Thinking in C[*]

Darin Brezeale

March 25, 2010

NOTE: This is definitely a work in progress.

1 Introduction

One of the most difficult parts of learning to program is knowing how to deconstruct a problem in order to represent it in a programming language. In this guide I will attempt to describe my thinking process when writing programs. One book that is very useful in this regard after you already have some programming experience is [Ben00].

When reading this document, please keep in mind the following:

- Programming is one of those activities described as being an art. Like most such activities, this means that how to do it can't be described exactly beforehand. If given a specific task, I might be able to tell you exactly how I would do it, but I can't give you a specific answer that will work for all situations.
- Some of what I will say below is written in a cookbook fashion, but as stated in the previous point, these may be generalizations that won't be true in all situations.
- For any given task, there can be many "correct" ways to write a program.

Here are some general thoughts on how to write your code to make your life easier:

- Have the computer do the work. Example: You need to find the divisors of 100. Instead of writing the code to check if 2 is a divisor, then the code to check if 3 is a divisor, and so forth up to 100, have the program generate the possible divisors and check each.
- Think about how to write your code in a general way so that it is not tied to a specific situation. Example: If you are finding the divisors of 100, instead of writing your code so that it would only work for 100, write it so that you can provide a number n and it will find the divisors of n. This allows the same code to work for any positive integer, not just 100.
- Large programs are collections of small tasks. Think about how to divide your code into these tasks. This not only makes the process of writing the code easier to understand, but it also helps you identify code that should be in a function.

[*]I was tempted to call this *I Can't C*, which is a play on the title of the book *Let Us C* by Yashavant Kanetkar.

2 Asking Questions

We often need to do a particular task if some condition is true. For example, if a person's age is greater than 80, the program will print "You are old." To ask questions and respond when the answer is yes, we use conditionals. There are three types of conditionals in C: `if` statements, `switch` statements, and the tertiary operator. I'll only discuss the first two. The `if` statement is the most common of the three and is used in situations in which we wish to do a specific thing if some condition is true. The example above would be implemented in C as

```
if(age > 80)
    printf("You are old.");
```

In this example, the statement "You are old." is printed only when the age is greater than 80. This may be all we are concerned with, but what if we want to do something else when the age is less than or equal to 80? In that case, we can add an `else` statement:

```
if(age > 80)
    printf("You are old.");
else
    printf("You are not old yet.");
```

Something to remember is that `else` statements are optional; we don't have to have them every time we have an `if` statement. However, every `else` statement must be preceded by a corresponding `if` statement.

While there are many situations in which we only need to ask a single question, there are times when we need to ask multiple questions and respond when they are all true. Example: We wish to print "You are old." when a person's age is in the range $80 \leq \text{age} < 100$. While you might think this is a single question, in reality it is two: "Is 80 less than or equal to the age?" and "Is the age less than 100?" We can see this if we consider an age of 120; the answer to the first question is yes, but the answer to the second is no.

We have a couple of options in this case. One is to use a set of nested `if` statements:

```
if(80 <= age)
{
    if(age < 100)
        printf("You are old.");
}
```

We only reach the second `if` statement if the first `if` statement is true. Another way to accomplish this task is to use the logical `and` operator, &&:

```
if(80 <= age && age < 100)
        printf("You are old.");
```

Notice that I have compared the variable `age` to each value individually. A common mistake is to try to write such questions as you would in math, that is,

```
if(80 <= age < 100)
        printf("You are old.");
```

The problem with this is that it always evaluates as true (in this example). Why? Because relational operators are evaluated from left to right, which means that 80 is compared to the age first and then the answer to that question (either 0 or 1) is compared to 100, which will always be true.

Sometimes we may have a sequence of tasks to complete and, depending on some integer value, jump to a specific point in that sequence. For these situations we may wish to use a `switch` statement. Example: We wish to print a countdown as words starting with a value in the range $1, \ldots, 4$.

```
switch(countdown)
{
    case 4:  printf("four\n");
    case 3:  printf("three\n");
    case 2:  printf("two\n");
    case 1:  printf("one\n");
}
```

If `countdown` has a value of 3, then this would produce

```
three
two
one
```

We could also do this using a set of `if/else` statements:

```
if( countdown == 4 )
    printf("four\n");
if( countdown >= 3 )
    printf("three\n");
if( countdown >= 2 )
    printf("two\n");
if( countdown >= 1 )
    printf("one\n");
```

Whether to use a set of `if/else` statements or a `switch` statement is largely a matter of preference, but there are situations in which the `switch` version is cleaner.

