Programming
and
Problem Solving using ‘C’

Lecture Notes

by

N S Kumar
kumaradhara@gmail.com

Introduction

We have already learnt programming and learnt programming in the language Python. In this course, we will learn programming and problem solving in 'C'. In the beginning of each unit, we will state a project or a problem to be solved and we will develop the required constructs in 'C and then at the end of the unit, we will solve the given problem and complete the given project.

1. Why should one learn C after having mastered Python? What are the advantages and disadvantages of Python over 'C'? Is not 'C' an outdated language?

We have to fill our stomach every day 3 or 4 times so that our brain and body get enough energy to function. How about eating Vidyarthi Bhavan Dosa every day? What about Fridays when the eatery is closed? Why not buy Dosa batter from some nearby shop? Or do you prefer to make the batter yourself? Would you have time to do that? Would that depend on how deep your pockets are? Would you like to decrease your medical bills?

Every language has a philosophy. The language used by poets may not be suitable for conversation. Poets use ambiguity in meaning to their advantage, and some verses in Samskrita have more than one meaning. But that will not be suitable for writing a technical report.

Python supports flexibility. Time taken for development tends to be small in Python. But Python is not necessarily efficient. If the goal is efficiency, Python cannot be the language of choice. Python is itself built over the scaffolding provided by 'C'.

The goal of 'C' is efficiency. The safety is in the hands of the programmer. 'C' does very little apart from what the programmer has asked for.

Example: When we index outside the bounds of a list in Python, we get an "index error" at runtime. To support this feature, Python runtime should know the current size of a list and should also check whether the index is valid each time we index on a list. You are all very good programmers and I am sure you never get an index error. But Python has to check even if you are right. So Python penalizes the right programmer. 'C' does not. You get what you deserve. If you are

lucky, the program crashes. Otherwise something subtle may happen, which later may lead to catastrophic failures.

So,

- C gives importance to efficiency
- C is not very safe ; you can make your program safe
- C is not very strongly typed; mixing of types may not result in errors
- C is the language of choice for all hardware related softwares – system softwares
- C is the language of choice for softwares like operating system, compilers, linkers, loaders, device drivers.

2. Is 'C' not an old language?

Yes and No.

It was designed by Dennis Ritchie – my prostrations to him –most of us working in the field of computer science owe our life to him. We use 'C' like languages and unix like operating systems both have his contribution – in 70s. But the language has evolved over a period of time. The latest 'C' was revised in 2011.

A language should be understood the same way by compiler writers and programmers all over the world. To facilitate this, the languages are standardized by international organizations like the ISO.

The language 'C' has 4 historical land marks.

- K & R 'C' : defined by the seminal book on 'C' by Kernighan and Ritchie
- C 89
- C 99
- C 11

There is one more reason to learn 'C'. 'C' is the second most popular language as of now. according to Tiobe ratings. https://www.tiobe.com/tiobe-index/.

This website gives the popularity ratings of programming languages. The rating of 'C' is more than 11% and Java is 14% as on date 8th Feb 2018.

Program Structure:

Let us summarize the characteristics of a program in 'C'

- The language is case sensitive like Python. The uppercase and lowercase are distinguished.

- The program is immune to indentation. In Python, we introduce indentation to indicate the structure of the program – be it control structure, class or function. ‘C’ follow the free format source code concept. The way of presentation of the program to the compiler has no effect on the logic of the program.

  However, we should properly indent the program so that the program becomes readable for humans. When you prepare a new dish, I am sure you will also consider how you present it to your kith and kin.

- The program has a well defined entry point. Every program has to have an entry point called main, which is the starting point of execution.

- We add comments for documentation. There are two types of comments.
  - Line comment : starts with // and ends at the end of the line
  - Block comment : start with * *and ends with* * and can span multiple lines

  Comments are ignored by the compiler.

- A program in ‘C’ normally has one or more pre-processor directives. These start with the symbol #. One of the directives is “include”.

  #include <stdio.h>

  The above line asks the translator to find the file whose name is stdio.h at some location in our file system and read and expand it at that point in the ‘C’ code.

  We will discuss later about .h or header file.

- A program in ‘C’ has to have a function called main. The function returns an int. The function may take a couple of arguments – will discuss this later. A function definition has a return type, then the function name and declaration of parameters with parentheses and a block(equivalent to suite in Python).

  The main function is invoked(called) from some external agency – like our shell. It is the duty of this function to tell the caller whether it succeeded or failed. So, by convention, main returns 0 on success and non-zero on failure.

- printf is a library function to display. The first argument is always a string. In ‘C’, string constants or literals are always enclosed in double quotes.

- A block contains a number of statements. Each statement ends in a semicolon. This symbol ; plays the role of statement terminator.

```
// name: 1_intro.c
// first program
#include <stdio.h>

int main()
{
        printf(“hello world\n”);
        return 0;
}
```

Steps involved in executing a program in ‘C’

- We enter the program into the computer. This is called **edit**ing. We save the **source program**.
- Then the source program is pre-processed. All commands starting from the symbol # are processed. The output is called a **translation unit or translation.**
- Then the translation is **compiled**. The output is called an **object file**. There may be more than one translation. So, we may get multiple object files.
- We put together all these object files along with the predefined library routines by **linking**. The output is called an **image or a loadable image**.
- Then we **load** the image into the memory of the computer. This creates a **process**.
- We **execute or run** the process. We get some results. In ‘C’, we get what we deserve!

The commands we use are as follows.

a) just preprocess:

gcc -E 1_intro.c

b) compile

gcc -c 1_intro.c

c) link

gcc 1_intro.o

d) load and run a loadable image

./a.out

Programming
and
Problem Solving using ‘C’

Lecture Notes

by

N S Kumar
kumaradhara@gmail.com

Output

We have had a look at our first program in 'C'. Let us understand how the function printf works. Please refer to the program ex1.c.

- printf("hello world\n");

  The first argument to printf is a string. The output depends on this string. String literals in 'C' are enclosed in double quotes.

  \n : is an example of escaping. This changes the meaning of 'n' - this character '\n' is the new line character – causes the cursor to go to the next line on output.

- printf("hello ", "world\n"); // hello

  Both these arguments are put on the activation record or stack frame. But the output depends on the first string. As there is no interpretation of the second parameter, it would not appear in the output.

- printf("hello %s\n", "world\n"); // hello world

  The presence of %s in the first string makes the function printf look for the next parameter in the stack and interpret it as a string.

- printf("%s %s %s %s\n", "one", "two", "three", "four");

  The function printf takes variable number of arguments. All arguments are interpreted as the number of %s matches the number of strings following the first string.

- printf("%s %s %s %s\n", "one", "two", "three");

  NO! we are in for trouble. There is no argument for the 4^{th} %s in the format string. So, printf tries to interpret whatever is there in the stack at the possible location as a string. If we are lucky, the program will crash. We have an "undefined behaviour". C does no checking at runtime!

- printf("%5d and %5d is %6d\n", 20, 30, 20 + 30);
  - Arguments to printf can be expressions – Then the expressions are evaluated and their values are passed as arguments.
  - %d : indicates to printf to interpret the next parameter as an integer.
  - %5d : tells printf to display the integer using 5 character width.

- printf("what : %d\n", 2.5);
  - GOD if any should help the programmer who writes such code!!

- undefined behaviour.

Let us digress to discuss the last statement in detail.

A few points to note about totally compiled languages like ‘C’.

- There is no translator at runtime in ‘C’. It exists in scripting languages like Python.

  As the translator exists at runtime, we can take any valid string in Python and compile and execute at runtime. You may remember the functions eval and exec. But this is not possible in ‘C’ as the compiler does not exist at runtime.
- In ‘C’, type is a compile mechanism and value is a runtime mechanism. In Python, a value has a type and the type remains at runtime. We can query any value regarding its type in Python at run time.

  Having type at runtime would require more space at runtime. Any functions based on type shall be also costly in terms of time and space. As the goal of ‘C’ is efficiency, only values are stored at runtime. There is no way to infer the type by looking at the bit pattern.
- All the code that executes at runtime should be compiled during compilation itself and cannot be added at runtime.

Let us look at our present example.

- printf("what : %d\n", 2.5);

The function printf takes varying number of arguments. In such cases, the compiler cannot match arguments and parameters for type. So, the compiler might emit some warning (if it knows what printf expects) – but cannot throw errors.

As the compiler does not know that 2.5 should be converted to an integer, it does not do any conversion.

So, the compiler puts the value of 2.5 the way it is expected to be stored – as per the standard IEEE 754. This standard uses what is called mantissa exponent format to store fractional (floating point) values.

At runtime, printf tries to interpret the bit pattern as an integer when it encounters the format %d. At runtime, no conversion can occur – we do not even know that what is stored as a bit pattern is a floating point value – Even if we know, there is no way to effect a conversion as we do not have a compiler at that point. We end up getting some undefined value.

So, in ‘C’, **you get what you deserve.**

There are a few basic types in ‘C’. int, char, long, short are called integral types and these are exactly stored in the computer. Float and double are approximately stored.

The size of a type – size required to represent a value of that type – can be found using an operator called sizeof.

The sizeof a type depends on the implementation. We should never conclude that the size of an int is 4 bytes. Can you answer this question – How many pages a book has? What is the radius of Dosa we get in different eateries?

The size of a type is decided on an implementation based on efficiency.

Variable:

This is the most important concept in all languages.

