amount is expected, and
- Values outside an acceptable range.

To create reliable, "bullet proof" programs, these boundary conditions need to be considered.

There are also time where a program may encounter data that, when used in operations, will cause the operating to stop the program.

In mathematics, if a number is divided by zero the result is undefined. If a program attempts to divide by zero, the operating system will close the program down. Whenever we perform a division where the second operand could be zero, we must test the second operand before performing the division and prevent the division from taking place if it is zero.

Here is an example in the context of a full program.

```
int main() {
      int firstOperand;  // First operator for division
      int secondOperand; // Second operator for division

      // Gather inputs for division
      printf("Enter two integers for division: ");
      scanf("%i %i", &firstOperand, &secondOperand);

      // Test second operand
      if(secondOperand != 0) {

            // Perform division
            printf(
                  "%i divided by %i is %i",
                  firstOperand,
                  secondOperand,
                  firstOperand / secondOperand
            );
      }
}
```

Here is the output of the above program when the value 5 is given as the second operand.

```
Enter two integers for division: 10 5
10 divided by 5 is 2
```

When a zero value is given for the second operand, no output is produced and the program ends.

```
Enter two integers for division: 10 0
```

Here is another example that incorporates Guarded Division into a function which calculates an average from a given sum and count.

```
int average(int sum, int count) {

      // Test against dividing by zero
      if(count == 0) {
            return 0;
      }

      // Perform division as normal
      else {
            return sum / count;
      }
}
```

Plan 8. Counter Controlled Loop Plan

This plan requires an understanding of looping constructs.

A Counter Controlled uses a counter variable which is incremented until a set number of repetitions is achieved. The loop will continue regardless of any other event that may occur during repetition.

The following example reads in 10 integers from a user and calculates the sum. The program will continue regardless of what the user inputs. We usually use **`for`** loops to achieve counter controlled loops.

```
#include <stdio.h>

const int NUMBER_OF_INPUTS = 10;

int main() {
      int i = 0;      // Loop iterator
      int sum = 0;    // Sum of numbers input
      int userInput; // Input from user

      // Calculate the sum
      for(i=0; i<NUMBER_OF_INPUTS; i++) {
            printf("Enter a number: ");
            scanf("%i", &userInput);
            sum += userInput;
      }

      // Output the sum
      printf("Sum: %i\n", sum);
}
```

Counter Controlled loops are often used with arrays. When this happens the loop iterator can serve the dual purpose of being an index into the array. For an example of this see the initialisation of an array in Plan 5.

Plan 9. Primed Sentinel Controlled Loop Plan

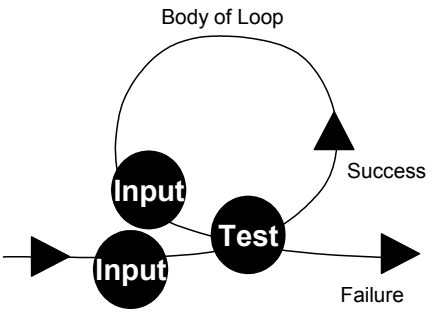
This plan requires an understanding of looping constructs.

A Primed Sentinel Controlled Loop allows repetition until an event takes place or some target value (the sentinel) is discovered.

Here is an example including a primed sentinel-controlled loop. Not that the loop tests **`userInput`** to determine if it should continue looping. The variable is being compared to the sentinel value **`SENTINEL`**. The value of **`userInput`** is primed with an initial user input before the loop begins. Although this adds some redundancy (the input statement appears twice) there can be efficiency savings made when the user enters the sentinel value in the first instance (which is not uncommon).

```
#include <stdio.h>

const int SENTINEL = 9999;

int main() {
      int sum = 0;    // Sum of numbers input
      int userInput; // Input from user

      // Get the first user input
      printf("Enter a number (%i to end): ", SENTINEL);
      scanf("%i", &userInput);

      // Calculate the sum
      while(userInput != SENTINEL) {
            sum += userInput;
            printf("Enter a number (%i to end): ", SENTINEL);
            scanf("%i", &userInput);
      }

      // Output the sum
      printf("Sum: %i\n", sum);
}
```