3 Repeating Tasks

When we wish to repeat a task, we use one or more loops. Here are some questions with answers to help you decide if and how to use a loop.

1. Are you repeating some task, at least in the general sense?

   (a) No. You don't need a loop.

   (b) Yes. Then keep reading this section.

2. If yes, does the task differ from iteration to iteration?

   (a) No. Example: You need to print the text "This is text." four times, once per line.

   ```
   int i;

   for(i = 0; i < 4; i++)
       printf("This is text.\n");
   ```

   Notice that since the task is identical for each iteration of the loop, we do not use the loop variable *i*. Therefore, all we are concerned with is making the loop run four times, so the particular starting value of the loop does not matter. For example, I could have done this:

   ```
   int i;

   for(i = -1234; i > -1238; i--)
       printf("This is text.\n");
   ```

   You probably would not want to do this since someone would look at your code and wonder if there is some significance to starting at -1234, but these starting and stopping values for *i* do cause the loop to run for four iterations.

   (b) Yes. Then will you use the loop variable to modify the task during each iteration?

      i. No. Example: If you start counting up from 12 and down from 20, at what point do the counts reach each other?

      ```
      int up = 12, down = 20;

      while(1)  /*  create what could be an infinite loop  */
      {
          if( up >= down)  /* stop when they meet or cross */
              break;

          up++;
          down--;
      }

      printf("up is %d, down is %d\n", up, down);
      ```

ii. Yes. Can the loop variable be used directly to modify the task during each iteration of the loop?

   A. Yes. Then the choice of starting and stopping values for the loop variable must have specific values. Example: Print the integers from 5 to 10, each on a separate line.

      ```
      int i;

      for(i = 5; i <= 10; i++)
          printf("%d\n", i);
      ```

      Here the integers $5, 6, \ldots, 10$ are needed, so the loop is used to produce them.

   B. No. Maybe the loop variable can be transformed by a mathematical function to produce the desired value. Example: Print the first ten elements of the sequence

      $$f(n) = \begin{cases} 3n+1 & \text{if } n \text{ is odd} \\ 4n-3 & \text{if } n \text{ is even} \end{cases}$$

      which begins $4, 5, 10, 13, 16, 21, 22, 29, 28, 37, \ldots$

      ```
      int i, fx;

      for(i = 1; i <= 10; i++)
      {
          if(i%2 == 1)
              fx = 3*i + 1;
          else
              fx = 4*i - 3;

          printf("%d, ", fx);
      }
      ```

Figure 1 shows this series of questions.