A variable has a name, a value, a location and a type. It also has something called life, scope, qualifiers etc. We will discuss the second part later in the course.

A variable has to have name by which we can refer to it in the program. Sometimes, the runtime makes variables with no names. These are called temporary variables. We cannot access them by name as we have no name in out vocabulary.

A variable has a type – like int, double. In Python, the type is based on the value assigned to the variable and the type of a variable can change at runtime. In 'C', type should specified before a variable is ever used. We declare the variable with respect to type before using it.

Example:

int a; double b;

The type of a variable can never change during the program execution. The type decides what sort of values this variable can take and what operations we can perform. Type is a compile time mechanism. The size of a value of a type is implementation dependent and is fixed for a particular implementation.

We can initialize a variable at the point of declaration. An uninitialized variable within a block has some undefined value. 'C' does not initialize by any default value.

int c = 100;

int d; // undefined value !!

A variable can be assigned a value later in the code.

c = 200;

d = 300;

Assignment is considered an expression in 'C' - whereas assignment is not an expression in Python.

Printf(“%d”, c = 400); // ok

This expression c = 400; assigns 400 to c and the value assigned to c is the value of the expression.

‘C’ follows parameter passing by value. So, we cannot call a function to change the value of a variable by passing the variable name as an argument. In such languages, we require a mechanism to pass the location – to be precise the address of the location – as argument to change the variable. Unary operator & gives the address of the variable.

Let us compare with Python. In Python, we always read the input as a string – there is no formatting of the input. We then split and pass the resultant strings to the constructors of the corresponding types. The input functions return a string back and do not change any argument – in fact there are no arguments!

We use scanf to read from the keyboard. We specify the format - %d to read an int and %f to read a float and so on.

Refer to the program ex2.c.

```
scanf("%d", a); // dangerous
scanf("%d", &a); // address operator
printf("a : %d ", a);
```

scanf like printf takes a format string as the first argument. The input should match the string. As the scanf below has a comma between two format specifiers, the input should have a comma between a pair of integers.

```
scanf("%d,%d", &a, &b);
printf("a : %d b : %d\n", a, b);
```

Operators

Refer to the program 1_operators.c.

An operator in a language has

- rank or arity: # of operands required
- precedence : order of evaluation of operators if an expression has more than one operator
- association : order of evaluation when there is cascading of operators at the same level of precedence.

Some operators exist as both unary and binary operators with different levels of precedence and association.

Example:

& : address operator and bitwise operators

* : multiplicative operator and dereferencing operator

Some unary operators can appear before or after the operand.

Some operators require a lvalue.

We shall discuss these in detail in this document.

We shall classify and discuss various types of operators of 'C'.

Arithmetic operators:

+ - * / %

There is no exponentiation operator.

All are binary operands – require two operands.

+ and – can also be used as unary operators – in such case, the operator is prefixed – appears to the left of the operand.

If the operands are int for /, the result is int quotient obtained truncation of the result. The operator % is defined on integral types only.

If the operands are mixed types(int and double) in operations + - * /, the result is of type double.

Bitwise operators:

<< >> & | ^ ~

The first 5 are binary operators and the last one is unary operator.

These operators are used on unsigned int only.

Common bitwise operations:

* multiply variable n by 2

n << 1

* check whether n is odd

n & 1

* swap two variables a b

```
a = a ^ b;
b = a ^ b;
a = a ^ b;
```

* check whether ith bit in variable n is 1

n & (1 << I) : 0 implies not set and non 0 implies set

* set the ith bit in n

n = n | 1 << i;

* clear the ith bit in n

n = n & ~(1 << I)

Assignment combined with arithmetic | bitwise operators:

+= -= *= /= %= <<= >>= &= |= ^=

a += b is same as a = a + b.

As the same variable occurs on both sides of assignment, the compiler may be able to produce efficient code in the former case.

Increment and decrement operators:

These are unary operators ++ and - -. These can be prefixed or postfixed. These are short cuts for assignment. Any assignment operator not only gives a value, but also modifies a variable. A pure expression gives a value without modifying any variable. All assignment operators have side effects – they modify the variable.

The difference between the post and the pre operators manifest only when the value is used.

Both a++ and ++a would increment a by 1. The return value in case of post increment is the old value and pre increment is the new value.

Post operation : use and change.

Pre operation : change and use.

Check the code below from 1_operators.c.

```
int e;
// change the variable by 1
```

```
// used as stmt; there is no diff
e = 100;
e = e + 1;
printf("%d\n", e);
e += 1;
printf("%d\n", e);
e++; // post increment op
printf("%d\n", e);
++e; // pre increment op
printf("%d\n", e);

// used as expressions
// preincrement
e = 100;
printf("%d\n", ++e); // 101
printf("%d\n", e); // 101
// postincrement
e = 100;
printf("%d\n", e++); // 100
printf("%d\n", e); // 101
```

Lvalue and Rvalue concept:

Let us look the following code from 2_operators.c

```
int a = 10;
int *p = &a;
printf("pointer to a : %p value \n", p, *p); // hex number 10
int b = 20;
*p = 30;
printf("a : %d\n", *p); // 30

p = &b;
printf("pointer to b : %p value \n", p, *p); // hex number 20
```

The variable a is of int type and (int *p;) p is a pointer to int. We always read a declaration right to left. A pointer variable of some particular type can hold the address of a variable of the same type.

& : in unary form – gives the address of the variable.

p = &a; assigns the address of a to p.

We can get back a through p by using * : in unary form - dereferencing operator.

*p is same as a.

We can change a by assigning to *p.

*p = 30;

We can also change p itself by assigning a pointer to b to p.

p = &b;

We talk about l-value and r-value with respect to assignment operator =.

r-value refers to whatever that could occur to the right of assignment operator.

L-value refers to whatever that could occur to the left of assignment operator.

A constant has only r-value. An initialized variable has both l and r value. An expression of the form a + b is only a r-value. The only operator which gives a value back is the dereferencing operator *.

Evaluation of an expression:

An expression consists of operands and of operators. How is an expression evaluated?

There are two parts in it.

1. Evaluation of operands :

This is fetching the operands from the memory of the computer to the registers of the CPU. All the evaluations happen within the CPU. This order is not defined.

The idea is to allow the compiler to optimize fetching of the operands. The compiler may generate code such a way that if an operand is used more than once, it may get fetched only once.

2. Evaluation of operators:

This follows the rules of precedence and if more than one operator has the same level of precedence, follows the rules of association.

Does the first rule have any effect on our programs? Yes. If affects our programs if the expressions we use are not pure and have side effects.

All assignments have side effects.

Let us consider an example.

a = 10;

b = a * a++;

The value of a++ is the old value of a and therefore 10. What is the value of the leftmost a? Is the value evaluated before incrementing or after incrementing? It depends on the compiler and the compiler options. So value of leftmost a is undefined. So, value of b is undefined.

```
//    10 * 10 => 100
//    11 * 10 => 110
//        undefined; depends on the order of evaluation of operands
//  bad code
```

Sequence Point:

How do we write safe and proper code in 'C' if operators have side effects and the variable values are not defined in such cases? The language 'C' also specifies points in the code beyond which all these effects will definitely be complete. These are called sequence points.

We shall mention sequence points at appropriate places.

Relational operators:

The operators are < > <= >= == !=.

The result of relational operation is 1 or 0 – 1 stands for true and 0 for false.

The relational operators should not be cascaded – not logically supported - 'C' is not as good as Python in Math!

5 == 5 == 5 is false !!

5 == 5 == 5 becomes 1 == 5 ; this becomes 0 !!

Logical operators:

! stands for not

&& stands for and

|| stands for or.

In 'C', 0 is false and any non-zero value is true.

As in Python, C follows short circuit evaluation. Evaluation takes place left to right and evaluation stops as soon as the truth or falsehood of the expression is determined.

&& and || are sequence points.

Consider the code from 2_operators.c.

// a = 10;

// a + a++ undefined

// to support short ckt evaluation, && becomes a sequence point

// expression before &&, || will be completely evaluated.

// a++ == 10 && a == 11 // ok; will be true 1

a + a++ is undefined as the value of left a is not defined.

a++ == 10 && a == 11 will be true.

The value a++ is 10 and a becomes 11 before the evaluation moves across the sequence point &&.

if expression:

An expression has a value and a statement does not. If expression is a ternary expression requiring three operands.

E1 ? E2 : E3

where E1, E2 and E3 are expressions.

The expression E1 is first evaluated. If it is true, then the value of the expression is E2 else it is E3.

? of this expression acts like a sequence point.

There are a few more operator not discussed – sizeof (<cast>) ….

We shall stop at this point.

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,

Character input and output

A character in programming refers to a single symbol – a smallest possible string. The characters in 'C' have no relationship with characters in the plays of Shakespeare even though those characters utter characters we could process in programming!

A character in 'C' is like a very small integer having one of the 256 possible values. It occupies a single byte. We code English alphabets A as 65, B as 66, Z as 90, a as 97 and so on in a coding scheme called ASCII. We have no concept of character type in Python – everything is a string there.

For the first unit, we are required to find the number of characters, number of words and number of lines in a given file. We want to simulate a program in Unix called wc – word count program.