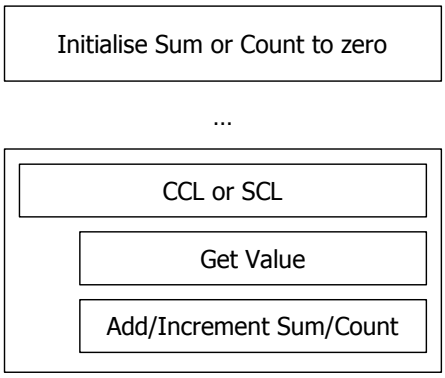
If the user where to enter the sentinel value as their first input, the loop would never be entered. The sum will also be correct as we are checking each user input before it is added to the sum. This avoids accidentally including the sentinel value in the sum.



Plan 10. Sum and Count Plans

This plan requires an understanding of looping constructs and initialization.

Two frequently practiced programming activities are summing or counting values. These simple processes are easily achieved, but also easily messed up. Both plans are achieved by using a variable to accumulate the sum or count as values are encountered. The key to both is assuring that the sum or count variable is initialised to zero. Failing to initialise such a variable will not stop your program from compiling. In many instances an uninitialised variable will have a value of zero so the program will work, but it will not work all the time. Just remember:



INITIALISE SUM AND COUNT VARIABLES

Below is an example which inputs and sums 5 numbers from a user. Note a Counter Controlled loop is used to control repetitions as we know how many are desired before the looping begins.

```
#include <stdio.h>

const int NUMBER_OF_INPUTS = 5;

int main() {
      int userInput = 0; // Input from user
      int sum = 0;       // Sum of inputs INITIALISED
      int i;             // Iterative counter

      // Counter Controlled loop to repeat inputs
      for (i=0; i<NUMBER_OF_INPUTS; i++) {

            // Prompt for input
            printf("Please enter an integer: ");
            scanf("%i", &userInput);

            // Add input to sum
            sum += userInput;
      }

      // Output the sum
      printf("Sum of numbers entered: %i\n", sum);
}
```

The output of the above program will resemble the following.

```
Please enter an integer: 1
Please enter an integer: 2
Please enter an integer: 3
Please enter an integer: 4
Please enter an integer: 5
Sum of numbers entered: 15
```

The following is an example which counts numbers entered by a user unit the value 9999 is encountered as a sentinel.

```
#include <stdio.h>

const int SENTINEL = 9999;

int main() {
      int userInput = 0; // Input from user
      int count = 0;     // Count of inputs INITIALISED

      // Prompt for initial input
      printf("Please enter an integer: ");
      scanf("%i", &userInput);

      // Test for sentinel
      while( userInput != SENTINEL ) {

            // Count input
            count++;

            // Subsequent input
            printf("Please enter an integer: ");
            scanf("%i", &userInput);
      }
      printf("You entered %i inputs\n", count);
}
```

The output of the above program will resemble the following.

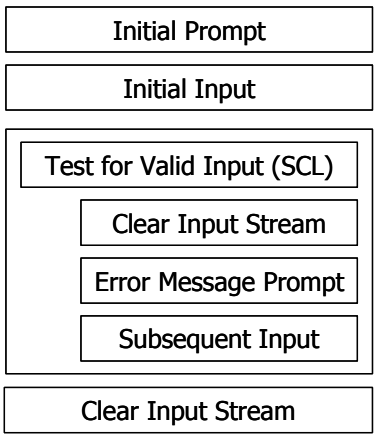
```
Please enter an integer: 1
Please enter an integer: 2
Please enter an integer: 3
Please enter an integer: 9999
You entered 3 inputs
```

Plan 11. Validation Plan

This plan requires an understanding of loops and the ***`scanf()`*** *function (or equivalent).*

When dealing with inputs from users one can never assume they will enter what is expected. It is therefore important, for critical systems, to validate that users have entered what they were expected to enter, and repeat inputs, with appropriate messages, in the case where users enter invalid inputs.

The plan shows here prompts the user and accepts an initial input. The value is then tested as the condition of a Sentinel Controlled loop where the sentinel is a valid input.



