**Control Structures**

A programming language supports a few looping and a few selection structures. It may also support some unconventional control structures.

The language ‘C’ supports the following.

a) Looping structure:

- while statement

- for statement

- do … while statement

b) selection structure:

- if statement

- switch statement

Let us start the discussion with the while statement.

Observe the file 1\_while.c

The following statement captures the structure of the while statement.

The while statement starts with the keyword while and is followed by an expression in parentheses and is followed by a statement or a block.

The expression of the while is evaluated. It could be non-zero or zero. If it is non-zero, the body of the while is executed and again the control comes back to the expression of the while. If the expression of the while is false, the loop is exited.

The body of the loop is not determined by indentation as in Python.

It is a good programming practice to always use a block even if only one statement need be in the loop.

The body of the loop is executed 0 or more times as the condition is checked top tested.

/<while stat>::= while (<expr>) [<stat>|<block>]

// expr : 0 is false; not 0 is true

// no data structure

// no indentation

// body: single statement;

// # of statements : grouped under { }

//

// top testing; execute body 0 or more times; conditional looping structure

Let us look at a few examples from the same program file.

// version 1

int n = 5;

while(n)

printf("%d ", n);

n = n – 1;

This is an infinite loop as n remains to be 5 – is always true!

// version 2

// terrible

int n = 5;

while(n){printf("%d ", n);

n = n – 1;}

This would display 5 4 3 2 1. But the program layout does not indicate the logic structure of the program. Always indent the code to indicate the logic structure.

// version 3

// indent your program

// always use a block

int n = 5;

while(n)

{

printf("%d ", n);

n = n - 1;

}

// output : 5 4 3 2 1

This is nice.

// version 4

int n = 5;

while(n)

{

printf("%d ", n--); // 5 4 3 2 1

}

Observe the value of n-- is the old value of n. When n is 1, the value of n becomes 0m but the decrement operator returns 1.

// version 5

int n = 5;

while(n)

{

printf("%d ", --n); // 4 3 2 1 0

}

Here, pre-decrement operator returns the changed value.

// version 6

int n = 5;

while(n--)

{

printf("%d ", n); // 4 3 2 1 0

}

Observe that the old value of n is used to check whether the condition is true or false and by the time, the body of the loop is entered, n would have the decremented value. The expression of while acts like a sequence point.

// version 7

int n = 5;

while(--n)

{

printf("%d ", n); // 4 3 2 1

}

In this case, the decremented value is checked for the truth of the while expression.

Compare the last two cases. In the first case, the loop is executed n times and in the second the loop is executed n – 1 times. In the first case, the value of n is -1 and in the second case, it is 0.

// version 8

int n = 5;

int f = 1;

while(n--)

{

f \*= n;

}

printf("what : %d\n", f);

The loop is executed n times. Each time, we multiply f with n. Would this find n power n? Or is it n!? You find out it yourself. I leave it to you.

Examine the program 2\_while.c.

Our requirement is to find the greatest common divisor(GCD) of given two numbers. How do we proceed to solve this problem?

Let us make our first attempt. Clearly the greatest common divisor of two numbers cannot be greater than the smaller of the two numbers.

Let us start our search with the smaller of the two numbers.

factor = (a < b) ? a : b;

We shall check if the factor divides both. If it does, our search is over. Otherwise, we shall decrease factor by 1 and try again.

while (! (a % factor == 0 && b % factor == 0))

{

--factor;

}

A few questions to think.

* Can we rewrite the expression of the while using DeMorgan Theorem?
* Is this loop guaranteed to terminate?
* How many times will the loop execute if the numbers are relatively prime?
* Can we make the program more efficient?

Instead of decreasing factor by 1, can we conceptually decrease by a bigger size. This algorithm used in the program 2\_while\_if.c is called the Euclid’s algorithm – supposed to be the oldest algorithm ever known.

Let us have a look at this program.

while (a != b)

{

if(a > b)

{

a -= b;

}

else

{

b -= a;

}

}

This algorithm states that if the two numbers are equal, that number itself is the GCD. Otherwise subtract the smaller from the bigger and repeat the exercise. This would definitely converge to the GCD faster than the earlier algorithm.

In this program, we are also using if statement.

If is followed by an expression within parentheses and then a statement or a block and then optionally followed by else and then a statement or a block.

If the expression of if is true we execute the block following if, otherwise execute the else part if any. It is always preferred to use a block even if there is a single statement in the if and else portions.

// selection:

// <if stat> ::= if (<expr>) <stat>|<block>

// <if stat> ::= if (<expr>) <stat>|<block> else <stat>|<block>

Division is repeated subtraction. Can we make this algorithm faster by using division in stead of subtraction. Here is the next version from the file 4\_while.c.

rem = a % b;

while(rem != 0)

{

a = b;

b = rem;

rem = a % b;

}

Divide a by b. If the remainder is 0, then b is the GCD. Otherwise, replace a by b and b by the remainder and repeat.

A point for you to think. Would this work if b is greater than a?