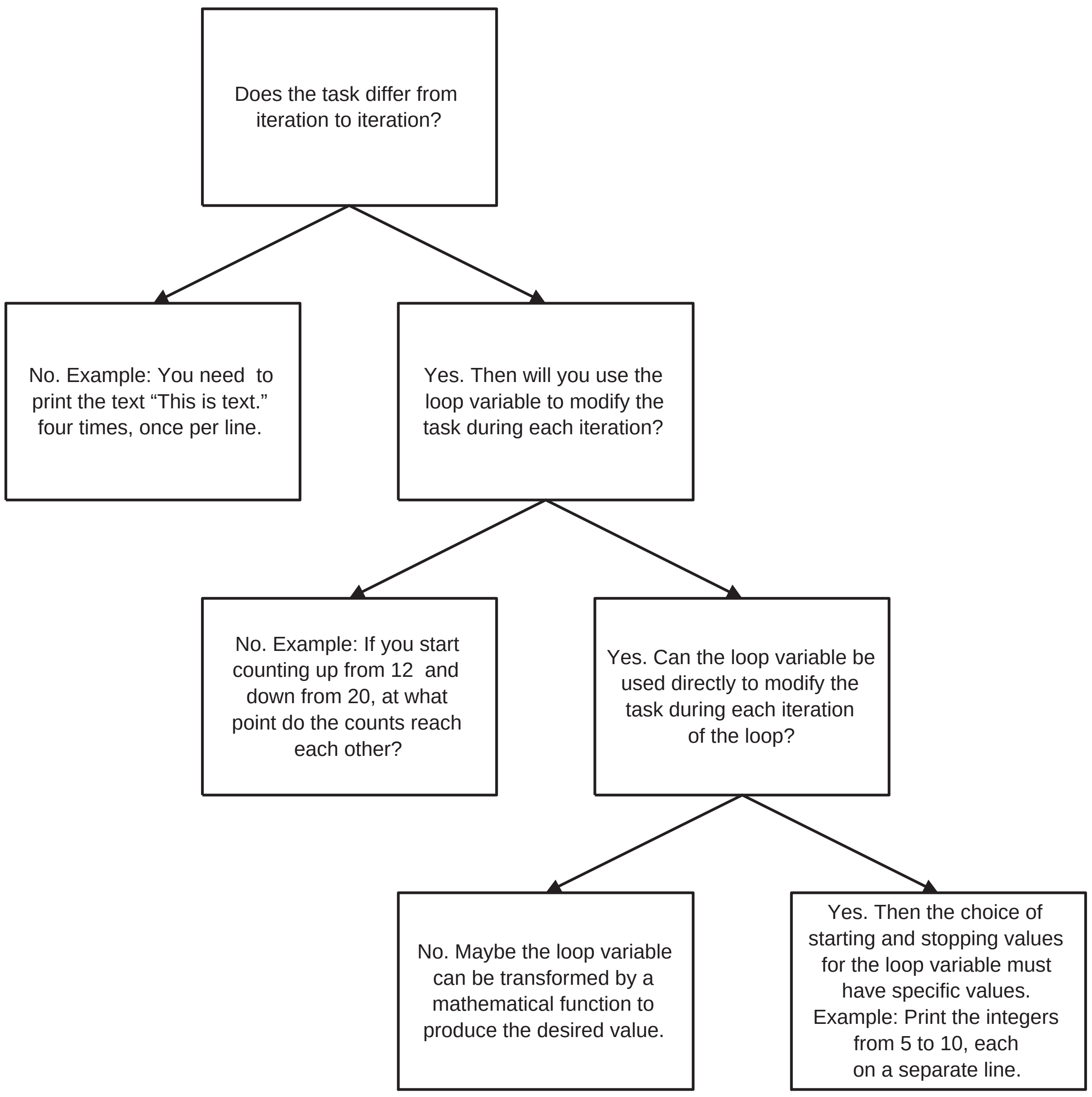
Does the task differ from iteration to iteration?
No. Example: You need to print the text “This is text.” four times, once per line.
Yes. Then will you use the loop variable to modify the task during each iteration?
No. Example: If you start counting up from 12 and down from 20, at what point do the counts reach each other?
Yes. Can the loop variable be used directly to modify the task during each iteration of the loop?
No. Maybe the loop variable can be transformed by a mathematical function to produce the desired value.
Yes. Then the choice of starting and stopping values for the loop variable must have specific values. Example: Print the integers from 5 to 10, each on a separate line.

Figure 1: Flow chart to determine how to set up loop.

Another consideration is whether the number of iterations can change.

1. No. Then it is OK to hard-code the values. That is, use explicit starting and stopping values for the loops. Our previous examples do this.

2. Yes. Use variables for the initialization and/or test of the loop variable. Example: For each of the integers i in the range of 1 to 5, print $1 \ldots i$, each on a separate line.

   What's the thinking process here? Is there a task to repeat? Writing the desired results makes the tasks more clear:

   ```
   1
   1 2
   1 2 3
   ```

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

We have two tasks here. One is to generate five rows of numbers. The other is to produce the integers 1 to n for a particular row. If we rewrite the problem in pseudocode, it may help us understand the relationships between the two tasks:

```
for each integer i in the range 1 to 5
    print 1 to i
```

The task

```
    print 1 to i
```

can itself be represented as

```
for each integer k in the range 1 to i
    print k
```

Together, these become

```
for each integer i in the range 1 to 5
    for each integer k in the range 1 to i
        print k
```

We can see that the inner loop, which produces k, is dependent on the outer loop, which produces i. Therefore, we need two loops: one to produce i and a second that for each iteration of the i loop produces each of the integers $1 \ldots k$. The k loop should be inside the i loop.

```
int i, k;

for(i = 1; i <= 5; i++)
{
    for(k = 1; k <= i; k++)
        printf("%d ", k);

    printf("\n");  /* go to next line after printing 1..k */
}
```

How do we represent a task to make the use of the loops easier to identify? Example: Print the integers from 5 to 10, each on a separate line.