Testing for validity can take two forms:

- Testing if a valid input type has been entered, for instance, if an integer is expected, it is important to know that one has been entered.
- Once the first test has been satisfied, and where a value within a specified range is expected, then the value of the input should be tested.

The user will usually enter a valid input in the first instance, but if they do not, in the loop an error message is output and a subsequent input is gathered. This looping can continue indefinitely until the user enters a valid value.

After each input (within the loop and after the loop) the input stream is cleared. If the user has entered additional, unwanted data, either accidentally or maliciously, then it will be removed before the next input is sought.

Here is an example function that gathers a valid integer in a specified range.

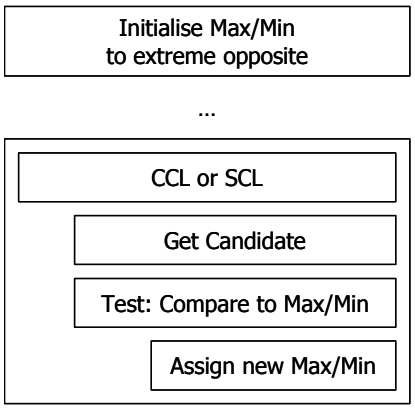
```
int getValidIntegerInRange(int lowestAllowed, int highestAllowed) {
      int userInput = 0;      // Input from user
      int inputsGathered = 0; // Number of inputs from scanf()

      // Prompt for initial input
      printf(
             "Please enter an integer between %i and %i: ",
             lowestAllowed, highestAllowed
      );
      inputsGathered = scanf("%i", &userInput);

      // Test for valid input
      while(
             inputsGathered !=1 ||
             userInput < lowestAllowed ||
             userInput > highestAllowed
      ) {

             // Clear standard input
             scanf("%*[^\n]");
             scanf("%*c");

             // Error message prompt
             printf(
                    "Invalid input. "
                    "Please enter an integer between %i and %i: ",
                    lowestAllowed, highestAllowed
             );
             inputsGathered = scanf("%i", &userInput);
      }

      return userInput;
}
```

Note that where inputs are gathered from the user, the return value from **`scanf()`** is also captured. The function **`scanf()`** will attempt to input values according to the format string, storing the values at the addresses provided. The return value of **`scanf()`** is not an input value, but the number of values that have been successfully input and stored. Using this we can determine if an appropriate value has been entered by the user. See the description of **`scanf()`** in Appendix 1 for more detail.

Plan 12. Min/Max Plans

This plan requires an understanding of looping constructs and the ***if*** *statement.*

To find the minimum or maximum from a number of user inputs, it is not necessary to keep all candidates, just the current min/max at any stage.



This process starts by selecting an initial value for the min/max variable. If searching for a maximum, initialise to the minimum possible value. If searching for the minimum, initialise to the maximum possible value. In that way the first value encountered will become the new min/max. Alternately the first value encountered (if it can be guaranteed there will be a single value) can be used as the initial value for the min/max.

As each candidate is presented within a loop (a counter controlled loop or sentinel controlled loop) it needs to be compared with the current-max/min. If searching for a maximum and the candidate is greater than the current maximum, then the candidate will be assigned as the new current-maximum.

The following example inputs 5 numbers between 0 and the largest integer value allowed. Inputs are gathered from a user using **`getValidIntegerInRange()`** as shown in Plan 11 above. The **`maxNumber`** variable is used to store the current maximum and it is initialised to 0 which is the smallest input allowed.

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

const int NUMBERS_TO_READ = 5;

int getValidIntegerInRange(int lowestAllowed, int highestAllowed);

int main() {
      int i;              // Iterative counter
      int input;          // Validated Input from user
      int maxNumber = 0; // Current maximum initialised to
                          //   minimum possible value

      // Get inputs from user
      for(i = 0; i < NUMBERS_TO_READ; i++) {
            input = getValidIntegerInRange(0,INT_MAX);

            // Compare with current max and assign if greater
            if(input>maxNumber) {
                  maxNumber = input;
            }
      }