We have not learnt as yet how to play with files in our programs. It is a bit too early in this course. We do know how to read from the keyboard. Can the operating system open a file for me and make it logically available on the keyboard? It can. This concept is called input redirection.
When we run our command cmd from the unix shell and specify <filename on the same line, our program cmd reads from the file whenever it reads from the keyboard.

input redirection:
$ cmd <filename

It is similarly possible to collect the output of the program which would appear on the screen in a file using output redirection.
output redirection:
$ cmd >filename

To complete this project, we should learn
a) how to read and display a character?
b) how to read a line?
c) how to make out when we reach the end of file?
d) how to break a given sequence of characters into words?

Let us check the program : ex1.c

To read a character from the keyboard, we could use scanf(“%c”, &x);

We could also use x = getchar();

We prefer the second as it is developed only for handling a single char and therefore more efficient even though it is not generic.

Similarly we prefer putchar(x) over printf(‘%c”, x);

This code shows how to read two characters and display them.

```
#if 0
scanf("%c", &x);
scanf("%c", &y);
printf("x:%c y:%c\n", x, y);
#endif
x = getchar();
y = getchar();
putchar(x);
putchar(y); printf("%d", y);
```

Let us try to read a line and display.

Let us examine the program ex2.c.

```
char ch;
while(ch != '\n')
{
        ch = getchar();
        putchar(ch);
}
```

This is a terrible code. The first time we enter the loop, ch is not initialized. If you are lucky, the loop is never entered!

In these cases, we require reading before the loop and reading at the end of the body of the loop.

```
ch = getchar();
while(ch != '\n')
{
```

```
        putchar(ch);
    }
```

Infinite loop unless the first character is new line!!

```
    ch = getchar();
    while(ch != '\n')
    {
        putchar(ch);
        ch = getchar();
    }
```

This is fine. But we can as well use assignment expression in the while.

This is ‘C’ code!!

```
    while( (ch = getchar()) != '\n')
    {
        putchar(ch);
    }
    putchar('\n');
```

A few points to observe. The input from the file or the keyboard is stored in something called the buffer. That is transferred to getchar only afte enter key is pressed.

The next hurdle is to read the whole file. How do we know we have reached the end of file. ‘C’ handles this in a very interesting way. When the end of file is reached – which the operating system can make out – a particular value gets is returned by getchar. This value is decided by ‘C’ and not by the operating system. This value is associated with the name EOF. So, we keep reading until getchar returns EOF.

```
// file: ex3.c
    while( (ch = getchar()) != EOF)
    {
```

```
        putchar(ch);
}
```

Let us examine ex4.c – the first attempt to count the number of lines and number of characters.

```
char ch;
int nl = 0;
int nw = 0;
int nc = 0;
while( (ch = getchar()) != EOF)
{
        ++nc;
        while((ch = getchar()) != '\n')
        {
                ++nc;;
        }
        ++nl; ++nc;
}
//putchar('\n');
printf("# of char : %d\n", nc);
printf("# of lines : %d\n", nl);
```

This could work but not preferred. Observe that we are taking input in two places. What if we reach EOF in the inner loop?

It is always preferred to read at only one place.

It is always preferred to have a while with an it over

while with a while within.

This program ex5.c uses the following logic.

Count the character each time we enter the loop.

Count the lines if the character is newline.

Count the words assuming that the words on a newline or space or tab – any white space.

```
char ch;
int nl = 0;
```

```
int nw = 0;
int nc = 0;
while( (ch = getchar()) != EOF)
{
        ++nc;
        if(ch == '\n')
        {
                ++nl;
        }
        if(ch == ' ' || ch == '\n' || ch == '\t')
        {
                ++nw;
        }
}
//putchar('\n');
printf("# of char : %d\n", nc);
printf("# of words : %d\n", nw);
printf("# of lines : %d\n", nl);
```

What if there are multiple spaces between the words? Then we are in for trouble.

Can we use a boolean variable to indicate whether we are in a word or not. If we are in a word and we reach a space, the word ends and we count. If we are not in a word and we encounter a space, we ignore it. We set in word when we encounter a non- white-space.

Let us have a look at the program ex6.c.

```
int inword = 0; // not in a word so far
while( (ch = getchar()) != EOF)
{
        ++nc;
        if(ch == '\n')
        {
                ++nl;
        }
        if(inword && (ch == ' '|| ch == '\t' || ch == '\n'))
```

```
                {
                        inword = 0; ++nw;
                }
                // avoid recomputation of white space concept
                else if (!(ch == ' '|| ch == '\t' || ch == '\n'))
                {
                        inword = 1;
                }

        }
```

You may compile this and run this against any text file say myfile.txt.

./a.out < myfile.txt

Compare with

wc myfile.txt

Both should give the same result.

I have tried on the collection of Shakespeare's works.

$ wc s.txt

 124210 899680 5447737 s.txt

$ gcc ex6.c

$ ./a.out <s.txt

# of char : 5447737

# of words : 899680

# of lines : 124210

Here we end the unit I.

Wish you happy learning 'C'.

Function

Mathematically, a function is a subset of relation. Relation is a subset of cartesian product of sets. A function associates a sequence of values – called arguments, with a value – called the result of the function application or the call. In Mathematics, the arguments will not change when the function is applied. Not only that, each time the function is applied to the same sequence of arguments, we get the same result. Such a function is said to be a pure function. It does not depend on Rahu kaala or gunika kaala. It does not depend on the gender or the mood of the person applying it!

In programming, a function is a sequence of code. When the function is called or invoked with arguments, that sequence of code will be executed and the result is given back to the caller- one who invoked the function – by the callee – one who got called.

We use functions in programming for a number of reasons.

- Divide and conquer:
    - We can deal with complex problems in one go. We may want to divide the problem into small parts and solve each one of them
- sharing:
    - We can develop solutions to commonly occurring problem. These solutions can be grouped together in what is called a library. We can avoid reinventing the wheel.
- Maintenance:
    - A code may be required at many places. By putting in a function, we avoid redundancy of code. This helps in maintenance. Any updates need be done at only one place.
- Reliability:
    - Functions which become part of libraries are normally well tested and are made flexible and efficient. We can use these functions with the guarantee of correctness and efficiency. We are not normally very sure of our functions !

- Reuse
    - A function may be used again and again. We do not have to keep writing the function.

Function Definition:

The structure of function definition is as follows.

returntype fnname(<list of parameters with type>)

{

<stmt>

return <expr>

}

A function should have a return type. A function always returns a value of a particular type. The compiler should provide a temporary location of this type to hold the result before it is consumed by the caller. The compiler may also convert the expression of return to that of the return type if they do not match.

If a function is not required to return a value at all, then the function return type is made void.

The function name follows the rules of an identifier.

Following the function name, with in parentheses, we specify the parameter list. A parameter list has a list of parameters comma separated. Each parameter is declared with respect its type. When a function is invoked, these parameters are created on an activation record or stack frame. When the function returns, this stack frame or activation record is removed or popped out.

Parameters are always variables. They are initialized on the function call with arguments being copied to the parameters. Parameters exist only within the function call. There is no concept of closure as in Python here.

A function may have no parameters – not at all common. Even then the parentheses are required.

The body of the function follows the parameter list.

Refer to ex0.c for some examples of function definitions and function calls.

```
int f1(int x)
{
    return x * x;
}
```

```
int f2(int x,  int y)
{
	return x + y;
}

double f3(int x)
{
	return sqrt(x);
}

void f4()
{
	printf("f4 says hello\n");
}
int main()
{
	// calling f1
	int a = 25; int b = 10;
	int res;

	res = f1(a); printf("res : %d\n", res);
	res = f1(11); printf("res : %d\n", res);
	res = f1(11 + a); printf("res : %d\n", res);
	// res = f1(int); // error

	res = f2(a, b); printf("res : %d\n", res);

	printf("sqrt : %lf\n", f3(25.0));
	printf("sqrt : %lf\n", f3(a));

	f4();
}
```

Function call:

fnname(arguments …)

A function call has the function name followed by parentheses within which we have a comma separated list of arguments. Arguments are expressions.

When a function is called, the following actions take place.

- An activation record or a stack frame is created. This contains the following.
  - Return address – a place to return when the function execution is complete.
  - Parameters
  - local variables
  - temporary variables
  - location for return value – of return type
- Arguments are evaluated. Arguments are always expressions. The order of evaluation of arguments is not defined.

  a = 10; f(a++, a); may result in any of the following calls.

  f(10, 10) or f(10, 11)
- Arguments are copied to the corresponding parameters.
- The control is transferred to the called function or the callee.
- The function body(called) is executed.
- When the function executes a return statement or reaches the end of the function body, the callee returns to the caller.
- If the function has a return statement of the form return <expr>, the value of the expression following return is evaluated and is copied to the return location.
- The caller picks up the return value. That is the value of the function call. A function call is an expression.
- If a function is a void function, the control is returned from the callee to the caller and the last two steps do not happen.

Declaration and Definition:

This is a function definition.

```
int f2(int x,  int y)
{
        return x + y;
```

}

Can we call the function before definition?

Can we call a function defined in another file?

How would the compiler know

- the number of arguments to pass
- the types of arguments to pass
- the order of arguments to pass
- the return type?

The compiler does not. Normally by default the return type is assumed to be an int. The compiler cannot check for any of these unless it knows what the function expects. We provide this information by declaring the function before calling.

int f2(int x, int y);

This is a function declaration. It is also called prototype or signature or specification of the function. This concept came to C from C++.

There are two terms which we should understand clearly.

Interface and Implementation.

