

C programming primer

This is some notes to explain how to write C program. It's intended for people that are not familiar with programming, so I'll start at the real beginning. I prefer to use example, in order to make things easier to understand, there will also be some exercise at the end of each sections to check if the concepts are properly understood. In order to stay concise I'll try not to exceed 30 pages including the exercises. Other example could be found in the algorithms primers notes.

An important motivation to start with C programming is that it allows to understand quite well the underlying mechanism that make the computer work. A lot of languages inherited for the C language, so that they turn them C almost as a basis of programming. That's the case notably with the C++ and the java programming language.

I | Hello world example (requirement)

As all programming course I'll begin with this simple but not very useful example :

```
// Printing Hello world example
#include <stdio.h>

int main(int argc, char **argv) {
    printf("Hello World\n");
    return 0;
}
```

Save the file as **1.c** and compile it using the command **gcc 1.c -o 1** to run it simply type **./1** in the shell prompt.

This first example will simply display the chain "Hello World", a chain of characters is a array of characters, each of them is coded in 8 bit (each bit is a 0 or a 1 for the computer), and one after there other they form a chain. The chain ends with the character `\0`, represented as 0 in memory it indicate the end of the chain. An array is a kind of data structure : each element are one next to the other.

On the first line of this example we have a comment, it's the C++ kind of comments : everything after a `/"` is the comment. On the second line we import a library through the include command, this is not really the C language but an extra that is processed before the real compilation begin, it's called the preprocessor. In this case it's useful in order to make the **printf** function available. This function allow to display chains of characters, and also to put the content of a variable in a given format.

The `{` indicate the beginning of a block, and `}` is it's the end. In each block we could use words that contain a values, they are called variables, they are local to the block. Most of the variable exists only in the block, in this case there is only both **argc** and **argv**. The first one is a integer number. The second is a location : a location to an array : a characters chains (a location is represented by a `*` when we define a variable. There are two `*` in our case, because it's a location to a location, we could also say a location to an array). Blocks could be preceded by a function declaration, in this case the function is called **main**. This function has two arguments **argc** and

argv, we call them also parameters, these parameters are only local to the block. Functions also return a value, in this case this value is an integer. The **main** function is some particular kind of function, it's the entry point of the program. Each program with a function called **main**, will run this function, and perform what is inside.

You should note that at the end of the **printf** there is a ";", that's because **printf** is an instruction. Instruction that affects variables by modifying them, or are call a function or **return** a value need to be ended by a ";". This is not the case for execution flow control instruction, neither for functions definition.

As announced **printf** allows also to display variables, this could be done, specifying for instance **%d** or **%s** in the first parameters (that is a chain of characters), depending if the variable is an integer or a chain of characters (we call them also strings). You could put this elements as much time as required, but each time you need to add a parameter to indicate the variable to substitute in the characters chain. Note also that chains of characters are delimited by double quotes "", this double quotes play a very important role, don't confuse them with the simple quote. They indicate that we are declaring an array of characters, and that this array ends with the '\0' character.

Exercise :

- Try to remove the \n in after the Hello World chain of characters ? What is the role of this group of characters ? NB : to display % we use %% and to display \ we use \\
- Display two lines one with Hello the other with World : do it using two time the function **printf**, and also using it only once.

We will now see how we could use the variable **argv** and **argc**.

```
/* Some more stuff on the Hello World example */

#include <stdio.h>

int main(int argc, char **argv)
{
    printf("Hello World was run with %d argument\n",argc-1);
    int i; // variable declaration
    for(i=0;i<argc;i++)
    {
        printf("%s:",argv[i]);
    }
    return 0;
}
```

run it with parameters like ./2 a b c, and you'll see :

```
Hello World with 0 argument
./2:a:b:c:
```

So we added lot of new features on this program, the first point is that we declare a variable "i", it's an **integer**, that will take the values 0,1,2 up to **argc** at each run of the block after the for loop declaration. **argc** is the number of arguments, and **argv** is the array that contains the location of the chain of characters to display. To get a element of an array we simply need to use [**i**] after the

name of the variable, where *i* is an integer contains the cell of the array to get. We could add more elements, to get elements in a multidimensional array, for instance :

```
printf("%c", argv[0][0]);
```

Will display the first character of the first array of characters : the 0'th arguments : the name of the function. We will soon see how to use this to display all characters of each arguments separately.

