**MODULAR PROGRAMMING**

FUNCTIONS

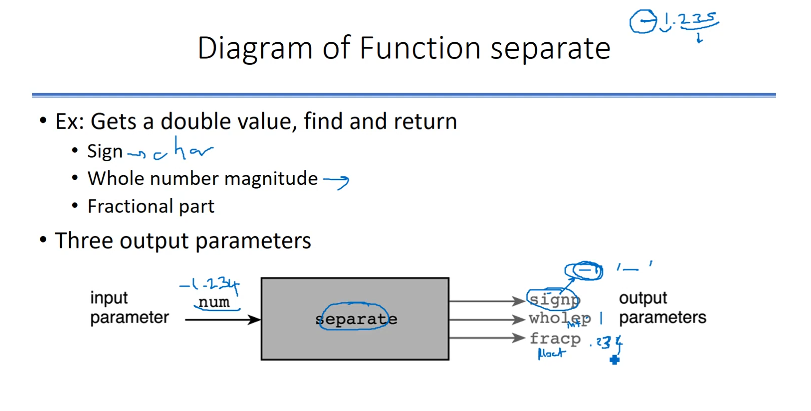
Functions : Components of a program

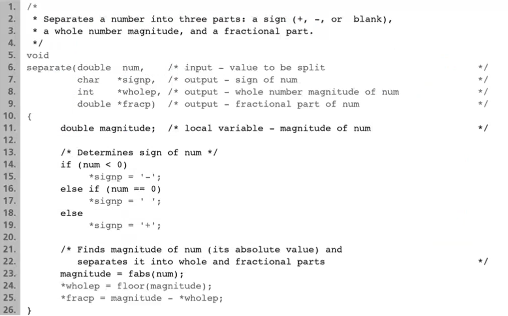
Each function has:

* Inputs
  + Parameters
  + Computes manipulate different data each time it is called
* Outputs
  + Returns a result with return statement
  + Output parameters to return multiple result

Function call steps:

* Allocate memory space for each formal parameter and local variables
  + Heap vs stack
  + Local variables are kept in stack, we are gonna learn heap later
* Store actual parameter value in the allocate space
* Execute function code
  + Manipulates the values of formal parameters

**POINTER** : Variable that keeps an address



separate(-1.234 , &variable1 , &variable2 , &variable3);

You should call this function like this. & sign means “I will retrieve one of the output of this function in this variable.”.

Use \* in front of the output parameters:

* declaration : char \*signp;
* assignment : \*signp = ‘-‘;

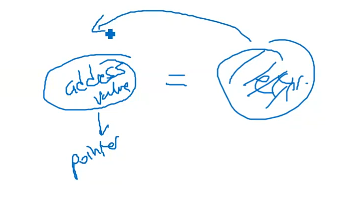
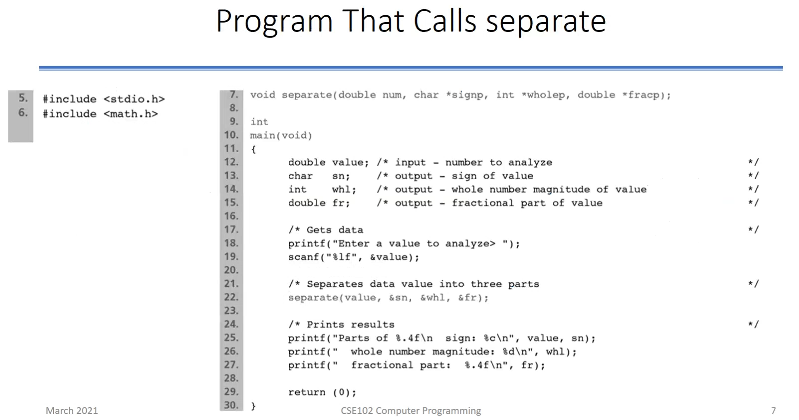
signp : pointer type, they are positive integers

* contains address of a char variable
* “p” is used because it is pointer

normally we do like : <type> <name>;

now we do like : <type \*> <name>; ----> <type \*> is pointer (like integer)

Assignment on pointers:

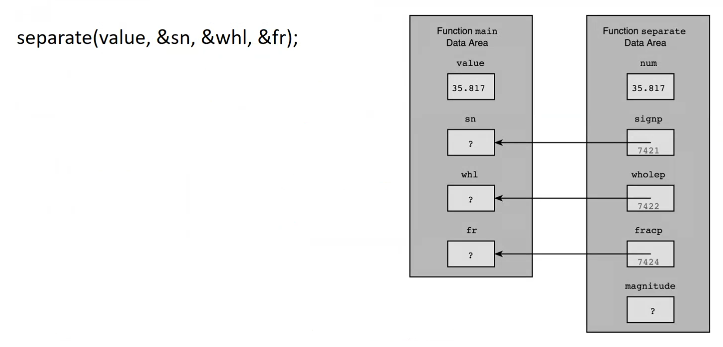


Three variables (sn, whl, fr) defined in main function

* values will be defined by function separate
* address of sn is stored in output parameter signp

Use & operator on the actual parameter

* separate(value, &sn, &whl, &fr);
* separate knows where sn is in the memory (like scanf)
* &sn is of type char-pointer



signp is the address of sn (7421).

wholep is the address of whl (7422).

fracp is the address of fr (7424).

When you say \*signp , you say “look at the value in signp, take that address inside signp, and go to that address wherever it is“.

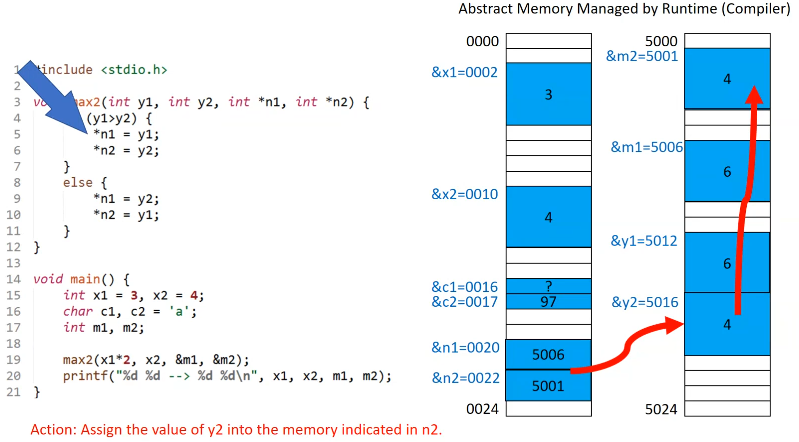
\*signp - - - > take the address 7421, and store your value in there

So when we say \*signp, we actually say sn.