      // Output the max
      printf("The maximum was: %i\n", maxNumber);
}

int getValidIntegerInRange(int lowestAllowed, int highestAllowed) {
      ...
```

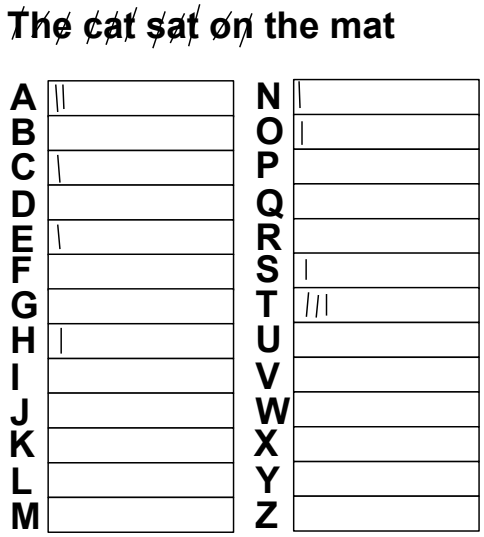
Note that each input is compared with the current maximum. Where a candidate is found to be greater than the current maximum it replaces the current maximum and is used for future comparisons.

Plan 13. Tallying Plan

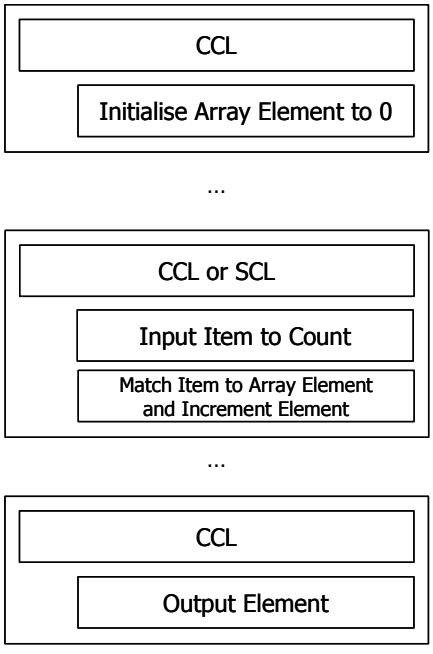
This plan requires an understanding of arrays and looping constructs.

As well as being able to store individual values in an array we can also use arrays to represent counts of occurrences of a set of values.

For instance if I asked you to count each letter in the sentence, "The cat sat on the mat", you could set up a sheet and tally each letter in the sentence. We start off with a blank sheet where the tally each letter is empty (zero). We process each letter in turn, crossing it off in the sentence as it is processed. When we encounter a letter, we place a tally mark in the box on our sheet that relates to that letter. We can continue this until all the letters are processed, at which stage the number of tally marks next to each letter is the number of occurrences of that letter.



We can apply a similar strategy in code using an array.



We will create an array with enough elements to represent the set of values we are counting. If we are counting the letters of the alphabet we need an array with 26 elements. Before we start counting we must first initialise the array to be sure the count of all values is zero.

We can then process the values, matching them to the relevant element of our array and 'adding another tally mark' (incrementing the count) for that value.

When we have processed all items of interest the values in the array will be the counts of the items encountered. If we wish we can output the counts of the letters encountered.

The following code is an example of such a strategy.

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

const int SENTINEL = 9999;

int main() {
      int letters[26];  // Array for tallying letters encountered
      int i;            // Iterative counter
      char inputLetter; // Letter from user

      // Initialise all array elements to 0
      for(i=0; i<26; i++) {
            letters[i] = 0;
      }

      // Process the user input until end of line
      printf("Please input a sentence...\n");
      scanf("%c", &inputLetter);
      while(inputLetter != '\n') {
            if(isalpha(inputLetter)) {
                  letters[tolower(inputLetter)-'a']++;
            }
            scanf("%c", &inputLetter);
      }

      // Output occurrences of letters which have occured once or more
      for(i=0; i<26; i++) {
            if(letters[i] > 0) {
                  printf("%c: %i\n", 'a'+i, letters[i]);
            }
      }
}
```

Notice first that the array is initialised, the values are counted and then the counts are output. See the language reference for descriptions of **`isalpha()`** and **`tolower()`**.