To display all arguments we use, a loop, that make "i" run over the values 0, 1, 2 up to **argc** (but not **argc**). If **argc** is equal to 0, the block will never be executed. The block is delimited by the { and } just after the the **for** loop declaration, will be executed for all values that the variable "i" will take. The for loop work in the following way :

- first **i=0** is the starting condition, **i** will be set to the value **0**.
- each time before the execution of the block the condition **i < argc** will be checked, so that **i** will never be bigger than **argc**.
- The last element, is a modification instruction, that will add one to **i**. This operation is called in our case an incrementation (more precisely, it's a post-incrementation, but it does play any role in this case). We could also write it **i=i+1** or **i+=1**

Before using **i**, we need to specify the type of the variable, we use the keyword **int**, that's mean that the variable is an integer, it could take any value between -2147483648 and 2147483647, it take 32 bits in memory. A bit is a 0 or a 1, by combining them it make 2^{32} numbers possibility. To have a fast idea how much it's we could take that 2^{10} is 1024, so 2^{30} is 1 billions, and 2^{32} is 4 times more, so 4 billions. One bit is their to encode the sign of the number, we call such numbers signed integers. On the opposite if there is no bit to indicate a sign we call them unsigned numbers.

Notice also some new kind of comment : the traditional C comment : they start with a **/*** and end with a ***/**.

Exercise :

- start the loop without displaying the program name
- put new line between each arguments that will be displayed
- do not print the function name
- avoid printing the ":" just after the last argument (tip : don't let "i" to be too big, and then print the last one)
- start with **i=argc-1** and end run still **i>=0**, replace the **++** by **--** to decrease by one at each turn (decrement). What is occurring ?

So let's turn to cutting the chain in smaller part : characters, and display the length of each of them.

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

```
int main(int argc, char **argv){
    printf("Hello World with %d argument\n", argc-1);
    int i;
    for(i=argc-1; i>0; i--){ // outer loop
        printf("\'%s\' :", argv[i]);
        int j;
        for(j=0; argv[i][j]!='\0'; j++)
```

```

        { // inner loop

        }
        printf(" has %d characters\n", j);
    }
    return 0;
}

```

So there is some extra features in the program, first there is an internal loop, inside the first one that is a loop over all arguments. The second loop, does nothing, except incrementing **j** (add one to **j** at each turn) until the end of the characters chain : until to find the '\0' character that indicate the end of the chain. Note that the character is inside simple quote ', characters chains and characters are not the same thing : the first are only one character. The second indicate the location of the first character, and add a '\0' character at the end of the chain of characters. The other point is that we access in fact in a 2 dimensional array, using the **[i][j]**, this mean the **j**'th character in the **i**'th argument. We first do the last argument, then the one before the last ..., and count characters using the variable **j** inside of the first loop. First we print the argument between quotes, and then without a new line between the number of characters that contain the argument.

So let's do a little bit more complicate things : other block of code : functions

As you can observe the following code is a little bit simpler because we separate the problem in two part : the main function call each time the length function to get the length of the characters chain. There is another advantage in defining function : we could use them more than in one location of the program.

Exercise :

- count the numbers of characters of all arguments (the sum). (Tip : use a counter outside of the most outer loop)
- display all characters for each arguments
- count the numbers of words. tip : put double quotes around the arguments when you execute the program : `./word_counter "this is a six words chain"`

```

#include <stdio.h>
int length(char *c)
{
    int j;
    for(j=0; c[j]!='\0'; j++);
    return j;
}

int main(int _c, char **_v){
    int i;
    char *str;
    for(i=_c-1; i>0; i--){
        printf("\n%s\n" : has %d characters \n", _v[i], length(_v[i]));
    } // it's not very well indented code, it was to save some place
    scanf("%a[ a-z]", &str); printf("%s", str);
    printf("%d character\n", length(str));
}

```

So we do different things in this code :

- we define a function : **length**. This function count the numbers of elements in a array of characters : like before incrementing a integer from 0 until the value of the character at the current position is '\0'.
- in the main function, the entrance point of the program, we first declare a reference to the starting of what will be the chain of character that we will read using **scanf**.
- we then use **scanf** function, and ask to scan a chain of characters that contains only whitespace and minuscule letters from a to z.
- then we display the chain we read before
- and display the numbers of characters, using the function we created previously

As the main function the function we declare take a position to the starting point of the chain, the * mean that it's a location, and not a value that it contains, two * : ** means that it's a location to a location, that's when we declare a variable. The & mean that we use the location of the element and the * mean that we use the value. So if we where using *c in the count function, it'd mean the first **character** of the characters string. By having the location of the first, we could go forward, until we reach the end of the chain that is the **character** '\0'.