\* : indirection operator. It says signp is pointer, it should have an address in it, hopefully it is address of a variable. Then I can go and check where that variable is with that address by putting a \* in pointer.

&sn - - - > & : address of sn

**All variables etc. are happening in stack in memory.**



Runtime system reserves the appropriate space from the memory. There is no way of knowing the allocation before runtime. Next run might result in a place for x1 that is completely different than this run. But you can get that address value if you want with & sign.

x2 doesn’t have to be next to the x1 in memory. Very high probability that they are gonna be next to each other.

max2 has 4 parameters: int (y1), int (y2), int pointer (n1)(pointing to an integer address), int pointer (n2)

<int \*><n1> - - - > I want to have an address (n1 : unsigned int), it should hold an integer value. int \* says “I am gonna use address to store and to read something. I want to store or read from that place only an integer value.”

So 0002 , 0010 , 5001 , 5006 are integer pointer (int \*).

We assumed that we don’t have very big memory. 4 hexadecimal character (2 bytes) is enough for integer pointers.

0000 - - - > hexadecimal. So we can understand that my memory amount is 2^16 bytes.

\*n1 - - - > go to n1, take the value in it (value in it is address because it is integer pointer), then 5006 is showing me m1. So we can say that \*n1 is actually same as m1.

\*n1 = y1; - - - > take 6 (value in y1), look what’s inside to n1, go to address that’s written in n1 and put 6 to that address.

When you typecasting a variable to another type, for example:

double x;

int y1 = 6;

x = (float)y1;

Typecasting doesn’t do anything with y1, so y1 doesn’t change. (float)y1 returns 6.0 and assigns it to x.

Operations that we can do on addresses :

* Assignment
* Increment, decrement (meanings are not always valid)
* Addition, subtraction (meanings are not always valid)

Program Style

* Use functions to take input parameters and return a value
  + Easier to understand and maintain
    - No indirect reference
    - No address operator
    - Return value is assigned to a variable at caller
* Math function are of this type

**SUBPROGRAMS (PROCEDURES)** : Not necessarily returning a value with return statement, but they may return values indirectly with pointers. Top to bottom, complexity increases.

|  |  |  |  |
| --- | --- | --- | --- |
| Different Kinds of Function Subprograms | | | |
| PURPOSE | FUNCTION TYPE | PARAMETERS | TO RETURN RESULT |
| To compute or obtain as input a single numeric or character value. | Same as type of value to be computed or obtained. | Input parameters hold copies of data provided by calling function. | Function code includes a return statement with an expression whose value is the result. |
| To produce printed output containing values of numeric or character arguments. | void | Input parameters hold copies of data provided by calling function. | No result is returned. |
| To compute multiple numeric or character results. | void | Input parameters hold copies of data provided by calling function.  f(int y1, …);  Output parameters are pointers to actual arguments.  f(…, int \*n1); | Results are stored in the calling function’s data area by indirect assignment through output parameters No return statement is required. |
| To modify argument values. | void | Input/output parameters are pointers to actual arguments. Input data is accessed by indirect reference through parameters.  f(int \*n1); | Results are stored in the calling function’s data area by indirect assignment through output parameters. No return statement is required. |

You should use simplest one suitable to what you want to do.

1 : input values -> return a value 3 : input values -> return multiple values in pointers

2 : input values -> output is void (io stuff) 4 : input and output on the same variables

At part 3, input values are not modified by the function. There may even not be any input values, just pointers as parameters of the function. Places of these pointers point to are not initialized. You don’t pass anything to that type of function.

At part 4, input values are modified by the function. Parameters must be initialized.

Scope of Names

Region of program that the name is visible.

Scope of

* constant macros
  + From definition do the end of source file
* function names
  + From function prototype to the end of source file
* variable
  + From declaration to closing brace
  + We can expand that with pointers.

What if an identifier is defined before?

int x; - - - - - > Global variable

int y; - - - - - > Global variable

int f1(int x){

int y; - - - - > When I come here, I will declare my own y. I don’t consider global one. This is **SHADOWING**.

…

}

int f2(int x){

int y;

…

}

We shouldn’t use global variables.

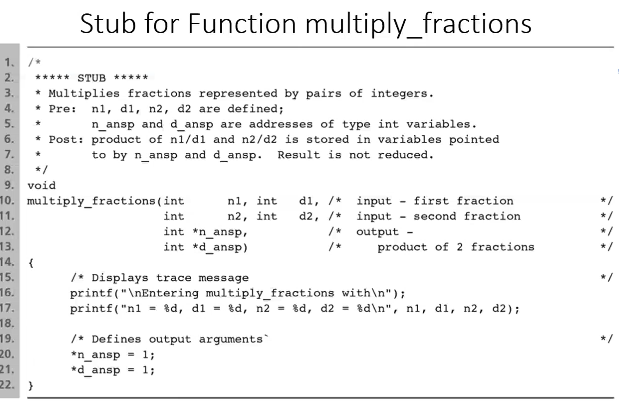
You shouldn’t have 100 lines of code in 1 function because it gets harder to follow where the variable is.