The function declaration is the interface. It tells us what the function expects. It does not tell us how the function works. If we change the function definition, the user or the client of the function is not affected.

```
int f2(int x,  int y)
{
        int temp = x+ y;
        return temp;
}
```

Parameter Passing mechanism:

In 'C', the argument is copied to the corresponding parameter. The parameter is not copied back to the argument. This is not possible if the argument is not a l-value.

Changing the parameter will not affect the corresponding argument.

Let us examine the code from ex2.c.

```
void foo(int x); // declaration
```

```
    int a = 10;
    foo(a);
    printf("a : %d\n", a); // 10
```

Observe that the variable a has not changed. It is impossible for the function foo to change the argument a.

```
void foo(int x); // declaration
    double b = 2.5;
    foo(b);
    printf("b : %lf\n", b); //  2.5
```

In this case, as the type of argument does not exactly match the type of parameter, the argument is cast into an integer.

```
foo((int)b);
```

But the value of the variable b is not affected either by casting or by the function call.

How to change arguments by calling function?

As the parameter passing is only by value, we may pass explicitly a pointer to an argument. Then the parameter becomes a pointer to the argument. Dereferencing the pointer gives us lvalue of the argument. By assigning to the dereferenced pointer, we can change the corresponding argument.

```
    int a = 10;
    int *p = &a; // pointer variable
    what(p);
    printf("a : %d *p : %d p : %p\n", a, *p, p);  // 100 100 ...

void what(int* q)
{
    int temp = 100;
    *q = temp;
}
```

Remember that you pass pointer to variables in scanf so that the variables are changed before scanf returns.

What if we just change the pointer? Let us look at this piece of code.

```
void bar(int* q)
{
        int temp = 100;
        q = &temp;
}
        int a = 10;
        int *p = &a; // pointer variable
        bar(p);
```

This will not change a. q gets a copy of pointer to a. That copy is changed to point to a local variable. So, clearly the variable a and variable p are not affected. If we want to change a pointer by calling a function, we should pass a pointer to a pointer. The corresponding parameter will be a pointer to pointer.

Return mechanism:

The return is always through a temporary. The expression of return is computed and copied to the temporary variable by the callee. The type of return comes into play here. If the expression of return is not same as the return type of the function, the expression is cast to the return type before copying to the temporary location. The caller will pick up the return value from this temporary and use it the way it wants.

The functions in 'C' return a single value. It could also be a pointer. Let us look at a couple of examples.

This function receives a pointer to a variable and returns the same. Not a very meaningful example. This shows how we can return a pointer from a function.

```
int* one(int* x)
{
        return x;
}
int c = 1000;
int *p = one(&c);
printf("%d \n", *p); // 1000
```

How about this?

```
int* two()
{
        int x = 111;
        return &x;
}
p = two();
printf("%d \n", *p);
```

x is a local variable. Local variables will be part of the stack frame which is created when the function is called and destroyed when the function returns. We are returning a pointer to such a variable. The pointer p in the caller will point to a variable which does not exist. This is a classical example of dangling pointer. The pointer exists; the location to which it points does not. There is no location; but we have access to it.

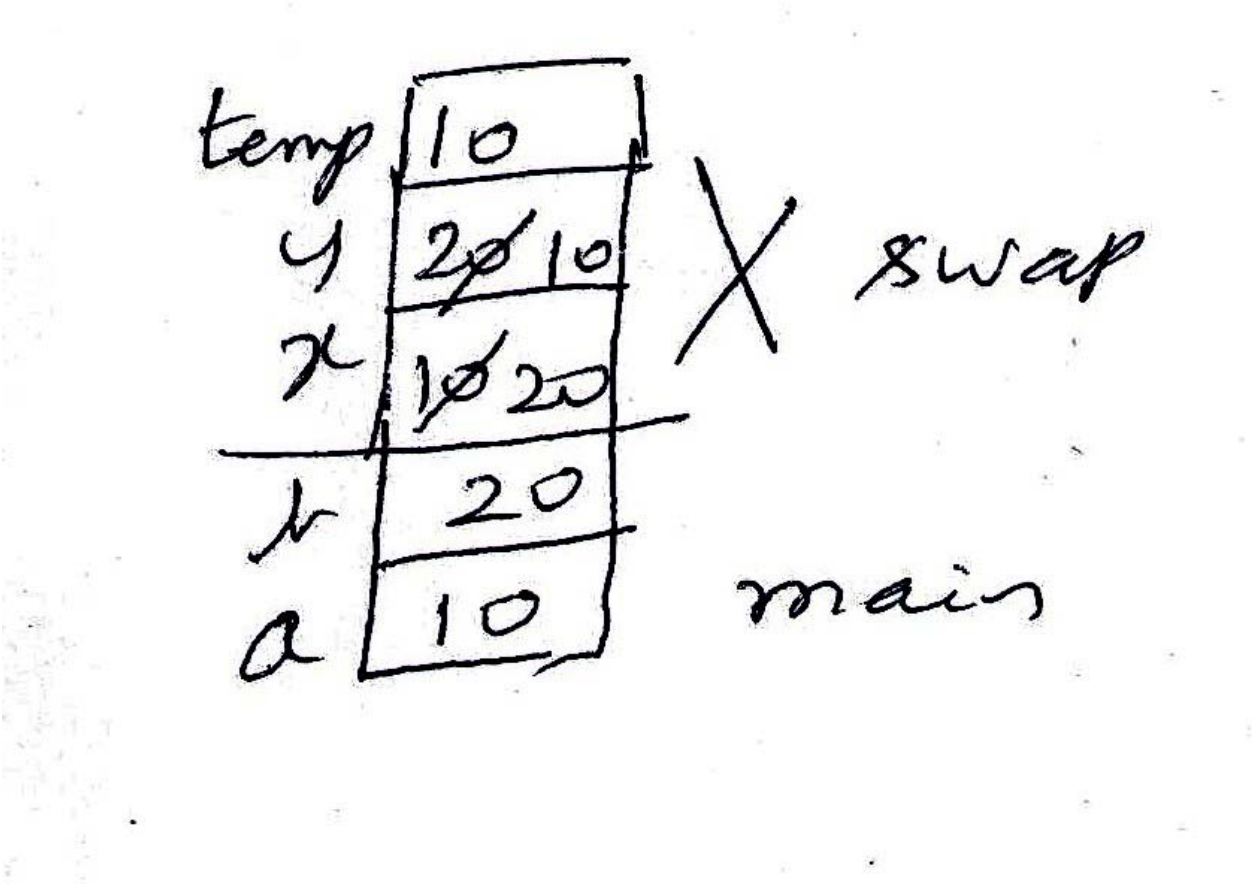
Dereferencing such a dangling pointer is extremely dangerous. This clearly is an undefined behaviour.

We will discuss function returning pointers later in the course again.

Function

Let us continue our discussion of functions with the following example from 1_swap.c.

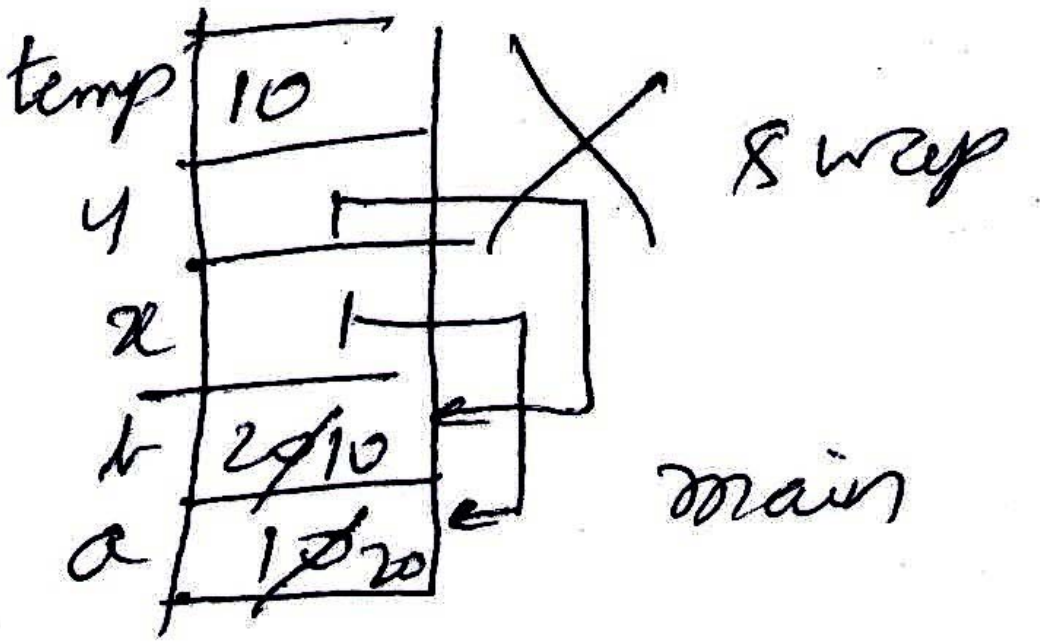
```
// Does not swap the arguments!!
void swap(int x, int y)
{
        int temp = x; x = y; y = temp;
}

int main()
{
        int a = 10; int b = 20;
        swap(a, b);
        printf("a : %d b : %d\n", a, b);
}
```



We know that the parameter passing in 'C' is by value. So, the copies of a and b get swapped; but not a and b.

To change a variable of the caller, the callee should receive pointer to the variable.