As you can see this is quite more advanced, if you didn't get everything the following sections will make things more clear. The goal of this section was to show you an overview of the language, and make you take more attention on the important parts in the next sections.

Notice also that the name of the variable could be the one you desire, in this example we used another name for the number of arguments, and the variable to the array of characters arrays in the declaration of the main function.

Exercise :

- write a function that count how many time one specified characters is contained in a characters chain, the function should be defined like : int **count(char *str, char c)**. You could define a block that is executed only if you are in presence of the appropriate character :

```
if(str[j]==c)
{
    sum++;
    //you could also write sum=sum+1, it's the same
}
```

II] Execution flow control

Programs without control are quite boring, you have different way to control programs, but for all of them, they control a block of code that will or not, or several time be executed. So the different options are :

- **if / else if / else**
- **switch**
- **for** : initialization, condition, evolution
- **while** : condition
- **do-while**
- functions

The **if / else if / else** is very common, it use a condition, the minimal way of using it, is by using only the if, for instance :

```
if(a>5)
{
    printf("a is bigger than 5\n");
}
```

at the end we could add a block that will be executed in the opposite case

```
if(a%2==0){ // % is the modulo operation, it return the
            // remaining part of the euclidean division
    printf("a is an even number\n");
}
else{
    printf("a is an odd number");
}
```

NB : we suppose that a is an integer number, otherwise it doesn't make sens to get even or odd numbers. If we want we could add one more condition :

```
if(table[current]>0)
{
    printf("Element is bigger");
}
else if(table[current]<0)
{
    printf("Element is smaller");
}
else
{
    printf("Element was found");
}
```

in this example if the current element in the array table is bigger than 0 then it display that it's bigger, else if smaller, then it display it's smaller, and when it's neither smaller or bigger, then it displays that it's equal !

We could also use switch condition if we have alternative choices, depending on the value of an integer.

```
#include <stdio.h>

int main(int c, char **v)
{
    int a=0;
    scanf("%u", &a);
    printf("You have entered :%d\n", a);
}
```

```

switch(a) {
    case 0:
        printf("zero\n"); break;
    case 1:
        printf("one\n"); break;
    //...
    default:
        printf("%d is unknown\n",a);
}
}

```

So first we read the value of an integer, and store it at the position of variable called **a** in memory using the **&** symbol before the variable name in order to get its the location, that's the second argument of the **scanf** function. It allows the **scanf** function to know where to write the value that it read.

Then we use the variable **a** in the **switch** statement (language instruction) and execute what is contained in the **case :** subsection then we continue, down to the default instruction, that will be executed in any cases, except if we use a break instruction that'll allow to leave the switch block. Once one of the element was found to be true, then the other case block will also be executed.

Sometime we want to do things more than only once, in this case we use structure called loops, they execute everything that is in the block bellow. Generally using a condition to stop the loop. We call each turn of the loop an iteration. There is another way to leave it, by using the **break** instruction. There is also a way to avoid to execute the whole loop, this could be done by using the **continue** instruction that's allow to jump to try to execute the next block : the next iteration. An iteration is a run of the block.

As we had already seen for loops are quite simple to use :

```

#include <stdio.h>
int main(int c,char **v){
    int i;
    for(i=1;i<=5;i++)
    { // go from i=0 to i=5 by one steps
        printf("%f\n",1.0/i); // display the inverse of i
    }
}

```

Note the 1.0 that is required in order to make a division and get the result as a float number, that mean a non integer number, otherwise we could get meaningless result, because float numbers are coded in a different way than integer.

```

#include <stdio.h>
int main(int _c,char **_v){
// we don't use any argument, we avoid confusion with the
// program variable by putting _ before the variable. Because
// it's quite unusual to start variable with _
    int i;
    scanf("%u",&i);
    while(i>0){

```

```

        i/=2;
        printf("%c", i==0?'1':'0');
    }
}

```

The `?:` indicate on the left of the `?` a condition. Between the `?` and the `:` what is returned if the condition is true, and then after the `:` what is returned if the condition is false.