Formal Output Parameters as Actual Arguments

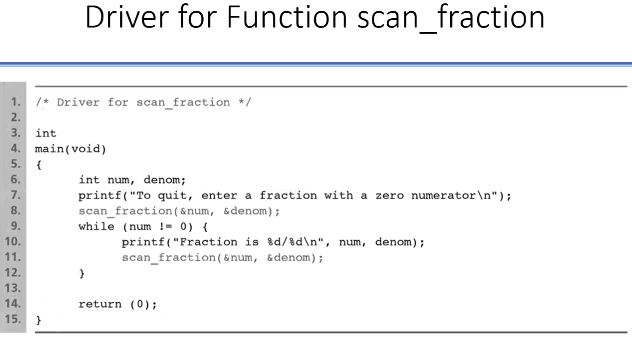
* Passing output parameters to other functions.
  + Ex : Reading values into output parameters

Testing

* Top-down testing
  + Test general flow of control
* Stubs
  + Used instead of functions not yet written
  + Team work
    - Enables testing and debugging
  + Displays an identification message
  + Assign values to output parameters



* Bottom-up Testing
  + First test individual functions
    - **Unit test**
  + Test entire system later
    - System integration test
* Unit Test
  + Preliminary test of a function separate from the whole program
  + Using driver program
    - Driver gives values to input parameters
    - Calls the function
    - Display and check function results



Debugging

* Good documentation is essential
  + Function’s purpose, parameters, local variables
* Debug each function as you write them
* Create a trace
  + Display the function name as you enter it
  + Display and verify the input parameters as you enter a function
  + Display and verify return values after function returns
  + After it works fine do not erase display statements, comment them out. You may need them later
* Use debugger (for example **dbx**)
  + First execute a function as a single statement
  + If the result is incorrect step in its statements

You can put printfs to critical places to see how your variables change.

**SIMPLE DATA TYPES**

Standard data types

* char, int, double, etc.
* logical values

Define new data types

* enumerated types

Passing functions as a parameter to subprogram

For data, we have 3 major parts that a type should have:

* set of values that data can have
* operations defined on this type
* representation (hidden in the compiler, abstracted concept, not all programmers have to know it)
* For example for integer:
  + we have values from {min, …, -1, 0, 1, 2, …, max}
  + we can do operations like +, -, \*, /, % …
* For float:
  + we have values with decimal points
  + we can do operations +, -, \*, / …
* Logical values are also a type that contains only 2 values that can be represented with 1 bit which are - - -> 0, 1. Machines don’t keep bits separate, they group them. They manipulate bits within groups. Smallest group usually is a byte for most of the architectures. So if I put logical values in a variable, I only use 1/8 piece. Actually generally C makes it simple and uses integer (efficiency issue) for logical values because we don’t use logical values in our programs so many times.
  + operations : &&, ||, !

You can do this operation with characters - - - > ‘a’ + 1.

All datas have bit representations.

You can imagine infinitely many different data types that you can create.

If we want to for example represent students in numbers, represent their grades, amount of attendance… We use user defined types for this kind of things.

Representation of Numeric Types

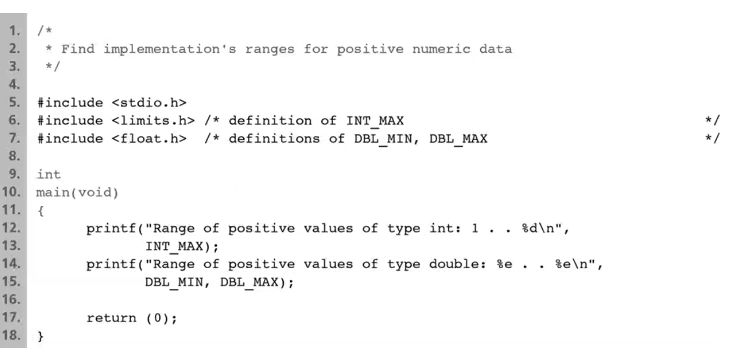
integers

* faster
* less space
* precise

double

* larger interval (large and small values)

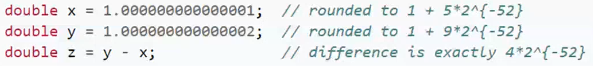




Numerical Inaccuracies

Errors in representing real numbers using double

* representational error
  + round-off error
  + magnified through repeated calculation
  + use as a loop control
* cancelation error
  + manipulating very small and very large real numbers
* arithmetic underflow
  + too small to represent
* arithmetic overflow
  + too large to represent



Type Conversion

* Automatic conversion
  + arithmetic operations (according to more precise one, float)
  + assignment
  + parameter passing
* Explicit conversion
  + casting
    - frac = n1 / d1;
    - frac = (double) (n1 / d1);
    - frac = (double) n1 / d1;

Representation and Conversion of char

ASCII

* numeric code (32 to 126 printable and others control char)

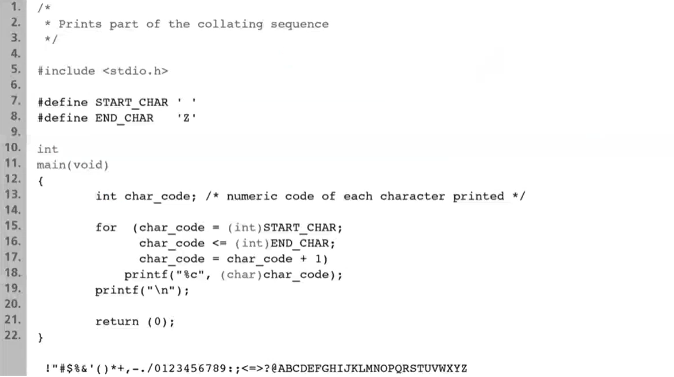
constant: ‘a’

You can do comparison: ==, !=, <, >, <=, >=

* if (letter > ‘A’)

Relation with integer

* compare
* convert



Enumerated Types

Defines new data type

typedef enum

{sunday, monday, tuesday, wednesday, - - - > within curly braces there are identifiers

thursday, friday, saturday}

day\_t;

enum : short for enumerated

day\_t is a new type. It has 7 possible values.

sunday is an enumeration constant represented by 0. Similarly monday = 1, tuesday = 2, etc.

General syntax :

* typedef enum

{identifier\_list}

enum\_type;

* enum\_type variable\_name;

typedef …… <typename>; - - - - - > we can use typename we defined now as if we are using what is writing in dotted part (for example instead of int).

We said type has 2 things:

* values : we have sunday, monday, tuesday, wednesday, thursday, friday, saturday (like 1, 2, 3, 4, etc. in integer). These are the names that I gave to my values. Reserved names cannot be used for these names. Also names that you have already defined cannot be used for these names.
* operations : we can do assignment, comparison

We can define a new variable: day\_t today;

today is of type day\_t

* manipulated as other integers

today = sunday;

You can do things like this : today < monday, today == monday

You can do : switch (today) and case monday , etc.