The array used is an array of integers, which is appropriate as we are storing counts of letters and not the letters themselves. The array elements are referenced by index and the indices are integers, so this means we have to translate each character into a number to find the array element that relates to that letter. We can associate each alphabetic letter with a number in order starting from 'a' being 0, 'b' being 1 and so on. To achieve this we can convert each letter to lower case and deduct the value of 'a' as follows.

```
'a' – 'a' → 0
'b' – 'a' → 1
'c' – 'a' → 2
      ...
'z' – 'a' → 25
```

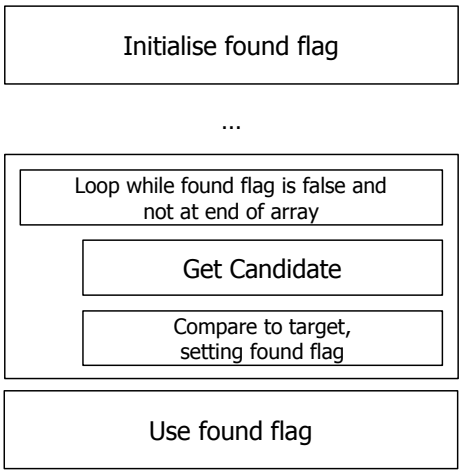
Once we have a letter's position in the alphabet we can use this as the index into the array to access the array element that relates to that letter of the alphabet. When we are counting a particular letter, we will translate it into a number, find the array element and increment its value. This is achieved in the statement from the above example shown below.

```
letters[tolower(inputLetter)-'a']++;
```

Plan 14. Search Algorithm

This plan requires an understanding of looping constructs and arrays.

This plan and the next are approaching the scale of a full algorithm and could exist independently as useful functions.



The key to efficient searching is to search only the parts of the search space (say the elements of an array) necessary to discover the value sought. Of course, if the location of the target value is unknown then the amount of searching required cannot be predicted, but, if we are seeking the presence of a target value we should be able to stop searching after we discover the value. In the case that the target value is not present, searching will continue until the end of the search space is reached.

One way to achieve this is through a combination of a sentinel controlled loop that searches for the target value as a sentinel and a counter controlled loop that stops when the end of the search space is reached. We can use a Boolean flag to control the test for the target value and the value of this flag after the search will tell us if the target value is present. Here is an example function that searches an array for a target value.

```
bool search(int targetValue, int array[], int arrayLength) {
      bool found = false; // Boolean search flag
      int i = 0;          // Iterative counter

      // Search until found or end of array
      while(!found && i<arrayLength) {

            // Match array element to target value
            found = array[i]==targetValue;
            i++;
      }
      return found;
}
```

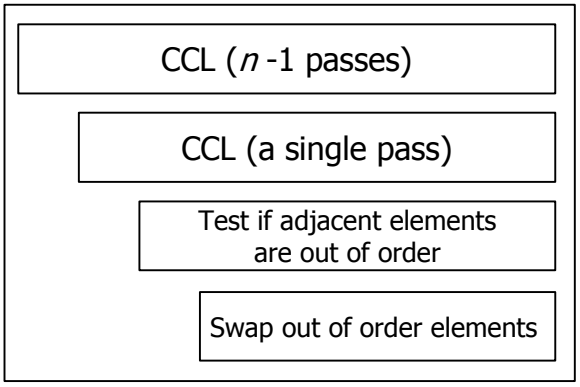
Of course, this approach will only work if we are seeking the presence of a target value. If we wish to count the occurrences of a value we will need to search the entire search space, so no saving can be made.

```
int countValues(int targetValue, int array[], int arrayLength) {

      int i;       // Iterative counter
      int count=0; // Times targetValue has been encountered

      // Search entire array for occurrences of target value
      for(i = 0; i < arrayLength; i++) {
            if( array[i] == targetValue ) {
                  count++;
            }
      }

      // Return the count of occurrences
      return count;
}
```

Plan 15. Bubble Sort Algorithm

This plan requires an understanding of looping constructs and arrays.

There are a many different algorithms which can be used to put elements in order. The Bubble Sort is presented here as it is easy to comprehend and use.