This works!




```
void swap(int *x, int *y)
{
	int temp = *x; *x = *y; *y = temp;
}

int main()
{
	int a = 10; int b = 20;
	swap(&a, &b);
	printf("a : %d b : %d\n", a, b);
}
```

Recursion:

A function calling itself directly or otherwise is said to be recursive. We may use if and recursion instead of a looping structure. In recursion, we try to express the solution to a problem in terms of the problem itself, but of a smaller size. We also require solution for one or more simple cases – called escape hatch or base case.

Let us examine how to trace a recursive function.

In this example, there are two places where recursive calls are made. We should know where these recursive calls will return. We can mark the position of the call with the number of the call – so that we will know where to resume the following of the code.

This function finds the number of 1s in the binary representation of the number.

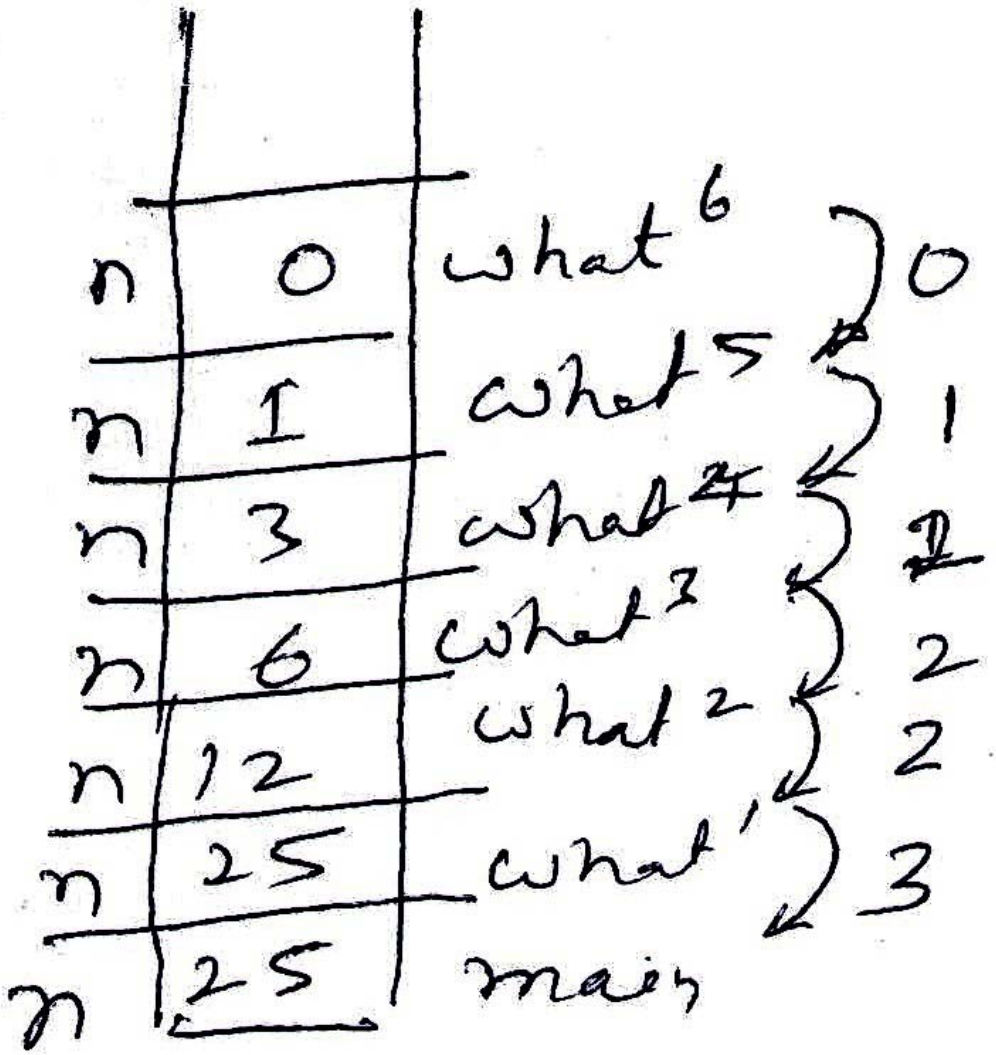
If the number is odd, we count 1 and remove the unit bit and call the function on the changed number.

If the number is even, we remove the unit bit then we call the function on the changed number.

If the number is 0, we return 0.

```
int what(int n)
{
	if(n == 0)
	{
		return 0;
	}
	else if(n % 2)
	{
		return 1 + what(n / 2);
		//	2 => 2
		//			5 => 1
		//			6 => 0
	}
	else
	{
```

```
        return what(n / 2);
        //              3 => 2
        //      4 => 2
    }
}
```




```
int main()
{
    int n  = 25;
    printf("val : %d\n", what(n));
```

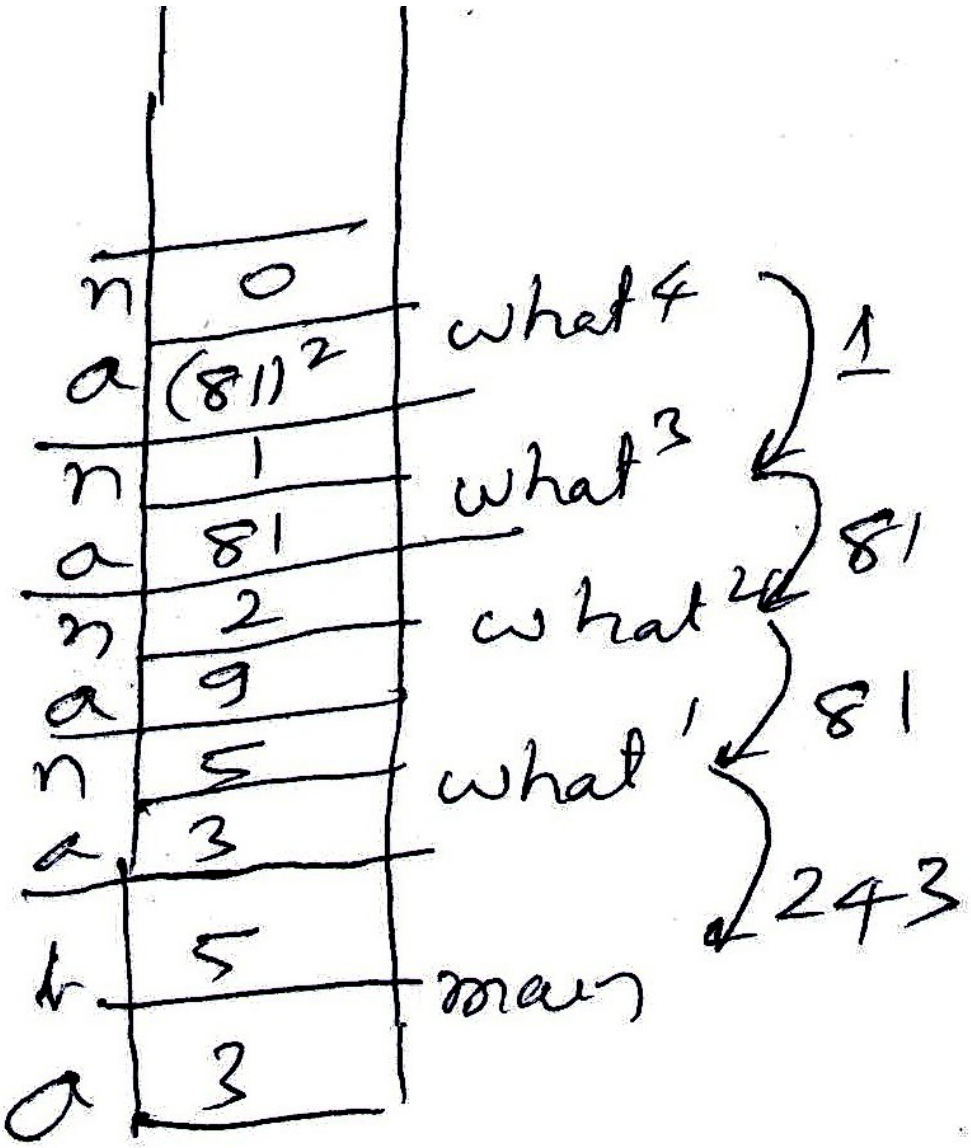
```
	//                                              1 => 3
}
```

Recursion: Another example:

This is the file 3_what.c.

What does this recursive function do? You may want to pause and think about this function.

```
#include <stdio.h>
int what(int a, int n)
{
	if(n == 0)
		return 1;
	else if(n % 2)
		return a * what(a * a, n / 2);
	else
		return what(a * a, n / 2);
}
int main()
{
	int a = 3; b = 5;
	printf("%d\n", what(a, b));
}
```



It finds a to the power of n. The definition is as follows.

If n is 0, the result is 1.

if n is even, square a and then exponentiate the result to n / 2.

If n is odd, square a and then exponentiate the result to n / 2 and then multiply this with a.