How C represents this?

day\_t is something like an integer. Then C assigned sunday, monday… to integer values that are known from 0 to 6 if you didn’t do anything else. For C, sunday and 0 mean the same thing.

You can do tuesday + 2 - - - > this is an integer 4

You can cast this back to day\_t like this - - - > (day\_t)(tuesday + 2) - - - > and this means thursday now.

You can do “(sunday - wednesday) + monday” and cast this again to day\_t. If result is out of range, not between 0 and 6, C does something completely wrong. So it is your responsibility.

What if you assigned your variable to something not in the names?

day\_t today;

day\_t = cumartesi; -> compiler is gonna say what’s cumartesi. It is not defined. You can’t do this.

What if you assigned your variable to integer not in range?

today = 10;

You shouldn’t do this. Because you know today should have sunday, monday, …. You shouldn’t give number.

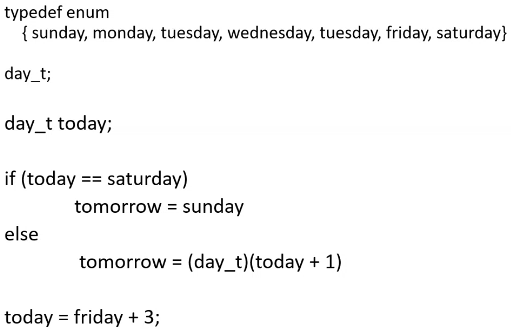
But if you do this :

today = 0;

C is gonna accept this. Because 0 and sunday are same for C. It knows what 0 means for your user definition type. Syntactically this is okay but semantically there is still a problem. If you were supposed to use those values as names, why don’t you do that way? Your reason that you named those sunday, monday,… you want to use that sunday in your semantic.

today variable will be treated like an integer by C even though it is enumerated type. But today=0 is against the semantic and it is not good. So we should use name values.

You can do today=0 kind of things but you have to know what you are doing. You have to make sure what you are using fits with your semantic.

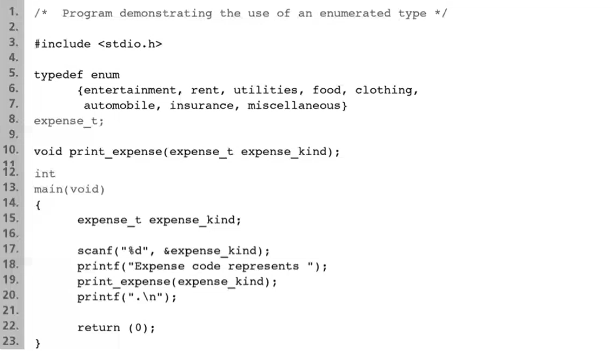


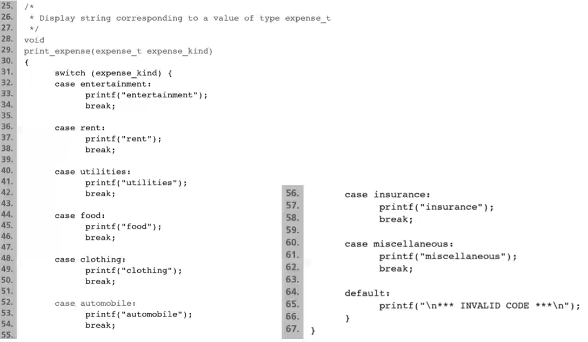
You can do ++today. But if today is saturday, C says this is 7 I don’t care. But 7 is not reachable by you with your names.

You can solve it with basic if block. If today is saturday today = sunday else ++today.

You can start sunday from a different number so other names also will be incremented by 1.

You can also say sunday is 5, monday is 10, tuesday is 15, wednesday is 52, tuesday is 88, etc.

You cannot assign same number to same name.

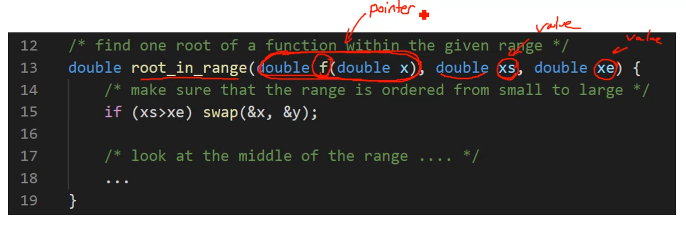


These enum types can be easily updated with new requirements.

If you don’t want to use typedef, you can do like:

enum {red, green, blue} x; - - - > one time thing, you can’t use it again

So better one is using with typedef.



double my\_func(double zz); ---> lets say this is defined. “my\_func” is sending me a pointer to the function

When we call root\_in\_range function, we call it like:

root\_in\_range(my\_func, -1, 2);

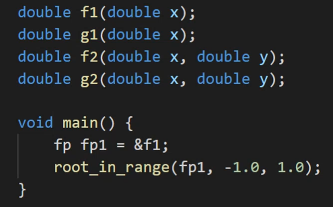
So when we use f(…) in our root\_in\_range function that points to my\_func.

**Function names represent the address (initial address) of that function just like arrays.**

typedef double (\*fp)(double); - - - > fp : name, function pointer

getting a double value and returning a double value

double root\_in\_range(fp f, double xs, double xe){



fp1 : pointer

**DECLARING ARRAYS**

* *Simple memory types : single memory cell (int, double… / only one value)*
* Group of related data items : adjacent memory cells
  + Array : uses consecutive area in memory
    - Can be referenced as a group
  + Array elements : each data item
    - Can be accessed individually

Example: double x[8]; (declaration of an array with 8 elements of type double)

* Name of the array : “x”
* There are 8 elements (memory cells)
* Each element is double
* x[5] is a subscripted variable
* 5 is an array subscript
  + Any integer from 0 to 7

Index that you write in [] can be a variable or expression evaluates to an integer.

Actually C doesn’t check if your index is between 0 and 7 (it is negative or bigger than 7). It’s your responsibility.