So let's have a look to the **do-while** structure :

```

#include <stdio.h>

int main(int argc, char **argv) {
    int i=5;
    do{
        printf("%d\n", i--);
    }while(i>0);
}

```

This way of writing allow to ensure the execution once before testing, it's much less frequent than the traditional for or while loops, but sometime it could be useful.

break/continue

Let's see how to use this two instructions that allow to control more precisely how loops are executed.

```

#include <stdio.h>

int main(int c, char **v) {
    int i=0;
    while(1) {
        printf("%d\n", i++);
        if(i==5)
        {
            break;
        }
    }
}

```

In this case the loop will never stop since 1 indicate true for the loop, but once `i` reach `5` the block containing a **break** instruction will be executed and will stop the run of the loop.

```

#include <stdio.h>

int main(int c, char **v) {
    int i=0;
    while(i<=10) {
        i++;
        if(i%2==0) {continue;}
    }
}

```

```

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

```

This last example on loops skip the execution of the content of the loop each time **i** is an even number. Later we will see example for which this way of doing is much more interesting.

Writing functions

Functions are piece of code that are reusable. They allow flexibility, by providing more modularity in programming. This is possible because they allow to organize the code, and divide the program. Calling function could be done within other function, or by the function itself. We call this way of doing : recursion, we give an example on it later. First we will write a function that display a help message :

```

#include <stdio.h>

void print_help(char **str){
    printf("Help message : \n\t%s\n", str);
}

int main(int argc, char **argv){
    print_help("help message displayer test");
}

```

So as you can see we display now the help message, with a little formatting around the character chain to display. The goal of this function is only to display the result, but to produce this, we need one argument, in this case this argument is the location of the first character of a character chain, some function could require more than only one argument. Despite the fact than by giving a location, we could alter the data that are at this location. As mathematic functions do, functions in C could also return a result, this result has a type. But it could also be **void** which mean that there is no return. Alternatively, we could specify one of the type of the C programming language, or a defined type (we will see later what it's). But first we will see how to calculation the Fibonacci (note that sometime this way of calculating are by far not the best, because it require too much calculation. Alternatively we could use dynamic programming, keeping result that where already calculated).

```

#include <stdio.h>

int fibo(int a,int b,int c){
    if(c<=2){
        return a+b;
    }else{
        return fibo(a+b,a,c-1);
    }
}

int main(int c,char **v){
    printf("%d", fibo(1,1,20));
}

```

This is what we call a recursive function, it's a function that call itself. But this process of

calling itself need to be stopped : this is done when a stopping condition is reached. This is to make the function stop to call itself. The **return** instruction, will return a value from the function, this allows to make further calculation, or further use of the function. The other point is that the parameters could be altered, after each call, this make the limiting condition to stop : the “c” variable diminish by one at each call, and the two previous arguments are summed.

III | Storing data

Storing data in memory could be done through different way, because there are different ways of organizing data in memory. There are basic type that could be composed together to form bigger structure, this could be done through two different ways :

- arrays : assembly of the same type of variable, one after the other
- structures : assembly of different kind of variable, that are named, it allow to transport data that go together, for instance an atom name, and 3 floating numbers to indicate it's spacial position.

Primitive type : this types are the smallest element we could manipulate (that's not really true because in fact we could manipulate each bit individually), there are two big types of element variables :

- variables containing directly data, they contain a value
- variable location of a variable, this location could change during the program execution.

We have already encountered the first one, there are three family, integers, characters, and floating numbers, on all of them we could apply the common mathematical operators.

Mathematical operators

The most basic one is the operator to change the value of a variable, this is done by the “setting” operator : = for instance :

```
int i, j, k;  
i=5;
```

assign the value **5** to the variable **i**. Do not confuse this operator with the comparison operator, has another action, it return a "true" value if both value are equal on both side of the operator, for instance **5 == i**, will return true if **i** is equal to **5**.

Then we could also perform standard operation on the variable, for instance :

```
j = i * i; // → 25  
k = i + i; // → 10  
k = j / k; //→ 2  
...
```

Note that when calculating with integers the division return the lowest integer just bellow the expected result. There is an additional operation : the remaining of an euclidean division :

```
k = j % k; // → 1 because at this point of the execution  
// k=2 and 25 is odd
```

There are also operations that act on the variables themselves : `+=`, `-=`, `*=`, `/=`, `%=` ; their action is respectively to add what is on the right, to remove it, to multiply by it, to divide by it, and to take the modulo of the right member. Modulo is the remaining part after an euclidean division : for instance :

```
k = 11;
k %= 3; // the new value of k will be 2
```

There is even more concise operator : self incrementation, and self decrementation operators, we already encountered them previously.

```
k=5;
k++; // the new value of k will be 6
k--; // k return to it's previous value
```

Something particular occurs with this variable, you could alter before or after using it's value, for instance :

```
k = 5;
j = k++; // k will be 6, but j will be 5
j = ++k; // j will be 7, and k as before will be incremented
```

Operators to manipulate the bits

Sometime it could be interesting to manipulate directly the bits this allow to modify precisely a number. This is interesting when you have to manipulate several two state system. For instance file permission are manipulated in this way : a read or not read permission this for each users. You also have them for write and execute, this will make a total of 9 bits (in fact there is 3 more bits indicated by `t`, `S` and `s`).