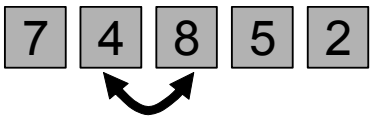
This algorithm works by looping through the array comparing each element with the following one, and swapping the values where necessary. Each pass through the array brings it closer to being sorted. The looping and swapping process must occur as many times needed to ensure the array is completely sorted. If we loop through the array *n-1* times (where *n* is the length of the array), it is guaranteed to be sorted.

The process can be summarized as follows.

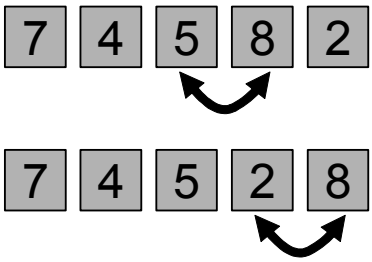
- Start at beginning of the array
- Compare first and second elements
- If out of order swap
- Compare the second and third elements
- If out of order swap
- Continue comparing adjacent pairs in the array, from beginning to end; this constitutes a single pass.
- Perform *n-1* passes to completely sort the array.

Consider the following array.

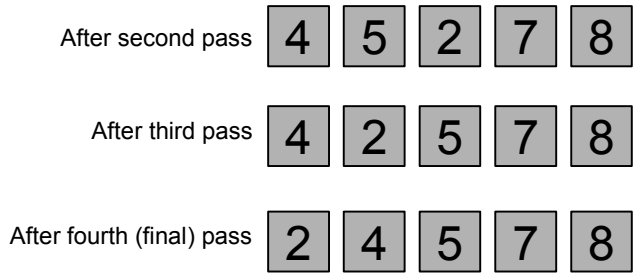
Starting at the beginning we compare the first two values. They are in order so we do not swap them. The second and third values are out of order and must be swapped. The outcome is shown below.



We continue comparing and swapping adjacent values if needed until we get to the end of the array.



The state of the array after one pass is shown above. We will complete four passes through the array. The state of the array after each pass is shown below.



The following program will perform a *bubble sort* on an array of integers to put them in ascending order.

```
#include <stdio.h>

const int MAX_LENGTH = 5;

int main() {
      int array[MAX_LENGTH] = {9,8,2,5,4}; // Unsorted array
      int i, j;                            // Loop iterators
      int temp;                            // For swapping

      // Pass through the array MAX_LENGTH-1 times
      for( i = 0; i < MAX_LENGTH-1; i++){

            // For each pair of consecutive numbers
            for( j = 0; j < MAX_LENGTH-1; j++) {

                  // Test if the pair is out of order
                  if ( array[j] > array[j+1] ) {

                        // Swap using triangular swap
                        temp = array[j];
                        array[j] = array[j+1];
                        array[j+1] = temp;
                  }
            }
      }

      // Output the array after sorting
      for(i = 0; i < MAX_LENGTH; i++){
            printf("%i ",array[i]);
      }
      printf("\n");
}
```

Notice the above code contains two for loops, one inside the other. The outer loop ensures that *n-1* passes are performed. Each iteration of the outer loop, the inner nested loop compared each adjacent value in the array and swaps it if necessary.

Bubble sort is not the most efficient sorting algorithm. For large and unordered data faster sorting algorithms are available. The efficiency of the Bubble Sort algorithm can be improved by applying the following two modifications.

- Reduce the number of comparisons by one for each pass. After the first pass the greatest value will be pushed to the rightmost element. After two passes, the final two elements will contain the two greatest values in sorted order and so on. To achieve this, the value of **`i`** can be deducted from the upper limit of the inner loop.

  `j < MAX_LENGTH-1-i;`
- For an array that contains values that are nearly already sorted, it is possible to reach a sorted state before *n-1* passes have been made. The array can be determined to be in a sorted state when a complete pass has been performed in which no swaps are made. A Boolean flag **`swapsMade`** can be used which is set to **`false`** at the beginning of each pass. If it is still false at the end of the pass, no swaps have been made and the array is in sorted order. This flag can be incorporated into the test of the outer loop.