What happens if you write x[18] ?

Your array was defined for 8 entry only.

|  |
| --- |
| … |
| 8 BYTES |
| 8 BYTES |
| 8 BYTES |
| 8 BYTES |
| 8 BYTES |
| 8 BYTES |
| 8 BYTES |
| 8 BYTES |
| … |

C gave you 64 bytes.

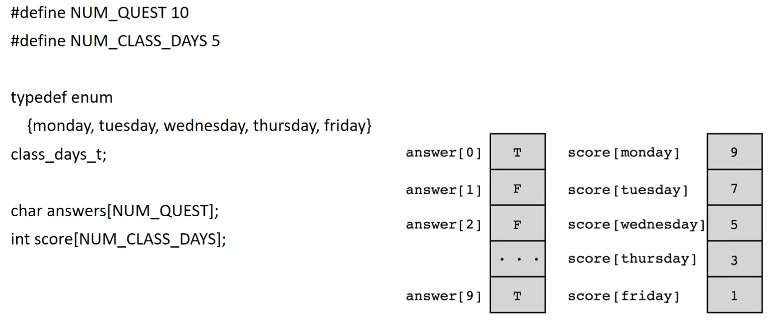
18th entry is somewhere down in the memory. It may cause problems at runtime. Maybe you are referring to place that you have no right to refer to. You don’t know if that memory place is actually belonging to your program or not. If it belongs to your program, you may still get a value. If it doesn’t belong to your program, than this will give you a segmentation error or some kind of memory error at runtime.

C reaches to index that you want by this formula:

&x + index\*8; - - - > that’s way index starts from 0 (8 is byte)

You cannot increase or decrease your array size at runtime. You should change before the compile if you need more or less area.

* Parallel arrays
  + id[i] and gpa[i] are related
    - First student’s ID is in id[0]
    - First student’s GPA is in gpa[0]



An array can be initialized in declaration:

* int primes[5] = {2, 3, 5, 7, 11};
* int primes[5] = {2, 3, 5, 7};
* int primes[] = {2, 3, 5, 7, 11};
* int primes[5] = {2, 3, 5, 7, 11, 13}; XXXNOT ALLOWEDXXX

Syntax:

* element\_type array\_name[size];
* element\_type array\_name[size] = {initialization list};

size has to be known at compile time so it has to be constant number or defined constant that can be calculated after the preprocessing.

Array Subscripts

subscript = index

Subscript specifies array elements

* Any expression that evaluates to int
* Must be between 0 to size-1 (This is not checked at compile time, it is not checked in runtime either unless it causes a problem. If it causes a problem, then OS or runtime engine will catch it and it will give an error. For example if you are out at the memory addresses that belong to your program, segmentation error will be given. Sometimes it doesn’t catch the error because it is not critical but your program may do very weird things. )

Syntax:

* array\_name[subscript]

Example:

i=5;

x[i-2] = x[i] - 2;

x[2\*i] = x[i--];

i = (int)x[(int)x[3+1]];

Example: Sum of scores

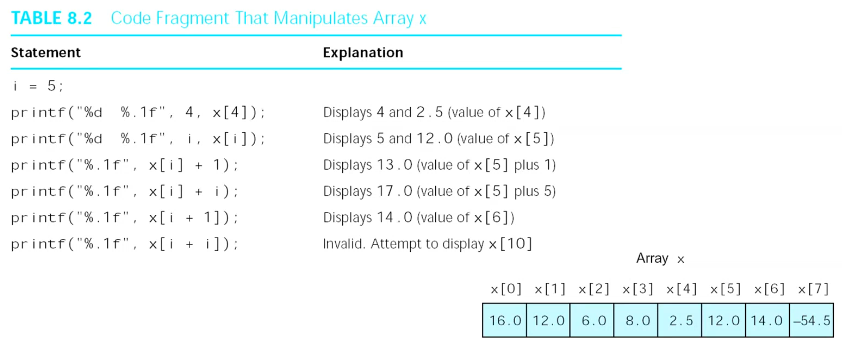
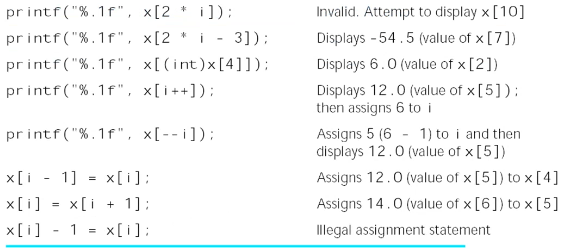
sum\_score = 0;

for (today = monday; today <= friday; ++today)

scanf(“%d”, &score[today]);

for (today = monday; today <= friday; ++today)

sum\_score += score[today];

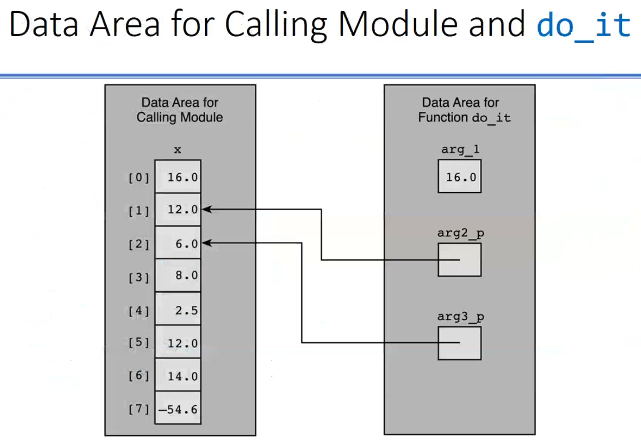


Array elements can be arguments to functions as other variables:

void do\_it(double arg1, double \*arg2\_p, double \*arg3\_p);

do\_it(p, &r, &s);

do\_it(x[0], &x[1], &x[2]);



You can send full array by sending address of a[0]. C thinks it is same as a. Variable a is the pointer. It points to the first address (a[0]). So you work on the original array.

int sum(int \*arr){ *//This is also same with (int arr[])*

*arr[0] is the first entry whatever arr is pointing at.*

…

}

But in the sum function I don’t know how many entries that array has. So you can add new parameter (int n) for the entry number or you can do it like this (100 is some number):

int sum(int arr[100]){ *//You can write 5+3 or MAX\_S (defined in preprocessor) instead of 100*