1. Here is a non-loop version, with values hard-coded.

```
printf("5\n");
printf("6\n");
printf("7\n");
printf("8\n");
printf("9\n");
printf("10\n");
```

2. Each line of the previous code is specific to the number that is to be printed, but we can see that the numbers are in order from 5 to 10. Generating numbers in order is easy for a loop, but we need to modify the tasks first. By creating a variable i and reassigning it to each of the integers in the range 5 to 10, we can rewrite the `printf()` statements to print i.

   ```
   int i;

   i = 5;
   printf("%d\n", i);

   i = 6;
   printf("%d\n", i);

   i = 7;
   printf("%d\n", i);

   i = 8;
   printf("%d\n", i);

   i = 9;
   printf("%d\n", i);

   i = 10;
   printf("%d\n", i);
   ```

3. Now the task is to simply print i. By using a loop to repeatedly generate new values of i, we can rewrite out program as:

   ```
   int i;

   for(i = 5; i <= 10 ; i++)
       printf("%d\n", i );
   ```

If you are unsure of what values a loop, or set of nested loops, produces, print them out:

```
int i, k;

for(i = 1; i <= 3; i++)
{
```

```
        for(k = 1; k <= 4; k++)
            printf("i=%d, k=%d\n", i, k);
    }
```

produces

```
    i=1, k=1
    i=1, k=2
    i=1, k=3
    i=1, k=4
    i=2, k=1
    i=2, k=2
    i=2, k=3
    i=2, k=4
    i=3, k=1
    i=3, k=2
    i=3, k=3
    i=3, k=4
```

We can see from this that for each iteration of the i loop, the k loop runs from 1 to 4.

4 Code Re-use

At times we find that we need to do essentially the same thing at multiple locations in a program.

Example: We want the square root of several unrelated numbers and are writing the code to calculate the square root. First, we need to know how to calculate the square root. One approach is described in the following pseudocode [RN95, pg 34]:

```
function $SQRT(x)$
$z \leftarrow 1.0$          /* initial guess */
repeat until $|z^2 - x| < 10^{-15}$
    $z \leftarrow z - (z^2 - x)/(2x)$
end
return $z$
```

Making use of this, our program is:

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

int main(void)
{
    double x, z;

    x = 100.0;
    z = 1.0;

    while( fabs(z*z - x) > 0.000000000001 )
    {
        z = z - (z*z - x)/(2*z);
    }

    printf("The square root of %f is %f.\n", x, z);


    x = 32.456;
    z = 1;

    while( fabs(z*z - x) > 0.000000000001 )
    {
        z = z - (z*z - x)/(2*z);
    }

    printf("The square root of %f is %f.\n", x, z);


    x = 11000;
    z = 1;

    while( fabs(z*z - x) > 0.000000000001 )
    {
        z = z - (z*z - x)/(2*z);
    }

    printf("The square root of %f is %f.\n", x, z);
}
```

We can see that we are using exactly the same code in each location that we wish to calculate the square root. It would be much easier if we segregated this code into a function:

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

double squareroot(double num);

int main(void)
{
    double x, ans;

    x = 100;
    ans = squareroot(x);
    printf("The square root of %f is %f.\n", x, ans);

    x = 32.456;
    ans = squareroot(x);
    printf("The square root of %f is %f.\n", x, ans);

    x = 11000;
    ans = squareroot(x);
    printf("The square root of %f is %f.\n", x, ans);
}

double squareroot(double x)
{
    double z = 1;

    while( fabs(z*z - x) > 0.000000000001 )
    {
        z = z - (z*z - x)/(2*z);
    }

    return z;
}
```

For this example, the sizes of the two versions of the program are not that different, but had the amount of code being copied each time been much larger the function version would have been much smaller. Creating a function for commonly used code has several other benefits:

- If the code needs to be modified, then we can simply change the function and recompile. Without the function, we would need to find and modify each set of square root code.
- We might want to use this code in another program and having it "packaged" into a function makes this much easier.
- Using the function makes the code where the function is called easier to read and understand.