Parameter passing and pointers:

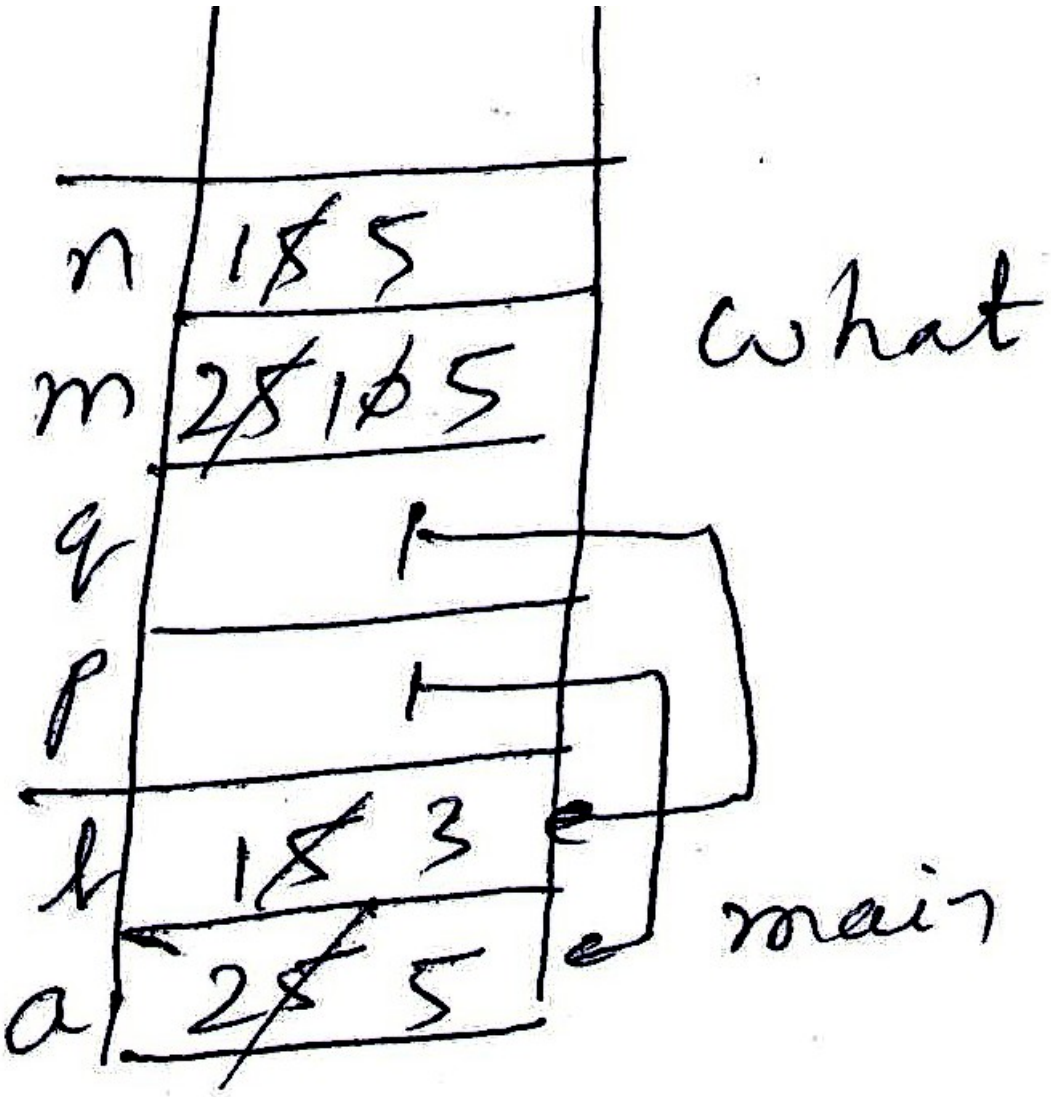
This is an interesting example of understanding parameter passing.

The loop of this function finds the greatest common divisor of m and n. p and q point to the corresponding arguments. m and n are copies of *p and *q and have nothing to do with the arguments.

*p /= m; *q /= n;

This statement causes division of the arguments by the greatest common divisor.

```
void what(int *p, int *q)
{
	int m = *p; int n = *q;
	while(m != n)
	{
		if(m > n)
		{
			m -= n;
		}
		else
		{
			n -= m;
		}
	}
	*p /= m; *q /= n;
}
int main()
{
	int a = 25; int b = 15;
	what(&a, &b);
	printf("%d %d\n", a, b);
	what(&a, &a);
	printf("%d\n", a);
}
```

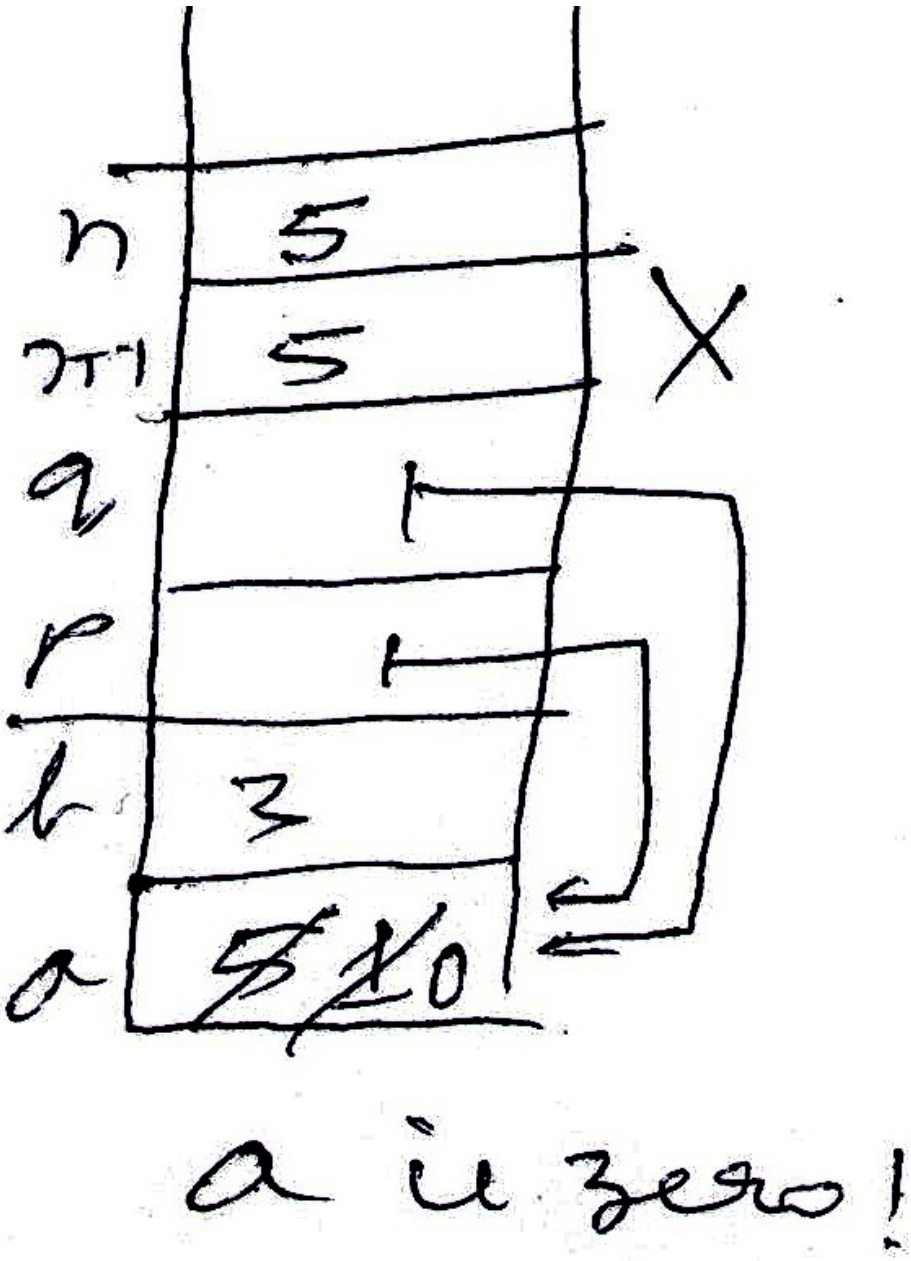



```
int a = 25; int b = 15;
what(&a, &b);
printf("%d %d\n", a, b);
```

So, a and b get divided by 5. So a becomes 5 and b becomes 3.

The next part of the code is interesting. What do you expect the variable a to become. - 1 ? Be in for a surprise!

Both p and q point to the same variable. This is called aliasing. m and n become 5. The loop is not entered. We update *p that is same as variable a by dividing by m. So a becomes 1. We update *q that is also a by dividing by n. So, a becomes 0!!



Separating the interface from implementation:

Our requirement let us say is to check whether a given number is a palindrome.

We can write the function ourselves. But we could use a function in the library or written by somebody as long as it satisfies the requirement.

Let us know that there is a function with the following signature.

int is_palindrome(int n);

Then we can write the code like this.

```
int n;
```

```
	scanf("%d", &n);
	// check whether it is a palindrome
	if(is_palindrome(n))
	{
		printf("%d is a palindrome\n", n);
	}
	else
	{
		printf("%d is not a palindrome\n", n);
	}
```

The writer of the function does not know how we use. We do not know how the developer of the function has written the function. The developer may write the logic together in the same function or call some other function to reverse the number.

```
int rev(int n)
{
	int r = 0;
	while(n)
	{
		r = r * 10 + n % 10;
		n /= 10;
	}
	return r;
}
int is_palindrome(int n)
{
#if 0
	int rev = 0; int m = n;
	while(n)
	{
		rev = rev * 10 + n % 10;
		n /= 10;
	}
```

```
        return m == rev;
#endif
        return n == rev(n);
}
```

So, we separate the client code and server code. The common code is put in the header file. It contains declaration of functions. The header file acts like the interface between the client and server.