Plan 16. Command Line Arguments Plan

This plan requires an understanding of command line arguments and the ***`if`*** *statement.*

If information provided to a program from the command line is crucial to the

successful running of the program, then the number of arguments needs to be checked at the beginning of program execution.

```
#include <stdio.h>

int main(int argc, char *argv[]) {

      // Check for the correct number of arguments
      if ( argc < 2 ) {
            printf("USAGE: %s secondArgument\n", argv[0]);
            exit(1);
      }

      // Rest of program
      ...
}
```

The arguments to the **`main()`** are **`argc`** (the number of command line arguments) and **`argv`** (an array of strings, each containing and argument). The code above shows a test for the minimum number of command line arguments needed. In this case the program expects two arguments and any extras will be ignored. If the user runs the program and does not supply a second argument, then an error message is output and the program exits. Note that the name of the executable file will be stored in **`argv[0]`** and this is used in the error message; the name of the executable could change, but the error message will always be correct.

Once the number of command line arguments has been checked, the validity of the values supplied may then also need to be checked.

Plan 17. File Use Plan

This plan requires an understanding of files and the if statement.

When using input files, where data sourced from those files is critical to the running of a program, the following 5 Step Plan should be taken. This plan takes checks that the file is available for use. It closes the stream when it is no longer needed; this is important to avoid data loss.

1 Create a stream (FILE) pointer

```
FILE *inputStream;
```

2 Open a file and attach the stream

```
inputStream = fopen("myfile.txt","r");
```

3 Test the stream, this testing the file opening

```
if (inputStream == NULL) {
      printf("Error opening file");
      exit(1);
}
```

4 Use the stream for input or output (this will of course vary according to the needs of the input stream)

5 Close the stream

```
fclose(inputStream);
```

Plan 18. Recursion Plans (single- and multi-branching)

This plan requires an understanding of the ***if*** *statement and calling functions.*

A recursive function is one which calls itself, either directly or indirectly. Recursive functions are very simple, but can achieve quite complex solutions by solving a problem a small part at a time. Recursion is a way of achieving repetition in a program.

Recursive functions have two parts: a stopping case and a recursive case. An **if** statement is used to determine which case should be used as shown in the skeleton below.

```
int exampleRecursiveFunction( ...ARGUMENTS... ) {

      // Stopping case
      if( TEST TO SEE IF RECURSION SHOULD STOP ) {
            ...;
      }

      // Recursive case
      else {
            ...
            exampleRecursiveFunction( ... );
            ...
      }
}
```

The recursive case contains a recursive function call. Each time the recursive function is called, the arguments passed should be slightly different to those used to call the current function. In that way progress is made towards the end of recursion.

The stopping case is reached when some end has been achieved. It contains no further recursive function calls.

The following function is a recursive function that counts down from any positive number to zero.

```
void countDown(unsigned int number) {

      // Stopping case
      if(number == 0) {
            printf("0\n");
      }

      // Recursive case
      else {
            printf("%i\n", number);
            countDown(number - 1);
      }
}
```

The stopping case for this function occurs when the value of **number** is zero. If we called this function once and passed it the value zero, it would use the stopping case immediately and end. If a greater number is passed the recursive case will be used and the recursive function call within that passes a number one less each time. In this way the stopping case will eventually be reached.

We could start the recursive process, starting at the number 3, by calling the **`countDown()`** function from the **`main()`** and passing the value 3.

```
int main() {

      // Start the count down at 3
      countDown(3);
}
```

The output of this program would be as follows.

```
% a.out
3
2
1
0
%
```

Below is an example of another recursive function that can be used to calculate factorials. The factorial of an integer is the integer multiplied by all the positive integers less than it to one. We denote the factorial of a number using an exclamation (!) like as follows.

$$5! = 5 \times 4 \times 3 \times 2 \times 1$$

The factorial for 4! can be expressed as follows.

$$4! = 4 \times 3 \times 2 \times 1$$

If we wanted to, we could now express 5! as follows.

$$5! = 5 \times 4!$$

You can see the recursive nature of this equation already. We can make this a general equation as follows. This is our recursive case.

$$n! = n \times (n-1)!$$

We also need to express a stopping case for this, which is when n is 1.

$$1! = 1$$

This is a mathematical definition of a recursive process. If we were to run it through for say 4! it would look as follows.