Observe one other point in this code. We have the statement rem = a % b; repeated before the loop and at the end of the body of the loop. This acts like a guard. Can we avoid repeating the code? If for some reason, we change the code, we should remember to change at both the places.

Here the ‘C’ way of writing the code.

while(rem = a % b)

{

a = b;

b = rem;

rem = a % b;

}

We can have assignment expression as the expression of the loop – not possible in Python. The loop is exited when rem becomes 0.

The following explanation pertains to the file 5\_for.c.

To find the sum of numbers from 1 to n, we can use the following while loop.

// n is a variable given small value.

int i = 1;

int sum = 0;

while(i <= n)

{

sum += i++;

}

This looping construct has clearly some initialization, some processing and some modification at the end of the loop. In such cases, we may want to use the for statement.

// <for stmt>:: for(e1; e2; e3) <block>|<stmt>

// e1, e2, e3 : expressions

// e1 : initialization

// e2 : condition

// e3 : modification

The semantics of the for statement is same as the while statement given below.

e1;

while(e2)

{

<block>|<stmt>

e3;

}

What does this code? Why is wrong?

int sum = 0;

// wrong code

for(int i = 1; i <= n; sum += i)

{

++i;

}

Observe closely. You will find that the summation starts from 2 and not from 1. We will also end up adding n + 1. Check the way a for statement executes.

This is the corrected code.

int sum = 0;

for(int i = 1; i <= n; ++i)

{

sum += i;

}

We cannot compare the forloop of ‘C’ with that of Python. Python for loop provides a mechanism to walk through an iterable. ‘C’ for statement is same as the ‘C’ while statement – we say they are isomorphic. It is a question of choice and style while writing programs in ‘C’.

Our requirement is to find what is called the digital root of a given number. The idea is to add the digits of a given number. If that sum exceeds a single digit, repeat the operation until the sum becomes a single digit. This finds use in what is called parity checking.

Check the file : 6\_do\_while.c

for(s = 0; n; n /= 10)

{

s += n % 10;

}

This loop finds the sum of digits of n. The result is in s and n would have become 0. If s exceeds 9, we would like to copy s to n, repeat this operation.

Can we put this code under a while(s > 9) { <this code> }?

The answer is a clear NO as the value of s is not initialized until we enter the inner loop? Shall we initialize to 0? Then the loop itself is not entered. How about making s 10? Looks very unnatural.

Can you realize here that the sum becomes available only after doing the summation once? We require in this case, a bottom tested looping structure which executes the code at least once.

This is the solution.

do

{

for(s = 0; n; n /= 10)

{

s += n % 10;

}

n = s;

} while(s > 9);

We have discussed all the looping constructs in ‘C’.

Let us turn our attention to the selection structure. We would to classify triangles given the 3 sides (which do form a triangle) as equilateral, isosceles or scalene. This is one possible solution.

Look at the program 7\_if.c

int count = 0;

scanf("%d %d %d", &a, &b, &c);

if(a == b) ++count;

if(b == c) ++count;

if(c == a) ++count;

if(count == 0) printf("scalene\n");

if(count == 3) printf("equi\n");

if(count == 1) printf("iso\n");

Compare every pair of sides and increment the count each time – initialized to 0 on start.

Can the count be 2?

Observe a few points here.

* We are comparing integers (integral values)
* We are comparing a variable(an expression) with constants
* We are comparing for equality (no > or < )
* In all comparisons, the same variable is used.

In such cases, we may use switch statement. Here is the example – file : 8\_switch.c

switch(count)

{

case 0: printf("scalene\n"); break;

case 3: printf("equi\n"); break;

case 1: printf("iso\n"); break;

}

The value of count is compared with case clauses. This comparison decides the entry into the switch and not exit from there. To avoid the code fall through, we use break statement. The rule of ‘C’ is “break if switch”.

We can also use default to capture when all other comparisons fail.

switch(count)

{

case 0: printf("scalene\n"); break;

case 3: printf("equi\n"); break;

default : printf("iso\n"); break;

}

These switch statements are more efficient compared normal if statements. Note that not all nested if statements can become switch statements.

Nested control structures:

We may have loops and selection nested. Here is an example to generate all Armstrong numbers between 0 and 999. The sum of cubes of digits is same as the number.

In this example, we have a loop on hundredth digit, a loop on tenth digit and a loop on unit digit. We cube the digits, find the sum, form the number using these digits and compare the sum with the number formed.

This program shows how not to write programs.

for(h = 0; h < 10; ++h)

{

for(t = 0; t < 10; ++t)

{

for(u = 0; u < 10; ++u)

{

hc = h \* h \* h;

tc = t \* t \* t;

uc = u \* u \* u;

n = h \* 100 + t \* 10 + u;

s = hc + tc + uc;

if(n == s)

{

printf("%d\n", n);

}

}

}

}

Think how many multiplications do we do for cubing? Should we repeatedly evaluate cube of t in the inner loop when it is not changing. Rearrange the code. Put the statements in the right blocks. You will be able to reduce the number of multiplications for cubing to approximately 1/3rd of what it is now,