Bit shifting `<<` and `>>` and other binary operators

This operation is the equivalent of multiplying or dividing by a power of two, for instance :

```
int a = 1
a = a << 2; // a → 2
a = 16;
a = a >> 3; // a → 2
```

and `"&"` operator, take the bit that are enabled in both variables :

```
5&4 // → 4
5&1 // → 1
```

The or `"|"` operator, turn bit to true if one of the bit in both variable is true :

```
4|3 //→ 7
```

the binary representation of 4 is 100 and 3 is 011, so we will get 111 that is 7. There is also the **xor** operator that is symbolized by `^`, and allow the exclusive or operation to be performed. This mean

that exactly one bit has to be enabled in one of both number to stay on in the result.

$7^3 // \rightarrow 3$

Because we could easily reverse this operation, it could be used to build a parity bit, you xor two set of data, obtain a third. This third block allows you to rebuild the data if one of the two previous is lost. Try to understand how this is working.

Exercise : using right shift operator `>>` and the and operator `&` try to convert an integer to it's binary code, what happens when you try to convert negative values ?

Hints : start with the instruction `for(int i=0;i<=32;i++){` and shift with this operator `>> 1` if you want to shift once ...

Problem : display it in the right order, not first the bit that is the most at the right, but the left most bit first.

Test operators

Test are the basic element that control the duration of a loop, they allow to make program run depending on the value of variables. The test that could be done are the comparison :

- equal operator : `==`
- not equal : `!=`
- smaller or bigger `<`, `>`
- smaller or bigger or equal : `<=`, `>=`, not that they don't take more time
- inversion of the condition : `!`

Boolean operators

Sometime it's necessary to combine different test, this could be done in different ways, sometime we want to have both test true, sometime one or the other or both. This is possible through and `&&`, or `||`.

```
if(4%2==0 && 6%2==0){printf("6 and 4 are even\n");}  
if(4%2==0 || 6%2==0){printf("at least 4 or 6 is even, possibly  
both\n");}
```

There is an important point to note about this operator, if when executing `&&` the first element is false, then the second will not be executed. At the opposite if when we use this or operator `||`, meaning one or the other, the first will be executed, and if false the second will not.

This behavior contrast with the one of the `&` and `|` operator, that will cause the execution of all elements, and not the minimal required one's.

Operator priority

When you are using several operators there are priority rules that apply. The best way to deal with them is to use parentheses when doubt is possible, so the only case in which you don't have to

use them is when the conventional mathematical rules apply (the one about addition that have lower priority than multiplication). You could get more precise rules in books, but they could be confusing to other programmer, because the good rules are not very well known.

Basic variables

The main basic type of C are :

- **char** : character : 8 bit, they are used to represent characters but we could also use them to manipulate integers.
- **short** : 16 bit integer
- **int** : 32 bit integer
- **long** : 64 bit integer
- **float** : 32 float
- **double** : 64 float

Note that the size could vary depending of the architecture of the computer and the compiler, be also careful about the real representation in memory, it's not necessary the same on all computers !!! (the size is not given by the standard, but the order of the size of all elements are conserved : a short is always smaller or equal to a int, itself also smaller than a long.

Composed Variables

Putting data together is important because it allow to exchange data that make a whole. For instance if you have to manipulate atoms, it's a good idea to have a link to the kind of atoms that the program is manipulating. But also a table of all neighbors in order to be able to run over the whole molecule, visiting each atoms. Another example is the association between first name and last name, birthday... that'll define an individual person. The two reason to do so are :

- bring better organization in your program : putting things that go together together, will help to get the program more organized.
- efficiency, sometime you manipulate different kind of data all time together, so in some case it's more efficient to have them one next to the other (this is even more true for hard disk storage, in which avoiding going from one part of the memory to the other is very detrimental).

```
#include <stdio.h>
struct Complex {
    double x;
    double y;
};

print_complex(struct Complex c){
    printf("%8.3f+%8.3fi\n", c.x, c.y);
}

int main(int _c, char _v){
    struct Complex d;
    d.x=2.0; d.y=3.0;
    print_complex(d);
}
```

As you can see we define a new type, to use it we need to use **struct Complex**, we could manipulate it over the whole program, but we could make things more concise :

```
typedef struct {
    double x;
    double y;
} Complex;
```

This define **Complex** as a new type, we avoid to have to type **struct** at each time.