*but you can’t write n\*3 because it is only available at runtime*

…

}

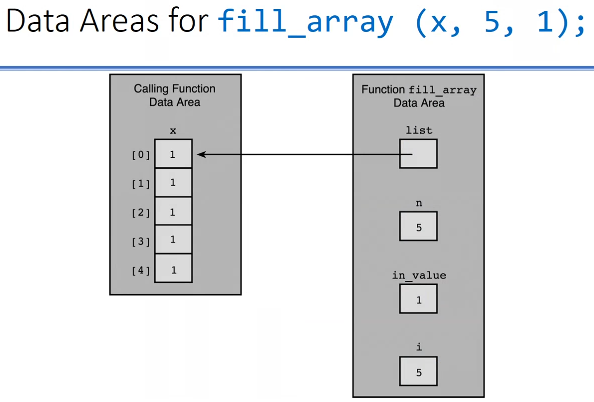
void fill\_array(int list[], int n, int in\_value);

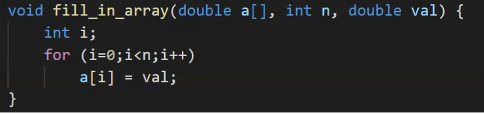
fill\_array(x, 5, 1); OR fill\_array(&x[0], 5, 1);

(int n) - - - > number of entries

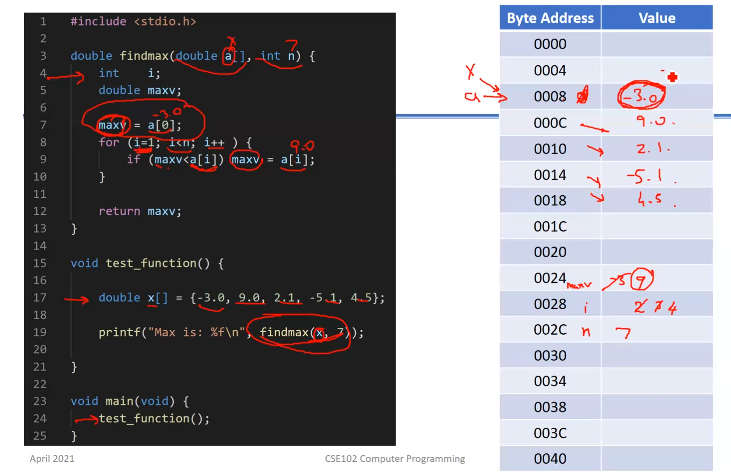
Because x itself is pointer to the beginning of the array.

x === &x[0]





If you pass &x[2], than array a will start from second index of the original array. So 2nd index of x will be 0th index of a.



7 must be 5 to not get out of range.

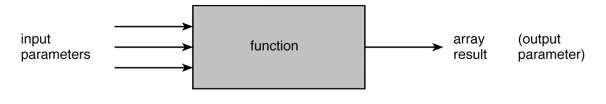
a has the value 0008.

You can use \*list instead of list[] in a formal parameter list.

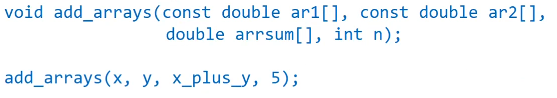
* Pass an array as a argument
* int list[]; - - - > means parameter is an array, more informative
* int \*list; - - - > is correct as well
  + Array argument : passing the address of the first element
  + But, it does not show that the argument is an array!
  + You should remember that it is array, not output parameter
* What if the array is only input parameter just as in findmax function?
  + You may don’t want to modify your array.
  + Use const qualifier (const double a[]) for parameter. You can also use “const double \*a”.
  + Const qualifier says don’t touch the entry, just read it, don’t write to it
  + You can not modify const parameters, otherwise the compiler will mark as an error
* Example : finding max element in an array
  + You do not need to modify array elements
  + It is safer to use const qualifier

Returning Array Result

* You can not return an array as a function’s return value
* You should define it as an output parameter



Example : Adding two arrays



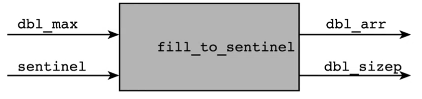
In this case ar1 and ar2 are input arrays, arrsum is output array.

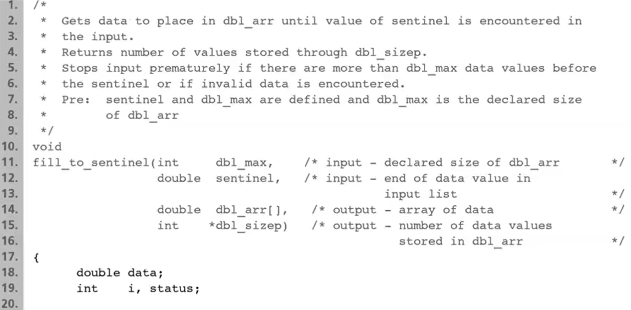
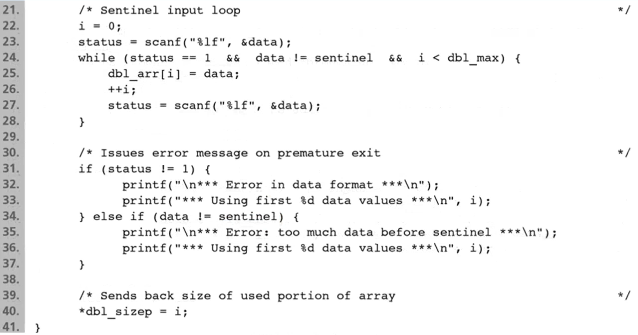
We need to declare arrays in compile time and we need to give them a fixed size.