Multiple files and make file:

The next example has 3 files.

6_server.c – provides implementation of squaring and cubing

6_client.c – uses these functions squaring and cubing

6_server.h : interface for these functions.

We compile both the .c files then link them to create the loadable image.

Each time we change any of the files, we should remember to recreate the files by using appropriate commands.

Can we transfer these two ideas to a tool?

- dependency of files
- command(s) to be executed when the dependent files change.

This tool called the make tool takes care of these.

This is the file : mymakefile.mk

The first line indicates the file created last in the sequence – the loadable image(executable). This is called a target. Then this line indicates which all file it depends on. They are called sources.

The next lines(s) start with a tab and any command to be executed when the sources are modified.

This sequence of dependencies and commands repeat.

The command to be executed is : make -f mymakefile.mk.

```
 a.out : 6_client.o 6_server.o
        gcc 6_client.o 6_server.o
6_client.o : 6_client.c 6_server.h
        gcc -c 6_client.c
6_server.o : 6_server.c 6_server.h
```

```
gcc -c 6_server.c
```

Experiment changing the files and observe that only required files are re-created.

Array

Array is somewhat similar to list in Python. There are also differences.

Array has the following characteristics.

- has # of elements
- all the elements should be of the same type – homogeneous
- size of the array is specified at the point of creation
  - size is given as a constant at compile time
  - size could be specified at runtime in C99 VLA : variable length array
- array cannot grow or shrink
- elements are selected based on index or subscript
- index is an integer; starting value is 0 – zero based indexing
- supports random access
  - time required to access the element does not depend on its position
  - is always same
- Array at compile time is an array. It knows everything about itself.
- Array at runtime degenerates to a pointer.
  - At runtime, the size of the array is not stored.
- Index out of bounds – accessing elements beyond the size of the array cannot be checked.
  - Checking would require the size of the array at runtime – extra memory
  - checking would incur some runtime cost.
  - This if done would be penalizing the right programmer
- So, 'C' follows the philosophy. The programmer gets what he deserves!
- C array at runtime is a constant pointer
- So, arrays cannot be assigned one to another. Arrays are not assignment compatible.

You may read the following notes along with the file 1_array.c.

This is the way to declare an array of 5 elements and initialize.

```
int a[5] = {11, 22, 33, 44, 55};
//printf("what : %d\n", a[5]); // undefined
```

In this array, the elements are in the position 0 to 4.

We can access the element as a[2]. This can be used as r-value or l-value.

What happens if we access outside the array bounds?

What happens if we access a[5]?

There is no checking at runtime. This is an undefined behaviour. The program may crash if you are lucky. You may get anything in the world. You get what you deserve.

Array initialization:

int a[] = {10, 20, 30, 40, 50};

The compiler will count the # of elements in the initializer and make the array have so many elements. This array will have 5 elements.

int b[5];

This array will have 5 uninitialized elements when declared within a block.

int c[5] = {11, 22};

When the array is partially initialized, the remaining elements become 0.

Array Traversal:

a) using indexing

```
int a[] = {10, 20, 30, 40, 50}; int n = 5;
 for(int i = 0; i < n; ++i)
{
        printf("%d ", a[i]);
}
printf("\n");
```

b) Using pointer: version 1

As the array is a constant pointer, we can initialize a pointer variable with an array or we can assign an array to a pointer variable.

```
int a[] = {10, 20, 30, 40, 50}; int n = 5;
int *p = a;
```

or

```
int *p; p = a;
```

We can then use p wherever we use the array name a.

```
 for(int i = 0; i < n; ++i)
{
        printf("%d ", p[i]);
}
```

```
printf("\n");
```

c) Using pointer: version 2

We can also use pointer arithmetic to access the elements.

These are the valid pointer arithmetic.

- Add an int to a pointer
- Subtract an int from a pointer
- take difference of two pointers when they point to the same array.

No other operations are allowed. We should never treat pointer as an integer.

Adding an integer to a pointer advances the pointer to the ith integer from there.

```
int a[] = {10, 20, 30, 40, 50}; int n = 5;
int *p = a;
 for(int i = 0; i < n; ++i)
{
        printf("%d ", *(p + i));
        // we can also use *(a + i)
}
```

d) Using pointer: version 3

The pointer can be incremented; array cannot. Array is a constant pointer.

There is a difference between these two expressions.

(*p)++ and *p++.

In the first case, *p is incremented. The value of the expression is the old value of *p.

In the second case, p is incremented. The old value of p is stored in temporary and then dereferenced.

We can use the following code for displaying the array.

```
int a[] = {10, 20, 30, 40, 50}; int n = 5;
int *p = a;
 for(int i = 0; i < n; ++i)
{
        printf("%d ", *p++);
}
```

d) Using pointer: version 4

As the array and pointer have the same value, can we replace p by a?

```
int a[] = {10, 20, 30, 40, 50}; int n = 5;
int *p = a;
 for(int i = 0; i < n; ++i)
{
        printf("%d ", *a++);
}
```

The answer is NO. Array cannot be changed.

Passing array as an argument to a function :

When an array is an argument, the corresponding parameter is a pointer. The parameter gets the value of the array.

The parameter is not an array. It can never be an array.

We can ask the parameter whether it refers to an array. There is no way to find the number of elements in the array through the pointer.

These are the possible ways to send array as argument.

a) array and its size

b) pointer to the beginning and to the end of the array

c) have a logical element to mark the end of the array – this concept is called sentinel.

In this example, we will pass the array and its size.

```
void read_array(int x[], int n);
void disp_array(int *x, int n);
```

int x[] and int* x are same.

These are the routines to read and display the arrays.

```
void read_array(int x[], int n)
{
        for(int i = 0; i < n; ++i)
```

```
        {
                scanf("%d", x++); // &x[i]   x + i  x++
        }

}

void disp_array(int *x, int n)
{
        for(int i = 0; i < n; ++i)
        {
                printf("%d ", *x++); // x[i]   *(x + i)
        }
        printf("\n");
}
```

check these files.

2_array.h 2_util.c 2_util.h makefile

Array & Pointer

Let us recollect why we require pointers and how arrays and pointers are always together.

a) Pointers are required to mimic parameter passing by reference. If we have to change a variable by calling a function, we have to pass pointer to the variable. The corresponding parameter will be of pointer type.

b) Array degenerates to a constant pointer at runtime. So, we can assign an array to a pointer. When we pass an array as argument to a function, the corresponding parameter is a pointer. Parameter can never be an array.

Do not ever worry about the value of a pointer.

Understand the following expression clearly.

a)

int a[10]; int *p = a;

int i = <some value between 0 and 9>;

a[i], *(a + i), p[i], *(p + i) are all equivalent.

a + i, &a[i], &p[i], p + i are all equivalent.

b)

int a[] = {11, 22, 33, 44};

int *p = a + 2;

printf("%d", (*p)++);

The value of the expression is the old value of *p which is33. Then *p is incremented. So, a[2] becomes 34.

c)

int a[] = {11, 22, 33, 44};

int *p = a + 2;

printf("%d", *p++);

* and ++ are both unary. The association is right to left. The variable p is incremented. The value of p++ is the old value of p as it is post increment operator. That old value of p stored in a temporary is dereferenced.

Display is 33. p will point to a[3].

Let us have a look at the following code.

```
int a[] = {11, 22, 33, 44, 55};
// a : array at compile time
int n = sizeof(a) / sizeof(*a); // sizeof(a) / sizeof(int)
printf("n : %d\n", n);
```

sizeof is a compile time operator. sizeof of an array returns the number of bytes occupied by the array. sizeof array / sizeof component gives us the number of elements in the array. But this can be applied at the place where the array is declared. We cannot use it on parameters as the parameter is always a pointer when the corresponding argument is an array – the parameter does not know that it refers to an array. At run time, array does not keep track of its size.

The group of program files (1_array.c, 1_array,h, 1_client.h, 1.mk) shows how to pass array as an argument and display it.

Strings:

The language ‘C’ distinguishes between character and string. ‘C’ does not have any basic type called string even though there are functions to play with strings.

‘a’ : character

“a” : string

A string is like an array of characters. Unlike normal arrays, strings are terminated by a character called NULL character whose ASCII value is 0. It is same as ‘\0’. So character ‘a’ occupies 1 byte where as string “a” occupied 2 bytes - ‘a’ ‘\0’.

Let us look at some examples from the file 2_c_str.c.

The string “pes univ” occupies 9 characters. One char in the array a is not used.

```
char a[10] = "pes univ";
```

The compiler counts and allocates an array of 7 characters (and not 6!) for array b.

```
char b[] = "python";
char c[] = { 'p', 'y', 't', 'h', 'o', 'n', '\0' };
```

The above two statements are equivalent. Initializing a c string as in the case of array b is a short cut available only for c strings.

We can read and write c strings using %s format in scanf and printf. scanf with %s option will introduce NULL character at the end of the string. Printf will display the characters until NULL character is encountered.

```
printf("str a : %s\n", a);
printf("str b : %s\n", b);
printf("str c : %s\n", c);
```

Effectively the string d is “cat”.

```
char d[] = { 'c', 'a', 't', '\0', 't', 'l', 'e', '\0' };
printf("d : %s\n", d);
for(int i = 0; i < 8; ++i)
{
        putchar(d[i]);
}
putchar('\n');
```

If the string is hand crafted, it is our responsibility to end the string by NULL character. All builtin string functions expect this sentinel value ‘\0’ failing which the results are undefined.