Let's do a more complicated example. Let's say we wish to convert a sequence of integers from base-10 to base-2. There are several ways to do this, but the approach I am going to take shows how to use loops in a very methodical way. First, though, let's review first grade math. When we have a base-10 number such as 456, this is

$$\begin{array}{rr} & 4 \times 10^2 \\ + & 5 \times 10^1 \\ + & 6 \times 10^0 \end{array} = \begin{array}{rr} & 400 \\ + & 50 \\ + & \underline{\;\;6} \\ & 456 \end{array}$$

Ten is our base and therefore our number is composed of a sum of multiples of powers of 10. Other bases work the same way. In the case of base-2, the number is composed of powers of 2, for example the quantity 13 (in base-10) is written in base-2 as

$$\begin{array}{rr} & 1 \times 2^3 \\ + & 1 \times 2^2 \\ + & 0 \times 2^1 \\ + & 1 \times 2^0 \end{array} = \begin{array}{rr} & 1000 \\ + & 100 \\ + & 0 \\ + & \underline{\;\;\;1} \\ & 1101 \end{array}$$

Therefore, the base-10 number 13 is written in base-2 as 1011.

The approach I am going to use to convert from base-10 to base-2 is to generate powers of 2 until their sum represents the same quantity as is represented by the base-10 number. In order to make this easier, my program will only be able to handle base-10 numbers in the range of 0 to 255; 255 in base-2 is 11111111. My program will prompt the user for a range of integers to convert, so the basic outline of my program is:

1. prompt the user for the starting and stopping values
2. perform error-checking to ensure that the starting and stopping values are in the valid range
3. for each integer in the range, convert it from base-10 to base-2

An application like this is a good place to use functions since I have several parts that do very specific things and looking at the function call names makes it much more clear what is happening. In `main()` I will prompt the user for the starting and stopping values, but the next two steps will each be in its own function. In addition, I will avoid using the math library, so in order to calculate the powers of two, I will write a function that can raise a positive integer to a nonnegative integer power.

`main()` and the function declarations is

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

void errorcheck(int, int);
void ten2binary(int);
int power(int base, int exp);

int main( void )
{
    int i, start, stop;

    printf("Please provide starting and stopping integers, each\n");
    printf("in the range 0 <= number < 256.\n\n");
    printf("Starting number: ");
    scanf("%d", &start);
    printf("Stopping number: ");
    scanf("%d", &stop);
    printf("\n");

    /* some simple error checking of the input values */
    errorcheck(start, stop);

    for(i = start; i <= stop; i++)
        ten2binary(i);
}
```

Once we have received starting and stopping values from the user, we need to determine if they are in the valid range (we assume they are integers). If so, we do nothing. If not, we tell the user and exit. Therefore, our function needs to receive two variables of type `int` and has a return type of `void`. Looking at the function call in `main()`,

```
    errorcheck(start, stop);
```

we can see that we are providing the two values to our function and not receiving anything, so our function declaration is correct.

```
void errorcheck(int start, int stop)
{
    if(start < 0 || start >= 256)
    {
        printf("Sorry, but that starting value is invalid.\n");
        exit(1);
    }
    else if(stop < 0 || stop >= 256)
    {
        printf("Sorry, but that stopping value is invalid.\n");
        exit(1);
    }
    else if(stop < start)
    {
        printf("Sorry, but the stopping value cannot be larger ");
        printf("than the starting value.\n");
        exit(1);
    }
}
```

If the values that the user provided are in the valid range, then we need to convert each of them from base-10 to base-2. To do this, we put our function call in a loop:

```
    for(i = start; i <= stop; i++)
        ten2binary(i);
```

So how does this function work? At a high level, what we want to do is to find the largest power of 2 that is less than or equal to our number. Print a 1 for this value and subtract this power of 2 from our number. Repeat this process for the new number.

We are only dealing with integers in the range of 0–255, so the biggest power of two that we will consider is 2^7. To make the process easier, we also write our base-2 number using eight digits, even if the beginning digits are zeros. To understand the process, let's work an example by hand using the base-10 number 45.

We begin by checking if the our base-10 number is greater than or equal to $2^7 = 128$; it isn't so we write a 0 as the leftmost digit. Now we check if our base-10 number is greater than or equal to $2^6 = 64$; it isn't so we write a 0. Now we check if our base-10 number is greater than or equal to $2^5 = 32$; it is, so we write a 1 and subtract 32 from the quantity to be accounted for. We repeat this process until the entire quantity has been represented by the sum of powers of 2. We can see the entire process in Table 1.

quantity to account for	base-2 number to check	digit written
45	$2^7 = 128$	0
45	$2^6 = 64$	0
45	$2^5 = 32$	1
45 - 32 = 13	$2^4 = 16$	0
13	$2^3 = 8$	1
13 - 8 = 5	$2^2 = 4$	1
5 - 4 = 1	$2^1 = 2$	0
1	$2^0 = 1$	1
1 - 1 = 0	stop	

Table 1: Process of converting 45_{10} to 00101101_2

So the base-10 number 45 is written in base-2 as 00101101. Now that we have seen the process in action, let's produce the algorithm, which should be make it easier to determine how to write our function.

1. Start with 2^7 and continue until 2^0
2. Check if our base-10 number is greater than or equal to 2^n
3. if so
   (a) write ‘1’
   (b) subtract 2^n from the base-10 number; this will be the new number to check at the next stage
   (c) reduce power of 2 from 2^n to $2^{(n-1)}$
   (d) go to step 2
4. if not
   (a) write ‘0’
   (b) reduce power of 2 from 2^n to $2^{(n-1)}$
   (c) go to step 2

When looking at the algorithm I can see several things:

- I need to generate the powers of two from 2^7 to 2^0. What changes each time is the exponent–it begins at 7 and counts down to 0. To do this, I can use a loop.
- Step 2 asks a question. If the answer is true, then we do step 3. If the answer is false, then we do step 4. So ask the question in step 2 and do one thing if it is true and another if it is false I can use an `if-else`.
- In both step 3 and step 4, the algorithm calls for reducing the power of 2 from 2^n to $2^{(n-1)}$; this is handled by the loop that goes counts down from 7 to 0.

So the function is

```
void ten2binary(int num)
{
    int i, pow2;

    printf("%3d  ", num);  // write the original base-10 number

    for(i = 7; i > -1; i--)
    {
        pow2 = power(2, i);
        if(num/pow2 > 0)   /*  remember this is integer division  */
        {
            printf("1");
            num = num - pow2;
        }
        else
            printf("0");
    }
    printf("\n");
}
```

To generate the powers of two, I decided to write my own function for raising a positive integer to a nonnegative integer power. Once again, to understand this, we need to think about what this means in first grade math. When we raise a number b to the power e, we are multiplying b by itself $e-1$ times. For example, $3^5 = 3 \cdot 3 \cdot 3 \cdot 3 \cdot 3$. We also need to keep in mind that the $b^0 = 1$ by definition. How do we write a function to do this? Since I know that I need to do something $e-1$ times, then I will use a loop that runs this long:

```
int power(int base, int exp)
{
    int i, output = 1;

    for(i = 0; i < exp; i++)
        output = output * base;

    return output;
}
```

What about handling $b^0 = 1$? In the function, when `exp` is 0, then the test in the loop will fail on the first iteration. Since we begin by assigning 1 to the variable `output`, this is the value returned by the function. The output from running our program is:

```
    Please provide starting and stopping integers, each
    in the range 0 <= number < 256.

    Starting number: 4
    Stopping number: 10
      4  00000100
      5  00000101
      6  00000110
      7  00000111
      8  00001000
      9  00001001
     10  00001010
```

5 Structured Data

Example: We need to print the elements of an array. We could do this explictly:

```
int data[5] = {9, 2, -4, 0, 18};

printf("%d\n", data[0] );
printf("%d\n", data[1] );
printf("%d\n", data[2] );
printf("%d\n", data[3] );
printf("%d\n", data[4] );
```

However, we can see that we are repeating a task (print an integer from the array) in which each specific task differs slightly (the index value changes). Therefore, this would be a good use for a loop in which the loop variable is used to produce the array subscript:

```
int data[5] = {9, 2, -4, 0, 18};
int i;

for(i = 0; i < 5; i++)
    printf("%d\n", data[i] );
```

References

[Ben00] Jon Bentley. *Programming Pearls.* Addison-Wesley, Reading, MA, 2nd edition, 2000.

[RN95] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, Upper Saddle River, New Jersey, 1995.