Partially Filled Arrays

* Array is not completely used
  + Some part is reserved for later use
  + Need to reuse the same array for other purpose later
* Need to remember the actual number of elements in the array
  + Declared size should be larger than actual size

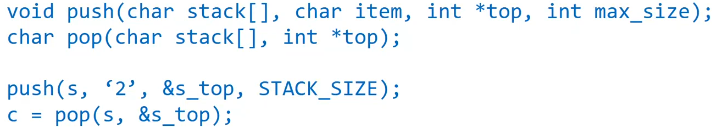
Example : fill an array until a sentinel value is entered





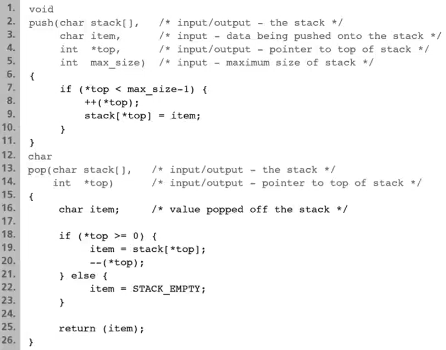
**STACKS (A DATA TYPE)**

* Only top element can be accessed
* Operations:
  + Push (Add on top)
  + Pop (Read top entry and remove it)
* First In Last Out / Last In First Out
* Array as a stack
* What should be the parameters to push and pop



char item - - -> item I want to put

int \*top - - - > like counter, go in array one by one and showing last item



We don’t change size or we don’t omit anything from the array but we are just decreasing the marker that shows where we are in the list in pop.

**Searching an Array**

* Two important problems in processing arrays
  + Searching: Locating a particular value
  + Sorting: Ordering the elements
* Searching: Linear search
  + Test each elements in the array one by one
  + Until the target is found or the array is ended

**Multidimensional Arrays**

* Array with two or more dimensions
  + Tables of data
  + Matrices

array[column][row] - - - > generally

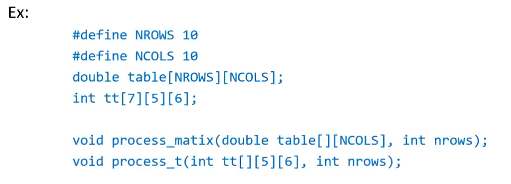
Syntax:

* Decleration:

element-type name[size1][size2]…[sizen];

* Parameter to a function:

element-type name[][size2]…[sizen]



We are sending a[] with one dimensional array, but now we have to define further sizes.

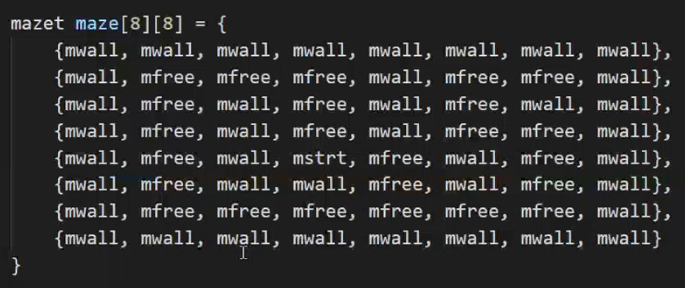
You can do “process\_matix(table, 10)” but you cannot do this if third line would be like “double table[10][2];”.



|  |  |  |
| --- | --- | --- |
| 0 | 1 | 2 |
| 3 | 4 | 5 |
| 6 | 7 | 8 |

[3][3][2] = { { {…, …} , {…, …} , {…, …} } , { {…, …} , {…, …} , {…, …} } , { {…, …} , {…, …} , {…, …} } }

[3][4] = { {…, …, …, …} , {…, …, …, …} , {…, …, …, …} }



PASSING 2D ARRAY TO FUNCTION (arr[3][3])

1. void f(int arr[][3]) - - - > just arr is used as 2 dimensional array in the function
2. void f(int (\*p)[3]) - - - > p is pointer to array, just p is used as 2 dimensional array in the function

**STRINGS**

* String : group of characters
  + Implemented as arrays of char
  + Essential for several applications manipulating textual data
    - Word processing
    - Databases
    - Scientific computing (Ex: DNA sequence, chemical compounds)
  + Already used string constants
    - printf and scanf format strings

\*\*\*Constant means : the value of the thing that we are interested in (whatever type it is) is known at the compile time. I know what that value is, I know how to put that in memory.

0 integer value will be put after the last character. It indicates end of the string.

It is 0 because 0 has no printable equivalent as character in ASCII table. It is not used.

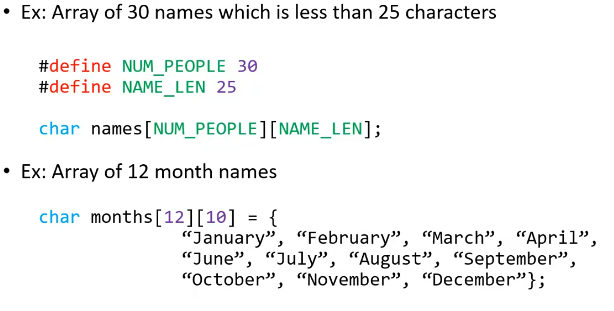
* Declaration: same as declaring array of chars

char string\_var[30];

* The variable string\_var can hold a string of 0 to 29 characters
  + NOT 30!
  + Use of null character at the end : ‘\0’ (this is int 0 as a character)
* String variables can be initialized
  + char string\_var[30] = “initial value”; (There is ‘\0’ at the end)
  + Upper one actually equal to :
    - char string\_var[30] = {‘i’,’n’,’i’,…,’u’,’e’,’\0’};
  + When C sees “ “, it interpretes it like string and put \0 to the end.
  + char str[] = “initial value”; (There is ‘\0’ at the end so 14 character places will be allocated in the memory.)
* What is the size of str? : The part of array after null character is ignored.
* What if you write [10] instead of [30] in the first part? : Either the compiler gives an error, or compiler goes for first 9 entries + ‘\0’.

Arrays of Strings

* An array of strings: a two-dimensional array of chars



First dimension : { } , Second dimension : “ ”

char name[] = “Ali”;

|  |  |  |  |
| --- | --- | --- | --- |
| A | l | i | \0 |

THIS IS STRING.

names[3] = ‘s’;

|  |  |  |  |
| --- | --- | --- | --- |
| A | l | i | s |

THIS IS NOW AN ARRAY OF CHARACTERS.