Note also that there is another instruction that allow to make variable that overlap, this could allow to have different view of a variable. To do this we use the **union** instruction : the different type of variable will overlap in this case. There is also a way to make multivalued variables : by using **enum**.

List of data

Sometime it could be interesting to organize your data one next to the other this could be done by creating arrays, we call them also tables. All data type including array themselves could be stored as arrays. To store a picture, for instance we could use an array of array to store all point that compose this picture. In the C language normally array have fixed size. There is no checking of going too far in the memory. So we could access to unwanted places, it's to the programmer to take care to avoid to access too far in memory. We have already used array to display text, because text is a array of **characters**.

```
#include <stdio.h>

int main(int c, char **v) {
    int LENGTH=10;
    int a[LENGTH];
    int i;
    for (i=0; i<LENGTH; i++) {
        a[i]=LENGTH-i;
    }
    for (i=0; i<LENGTH; i++) {
        printf("%i\n", a[i]);
    }
}
```

Note that arrays start at the position 0, and finish at **LENGTH-1** (in this case). So let's try a two dimensional array :

```
#include <stdio.h>

int main(int c, char **v) {
    int multtable[10][10];
    int i, j, x, y;
    for (i=0; i<10; i++) {
        for (j=0; j<10; j++) {
            x=i+1; y=j+1; multtable[i][j]=x*y;
        }
    }
}
```

```

    }
}
printf("%i\n", multtable[3][4]);
printf("%i\n", multtable[6][7]);
}

```

We could also transmit array to a function :

```

#include <stdio.h>

int f(int c[], int Lx, int x, int y) {
    return c[Lx*x+y];
}

int main(int c, char **v) {
    int data[9];
    data[0]=1;    data[1]=2;    data[2]=3;
    data[3]=2;    data[4]=4;    data[5]=6; // 3*2
    data[6]=3;    data[7]=6;    data[8]=9;
    printf("%d", f(data, 3, 1, 1));
}

```

This map two variable to the position in a two dimensional array

Exercise (hard) : the array you created is a symmetric array. As usage could be an issue, in the real life you need only to store half of the data, this could be done by comparing the index of the variable and exchanging them, to calculate the position in the 1D array.

Hint :
$$\sum_{i=1}^n i = \frac{n \cdot (n+1)}{2}$$

Locating variable in memory

The main problem with variable, is that when you pass them to a function, the function will have to copy the data. If the function modify this copy it will not affect the original version of the variable. And so passing arguments to a function will not allow you to modify them. The other option is returning a variable from the function, but sometime variable could be quite big, and this might not be an option. This is for instance the case in for sorting algorithms, some of them sort data without creating another array, avoiding to spend unnecessary memory. In fact with array we already used variable location. This location are commonly called pointer, a last name for them is reference, note that in C++ reference could be somethings other than pointer. Pointer correspond to a location in memory. They could be of different type, it's important, to note that the type of a pointer is important, because increasing the pointer of one element will bring you to the next element of the array (in memory : the element just next to the current). The type of the reference knows the exact size of the jump, an so allows you to get elements in the array increasing step by step the pointer value that correspond to a location in memory (also called address).

To define a pointer you simply need to write :

```
int *p;
```

p is now a pointer to an integer, up to now this pointer don't reference any integer, to make it reference to one, you need to extract the location in memory of a variable, it's important to understand that this is language manipulation, and not a transformation of a variable to a location, that would not be possible !!

```
int a=5;
p=&a;
```

So the & symbol in front of the variable transform the variable to it's reference, note that we don't put the * in front of p, it's only when you declare a pointer or extract it's value (we call it also to dereference it) that you need this *.

```
#include <stdio.h>
int main(int _c, char **_v) {
    int *p, a=5;
    p=&a;
    printf("At the location p=%p we have a variable that is named
a in the program and it's value is %d\n", p, *p);
}
```

So this program display the memory location of the variable and display it's location, note that the second argument of the printf function p could have been replaced by &a. We have already used this kind of variable in the example of the scanf function. That's because this function require to write modify the variable we transmit, so we have to transmit a place to write, instead of simply a value, the latest will be a copy, and changing it that will not affect the original variable.