$$4! = 4 \times 3!$$

$$3! = 3 \times 2!$$

$$2! = 2 \times 1!$$

We know that 1! is equal to one. We can now start working our way back up.

$$2! = 2 \times 1! \rightarrow 2 \times 1 \rightarrow 2$$

$$3! = 3 \times 2! \rightarrow 3 \times 2 \rightarrow 6$$

$$4! = 4 \times 3! \rightarrow 4 \times 6 \rightarrow 24$$

So 4! is 24. We can write a function that calculates factorials using the process we have described as follows.

```
int factorial(unsigned int number) {

	// Stopping case
	if (number <= 1) {
		return 1;
	}

	// Recursive case
	else {
		return number * factorial(number – 1);
	}
}
```

You will notice that with this function, as well as actions being achieved on the way to the stopping case, calculations are happening through the return values after the stopping case has been reached and while working back to the original function call. In order to complete the expression in the recursive case…

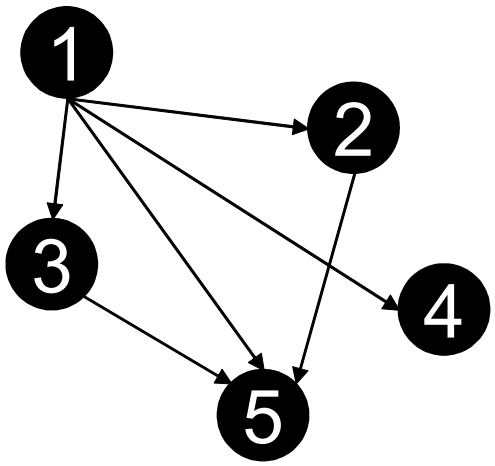
```
		return number * factorial(number – 1);
```

…the factorial function needs to be called. We must wait for this function to end and return a result before we can complete the expression.

This function is an example of single branching recursion. The recursive case contains only a single function call, so the recursive process will continue until a single stopping case is reached, after which the calls will roll back to the original function call.

A multi-branching recursive function contains more than one recursive function call in the recursive case. This is useful for problems where from a particular point there may be several following points that need to be probed and from each of those points further points need to be probed and so on. There may be multiple stopping points that can be reached in such cases also. Consider for example, a directed graph. A directed graph is described by its points and the vertices between points that run in one direction only. The vertices are like one-way streets that join one place to another.

The picture below describes a directed graph. The starting point is 1 and the ending point is 5. We can represent this information textually as shown with each vertex having a starting and ending point and a series of directed vertices that make up the graph.




```
1
5
1 2
1 3
2 5
3 5
1 4
1 5
```

Our task is to find how many paths lead from the starting point to the ending point assuming that there are no cycles in the graph. We can represent a graph as follows.

```
struct directedGraph {                 // Describes a directed graph
      vertex vertices[MAX_VERTICES]; // The vertices that make up the graph
      int numVertices;                 // The number of vertices
      int startPoint;                  // The starting point
      int endPoint;                    // The end/target point
};
```

We can then create a recursive function that, when started at the start point, will discover how many paths lead to the end point.

```
int countPaths(directedGraph graph, int currentPoint) {
      int countPathsFromHere=0; // Paths in the graph starting here

      // Stopping case
      if(currentPoint == graph.endPoint) {

            // A complete path has been found
            return 1;
      }
      else {

            // Probe all paths that start here
            for(int i=0; i<graph.numVertices; i++) {
                  if(graph.vertices[i].from == currentPoint) {
                        countPathsFromHere += countPaths(
                              graph,
                              graph.vertices[i].to
                        );
                  }
            }

            // Return the number of completed paths staring here
            return countPathsFromHere;
      }
}
```

Assuming we have read in a graph into a structure variable called **`graph`** we could start this recursive process as follows, printing out the number of paths returned.

```
printf("%i\n", countPaths(graph,graph.startPoint));
```

Recursion is a less efficient way of achieving repetition than when using loops. However when a problem is being solved that is recursive by nature, writing recursive solutions can be far simpler than writing an iterative solution for the same functionality. Where the depth of recursion is on too deep, recursive solutions can be quite acceptable.

Strategies Index