Let us write three functions to get a feel how these string functions operate.

Check the files : 2_c_str.c 2_str_fn.h 2_str.fn.c

Length of a string.

The client code:

```
char e[10] = "india";
printf("length : %d\n", mystrlen(e));
```

The implementation:

version 1:

In this function, we access the elements of the array through an index i. We stop when we encounter the NULL character. The value of i is the length of the string. You may want to think why is it not that plus 1.

```
int mystrlen(char s[])
{
// version 1
	int i = 0;
	while(s[i] != '\0')
	{
		++i;
	}
	return i;
}
```

version 2: This uses the pointer to access the characters in the string.

As the expression of while, we may use any of these three expressions and all are equivalent. Would changing the pointer here affect the argument?

```
int mystrlen(char s[])
{
// version 2
	int i = 0;
//	while(*s++ != '\0')
//	while(*s++ != 0)
	while(*s++ )
	{
		++i;
	}
	return i;
}
```

version 3:

This is recursive. The length of a string whose first character is NULL is zero. Otherwise add 1 to the length of the string starting from next position.

You may want to think the difference between mystrlen(s++) and mystrlen(s + 1).

```
int mystrlen(char s[])
{
	// version 3:
	if(*s == '\0')
	{
		return 0;
	}
	else
	{
		//return 1 + mystrlen(s++); // infinite recursion
		return 1 + mystrlen(s + 1);
	}
}
```

copy string:

we are trying to copy the string f to string g. The string g should have enough space to hold the string f. Otherwise, it is an undefined behaviour.

The client code:

mystrcpy(g, f);

The implementation:

version 1:

Copy the characters from src to dst until the element in the src is a NULL character. We should not forget to copy the NULL character to dst after the loop. Otherwise all functions called on dst would be in trouble.

Can we say : dst[i++] = src[i]; instead of 2 lines in the loop?

```
void mystrcpy(char *dst, char *src)
{
	int i = 0;
```

```
	while(src[i] != '\0')
	{
		dst[i] = src[i];
		++i;
	}
	dst[i] = '\0';
}
```

version 2:

This is an amazing code. *src++ returns the char to which src was pointing before incrementing src. *dst++ returns the location to which dst was pointing before it is incremented. We copy the char returned by *src++ to *dst++. If the assigned char is not false(not NULL), we keep doing nothing in the loop.

Nothing to do after the loop as well.

```
void mystrcpy(char *dst, char *src)
{
	while(*dst++ = *src++)
		;
}
```

compare two strings:

The client code:

```
printf("compare : %d\n", mystrcmp("amar", "amar")); // expect to get 0
printf("compare : %d\n", mystrcmp("amar", "akbar")); // expect +ve value
printf("compare : %d\n", mystrcmp("amar", "anthony")); // expect -ve value
```

The implementation:

```
int	mystrcmp(char *s1, char *s2)
{
	while(*s1 && *s2 && *s1 == *s2)
	{
		++s1; ++s2;
	}
```

```
        return *s1 - *s2;
}
```

The concept is like this. If the two strings matched till the end including the NULL character – take the difference of these NULL characters – return 0 as the result.
Keep comparing the corresponding characters from left to right until a mismatch. Subtract the first character from the second. If the strings are in order based on ASCII, the difference will be negative – otherwise positive.

Can we say s1 – s2 instead of *s1 - *s2?
You may want to think.

Strings

Strings are arrays of char with a sentinel marking the end. We play with arrays either using indices or using pointers. So, arrays, strings and pointers go together.

Let us recollect that pointers and integers are basically different types. Only arithmetic allowed on pointers are the following.

```
// add an int to a pointer : pointer
//  subtract an int from a pointer : pointer
//  diff of two pointers : int
```

We can write the function to find the length of a string using pointer arithmetic in this way. Check the file 1_ex.c.

```
int mystrlen(char *y)
{
        char *z = y;
        while(*z)
        {
                ++z;
        }
        return z - y;
}
```

Make z point to the first character of the string pointed to by y.. Advance z until z points to the NULL character. Now, z – y is pointer arithmetic resulting in the number of components between the two pointer values. This is same as the length of the string.

There are a few functions which operate on strings. These are declared in header file string.h. We can also use man pages to find details of usage on a linux system. For ex, we can say

```
$ man strlen

STRLEN(3)          Linux Programmer's Manual          STRLEN(3)

NAME
    strlen - calculate the length of a string
```

SYNOPSIS

#include <string.h>

size_t strlen(const char *s);

DESCRIPTION

The strlen() function calculates the length of the string s,

excluding the terminating null byte ('\0').

RETURN VALUE

The strlen() function returns the number of bytes in the

string s.

string concatenation:

strcat function takes two arguments dst and src and appends src to dst.

char x[10] = "pesu"; // has 5 char

char y[5] = "univ"; // has 5 char

strcat(x, y); // x will not have 9 char; think why not 10?

printf("%s %s\n", x, y);// pesuuniv univ

If the destination does not have enough memory, the behaviour is undefined.

strcat(x, " India");

The program will crash if you are lucky!

Search for a char in a string:

The character we are searching for may or may not be found in the string.

We may decide to return the position if the element is found (0 to strlen of the str – 1) or -1 if the element is not found.

We will also try to find the leftmost occurrence.

Check the files : 2_ex_str.c 2_mystr.h 2_mystr.c.

The function signature :

// find the pos of ch in src;

// return the pos if found

```
//      -1 if not found
int find_left(const char *src, char ch);
```

Let us look at the implementation.

```
int find_left(const char *src, char ch)
{
        int pos = 0;
        while(src[pos] != '\0' && src[pos] != ch)
                ++pos;
        return src[pos] ? pos : -1;
}
```

The parameter src is supposed to point to a string which we do not plan to change. We can always treat a variable as a constant. It allows the compiler to optimize the code.

We start from the left end of the string – pos is 0 – and compare with the character ch until we find the character in the array or we come to the end of the string.

If the character is found, return pos else -1.

This is part of the client code.

```
char x[] = "kitkat";
// find the leftmost occurrence of say t
char ch;
ch = getchar();
int pos = find_left(x, ch);
if(pos != -1)
{
        printf("%c found at pos %d\n", ch, pos);
}
else
{
        printf("%c not found in %s\n", ch, x);
}
```

What if we want to find all the occurrences of the given character. Once we find the character, we have to search from the next position. We should keep repeating this until the end of the string is encountered.

```
int pos = -1;
int i;
while( (i = find_left(x + pos + 1, ch)) != -1)
{
        pos = pos + i + 1;
        printf("%d ", pos);
}
printf("\n");
```

The first time we call find_left, the string starts at x – offset is 0

If the string is found at pos i, the next search from position pos + i.

Please go through the code to understand the logic.

We should also be able to appreciate multifile development. Should the developer of find_left know how we are using the function in the client code?

Should the client know how the function is implemented in the implementation file?

String matching:

This is one of the biggest topics in algorithms. We search for strings in google. Now google is a verb! Searching for the presence of a string in another is string matching. There are a number of algorithms for string matching. We shall discuss the simplest algorithm in this course.

The problem statement:

We have a string : called text hereafter : with n characters.

We have a string : called pattern hereafter; with m characters.

Check if the pattern occurs in the text. If yes, return the position as an index else return -1.

Let us examine the files : 3_client.c 3_mystr.h 3_mystr.c

The function signature is as follows.

```
// pos in the text if pattern found in the text otherwise -1
int mymatch(char text[], int n, char pattern[], int m);
```

Let us examine the algorithm.

```
int mymatch(char text[], int n, char pattern[], int m)
{
        int i; int j;
        int res = -1;
        // outer loop : walk thro the text
        for(i = 0; res == -1 && i <= n - m; ++i)
        {
                // inner loop : walk thto the pattern
                for(j = 0; j < m && text[i + j] == pattern[j]; ++j)
                {

                }
                if(j == m)
                {
                        res = i;
                }
        }
        return res;
}
```

The outer loop indicates the position in the text from which we start matching the pattern. We start with i = 0. We increment i on an unsuccessful match. When the number of characters is less than the length of the pattern, we shall stop – i < (n – m) indicating the element is not found.

The inner loop compares the jth character of pattern with the ith character of text starting with j = 0. If the whole pattern is matched, then j == m. then we set a variable res to indicate the position of the match. If there is a mismatch, we start the iteration in the outerloop with the next value of i.

We exit the outer for loop when the string is matched – res will be no more -1 or when the matching fails – i < (n – m) becomes false. In any case, the variable res will have the right result.

Difference between pointer to a string literal and an array of char:

Let us examine the file 4_str.c.

```
char x[] = "pes";
char* y = "pes";
```

x is an array of 4 characters with ‘p’, ‘e’, ‘s’, ‘\0’. This is like a book owned by me.

y is a pointer to a string constant or literal. This is a like a library book – borrowed and not owned by me.

We can change the elements of x.

x[0] = ‘P’; // we can tear our book.

y0] = ‘p’ // undefined behaviour. Do not know what happens if you tear the pages of a borrowed book. I do not encourage to try it.

We can increment y. y is a pointer variable. Therefore it can be incremented.

We can not increment x as it is an array name. x is a constant.

Thats all about arrays, strings and pointers.