Pointer and Arrays

Pointer are very closely related to array, in fact by manipulating arrays we are manipulating pointer :

```
#include <stdio.h>

int main(int c, char **v) {
    char *str="Hello World";
    int i; // in printf %s require a pointer to a character to work
    printf("%s\n", str);
    for(i=0; str[i]!='\0'; i++) {
        printf("%c", str[i]);
    }
    printf("\n");
    while(*(str++)!='\0') {
        printf("%c", *str);
    }
    printf("\n");
}
```

The output is :

```
Hello World
Hello World
```

ello World

So as you can see, in the first loop (**for**) we used the **str** variable that is a reference to a character as a array, then we displayed each characters one after one, to display the whole string (that's why we called it **str**). The second loop use the while instruction, this loop is a little bit more complicate to understand, so at each turn we increase the value of **str** by one making it reference to the next element of the array, the star convert this pointer to a variable to it's value in memory, then whether is value is equal or not to the string end characters, that is the null character : '\0'. Note also that '\0' is something really different from "\0", the first one is a variable of type char, the second is a pointer that refer to a variable that is '\0' followed by a second '\0'.

Exercise :

Why the last Hello World output start one step too late ?

How to avoid this by manipulating the **str** pointer before the loop ?

Use another kind of loop to avoid this, and place the condition after the first print.

Pointer and Functions

As I told before, pointer are a very important for using function, because they allow to alter what we give to the variable.

Usual function :

```
int f(int x){
    x=5;
    printf("%d", x);
    return x;
}

void turntofive(int *x){
    *x=5;
    printf("%d", *x);
}

main(int c_, char *_vv){
    int x;
    turntofive(&x);
    f(x);
}
```

Both of this function will display the new value of respectively **x** and ***x**. The difference is that in the first case the variable we pass to the function is not modified, it's a copy of it that is modified, at the opposite in the second case we pass the reference to the variable and so it's the variable itself that is modified.

More advanced Pointers

Pointers could be used in more advanced way, we could for instance use them in a **structure**, so to build integer that are connected to each other by a logical link. But first I need to introduce

you a new symbol :

The **p->** allows you to get an element of a structure that is pointed by **p** pointer this is very useful because it simplify the writing, you could also write : **(*p).x** but it's quite boring.

```
#include <stdio.h>

// Create a new type
typedef struct{
    int x;
    void *next;
}intNode;

// recursive function that called itself until finding a element
// on which pointer to the next element is equal to zero.

void run_chain(intNode *n){
    printf("%d\t", n->x);
    if(n->next==0){
        printf("\n");
        return;
    }else{
        run_chain(n->next);
    }
}

int main(int _c, char **_v){
    intNode a,b,c;
    a.x=1;b.x=2;c.x=3;
    a.next=&b;b.next=&c;
    c.next=0;
    run_chain(&a);
}
```

So this example is quite interesting and give a nice insight to what is possible through using pointer : we could organize data in memory as we wish, this I think is really that underlying basis on which object oriented programming languages rely.

But there is still something more tricky that allows very advanced things, for instance you want to sort data strings, this could be done by an usual sort function. But first remember what we all time put on the main function as second parameter : a pointer to a pointer : it's a array of string, or a 2D array. This structure is quite complex, but knowing the number of element we could extract the reference to the starting element simply doing **argv[0]** for the first one (the name we used to start the program). Then **argv[1]**, and so one until we reach **argc** that indicate us that there is no more elements to extract.

To sort an array of strings we need simply to manipulate on the pointer to their first character. So we need two things : a sorting algorithms, that don't know anything about the element we are sorting and a comparator function that will take an two arguments to make the comparison.

```

#include <stdio.h>

int comparator(char *stra, char *strb) {
    int i;
    for (i=0; stra[i]!='\0' && strb[i]!='\0'; i++) {
        if (stra[i]>strb[i]) {
            return 1;
        } else if (stra[i]<strb[i]) {
            return -1;
        }
    }
    if (stra[i]=='\0' && strb[i]=='\0') {return 0;}
    else if (strb[i]=='\0' ) {return 1;}
    else { return -1;}
}

void sort(char **strarr, int len) {
    int i, j; int res; int min;
    char *temporary;
    for (i=0; i<len; i++) {
        min=i;
        for (j=i; j<len; j++) {
            res=comparator(strarr[min], strarr[j]);
            if (res==1) { // only if bigger we change the max
                min=j;
            }
        }
        // exchange elements ( swapping )
        temporary=strarr[i];
        strarr[i]=strarr[min];
        strarr[min]=temporary;
    }
}

int main(int c, char **v) {
    int i;
    char *array[3];
    array[0]="Ax";
    array[1]="Ar";
    array[2]="Aa";
    sort(array, 3);
    for (i=0; i<3; i++) {
        printf("%s\n", array[i]);
    }
}

```

So this is a quite long code in comparison to the other I show previously, you could see the large advantage of using pointer : they allow an easier manipulation of data in memory, we don't move the strings we only move what refer to them. The algorithms we use take all element of after the other, and compare to the next, if it find bigger take it for min, and compare this max to the other, exchange so that at the end exchange the first, then the second with each time the smallest

one. Note that there are some much more clever algorithms to to sort element. This method has a number of operation that scale with the square of the length of the data. There are other algorithms that scale as product of length of the data with it's logarithm. The former is known as selection sort, the last one are **quickSort** and it's derivate. But we could still enhance what we have done we could use any kind of comparison function instead of only this one, for instance a comparison that don't care about lower or upper case characters.

I need space to store data : malloc

Sometime you don't know how much memory your program will need before it's execution. This is for instance the case when you read data from file, or receive data from the network (or other sockets), or simply keyboard input. You need to allocate memory to perform the data manipulation, this could be done with the instruction **malloc**. So let's see an example that require memory, the example I'll show you is a data structure that is called heap, the philosophy behind it is First In Last Out, that's also known as FILO, it's like a heap of plates.

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

typedef struct{
    void *prev;
    char *str;
}linkednode;

linkednode *mknode(int size){
    linkednode *new=malloc(sizeof(linkednode));
    new->prev=0;
    new->str=malloc(size);
    return new;
}

void push_elm(linkednode *new,linkednode *prev){
    printf("Create with prev=%p\n",prev);
    new->prev=prev;
}

linkednode* pop_elm(linkednode *curr){
    printf("%p:%s\n",curr->prev,curr->str);
    return curr->prev;
}

int actions(){
    int res;printf("1) push\n2) pop\n");
    scanf("%d",&res);return res;
}

int main(int _c,char **_v){
    linkednode *new,*prev; int act=3;
    prev=mknode(60); push_elm(prev,prev); // previous of prev is
prev
    while(act!=0){
```

```

    act=actions();
    switch(act) {
        case 1:
            new=mknnode(60);
            scanf("%s", new->str); push_elm(new, prev);
            prev=new; break;
        case 2:
            prev=pop_elm(prev);      break;
    }
}
}

```

In this example, we first created a new type that correspond to a structure : linkednode using the instruction typedef, this structure is able to reference previous elements. Each time we create a new linkednode we use the malloc instruction to allocate memory, then make a link in the new element to new previous one.

Exercise :

- modify this program in order to put together the instructions mknnode and pushelm, so that the new instruction return a node, that we could modify.
- Make more clever memory allocation by doing it before required, so instead allocating memory each time, allocate it for 10 nodes, then return elements references. The performance will be much better because the program will need less system calls, and because elements are one next to the other which is much more efficient.
- Make a structure that contain the height of the heap (the number of inserted elements) and the last inserted element.
- Use the free(*ptr) instruction to restore the memory you got from the allocation each time you take an element. ptr is the reference to the memory space you got from malloc.
- Difficult : store text as small part of text, so that when you modify text you don't have to modify a big array of text but only a small part.
- Difficult : write a word occurrence counter : store the word and a occurrence counter.

IV] Input/Output (IO)

In this section, I'm going to talk about two point, the first one is really about the Input/Output, that's mean, how to talk to the kernel to get access to the files that are stored on the hard disk. Then, I'll explain how to get formatted strings using variables variable, that's mean have the desired characters representation of variables, this could be stored in memory, or to any kind of flow (that's mean any kind of input output). That's why formatting text and input/output are very tightly connected field. Usually people want formatted output in order to let them human readable.

By formatting data we also ease the problem of communicating between different internal representation of variable : the relative position of bites in memory could vary depending on the computer architecture you are using.

Working with IO is about working with flows, flows have to be open before using them, and then

we could read or write on them, we could also have random access to them.

V] Math library

VI] Write you headers files, and your Makefile

<http://bits.stephan-brumme.com/>

http://en.wikipedia.org/wiki/Bit_manipulation

http://en.wikibooks.org/wiki/C_Programming/Pointers_and_arrays

<http://www.gamedev.net/reference/articles/article1697.asp>

Books

C programming course by Kernighan and Ritchie

Algorithms in C, Robert Sedgewick

Mastering Algorithms with C Oreilly Editions

Practical C Programming Oreilly Editions