Input/Output of Strings

* Place holder: “%s” - - - > Take your input as a special character array until \0
* printf prints characters until null character
* scanf can used to input strings

scanf(“%s”, string\_var); - - > string\_var is already an address

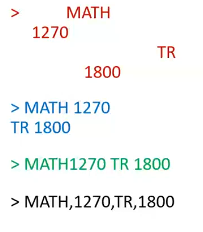
* string\_var is an array
* scanf
  + skips leading whitespace (space, tab,…) characters
  + copies subsequent characters in memory cells
  + copying stops when a whitespace character is seen, or carriage return (enter)
  + places a null character at the end of string

%7s - - -> Allocate 7 spaces at least and align string to right

%-7s - - - > Allocate 7 spaces at least and align string to left

scanf(“%s%d%s%d”, …); - - - > You can use like this because when user space out, scanf takes what is entered. Also %s starts reading when it skips spaces so you can print s after d.

scanf(“%s%d%s%d”, … );

First two are same.

For the third one, scanf will read 3 values.

For the last one, scanf will read 1 value. An error may occur if your string variable has less space (10) than “MATH,1270,TR,1800”. So scanf will take “MATH,1270,” in your string so your string became char array. “,TR,1800” goes after your array in the memory. So this is ERROR.

We said for type we need to have 2 things (and representation). For strings:

* Values: any combination of characters ending with null character.
* Operations: comparison, copy one string into another, append, reversed, capitalize, lower, upper, etc.

We don’t use == or + , we use function equivalent.

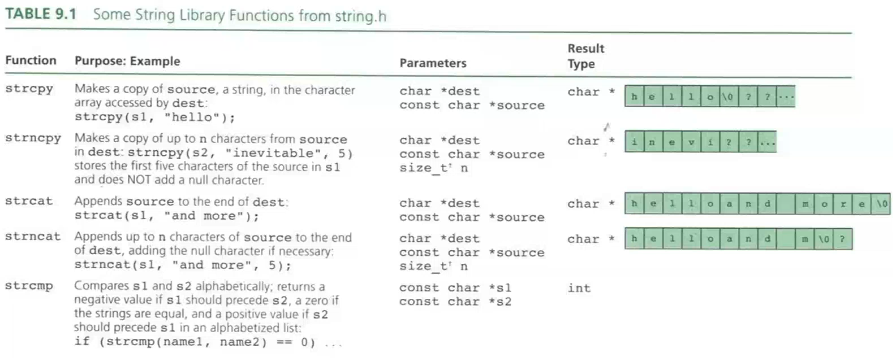
We usually don’t assign a string to another string, we use copying a string to another string.

\*\*DON’T FORGET YOU CAN COMPARE CHARACTERS.

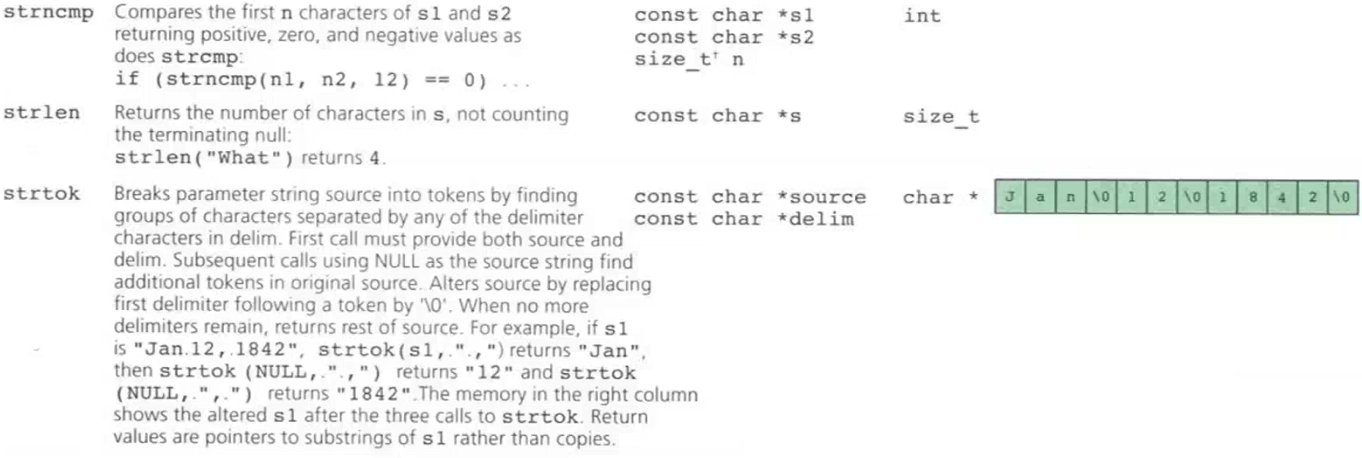
enum and logical types are represented as integers.

ch + ‘A’ - ‘a’ ---> Capitalize your letter.

String Library Functions: Assignment

* Assignment operator: =
  + Used for assigning simple types
  + Can not be used for arrays and strings
    - Other than in declaration with initialization
* C provides library function for assignment
  + Library in string.h
  + Includes several operations
    - Substring functions, concatenation, comparison, length, etc…

strncpy, s2 may not be array because it may not have \0 in the end.

size\_t is actually an integer, but it indicates that I am interested in how many bytes of things I have in the memory. So it is integer but the meaning is in bytes what do I have.

strlen goes until it sees null character.

* strcpy

strcpy(str, “test value”); - - -> str should be declared first with size at least 11 for \0

* Be careful about overflow!

strcpy(str, “A very long string test value”); - - - > additional memory doesn’t belong to str. overflowed memory may belong to other variable so you may messing the content of that variable, or if you are lucky it doesn’t belong to any variable. Also maybe that additional memory doesn’t belong to your program at all so this will cause a segmentation error. First one is more problem because it doesn’t give an error, you can’t see what’s happening.

* strncpy : copies first n characters

strncpy(str, “test value”, 20); - - -> copy the null character because it knows the end

* Be careful to copy a valid string!

strncpy(str, “A very long string test value”, 20); - - -> str don’t have null character. It has (…long string t) values.

* Safer to use

strncpy(str, “A very long string test value”, 19);

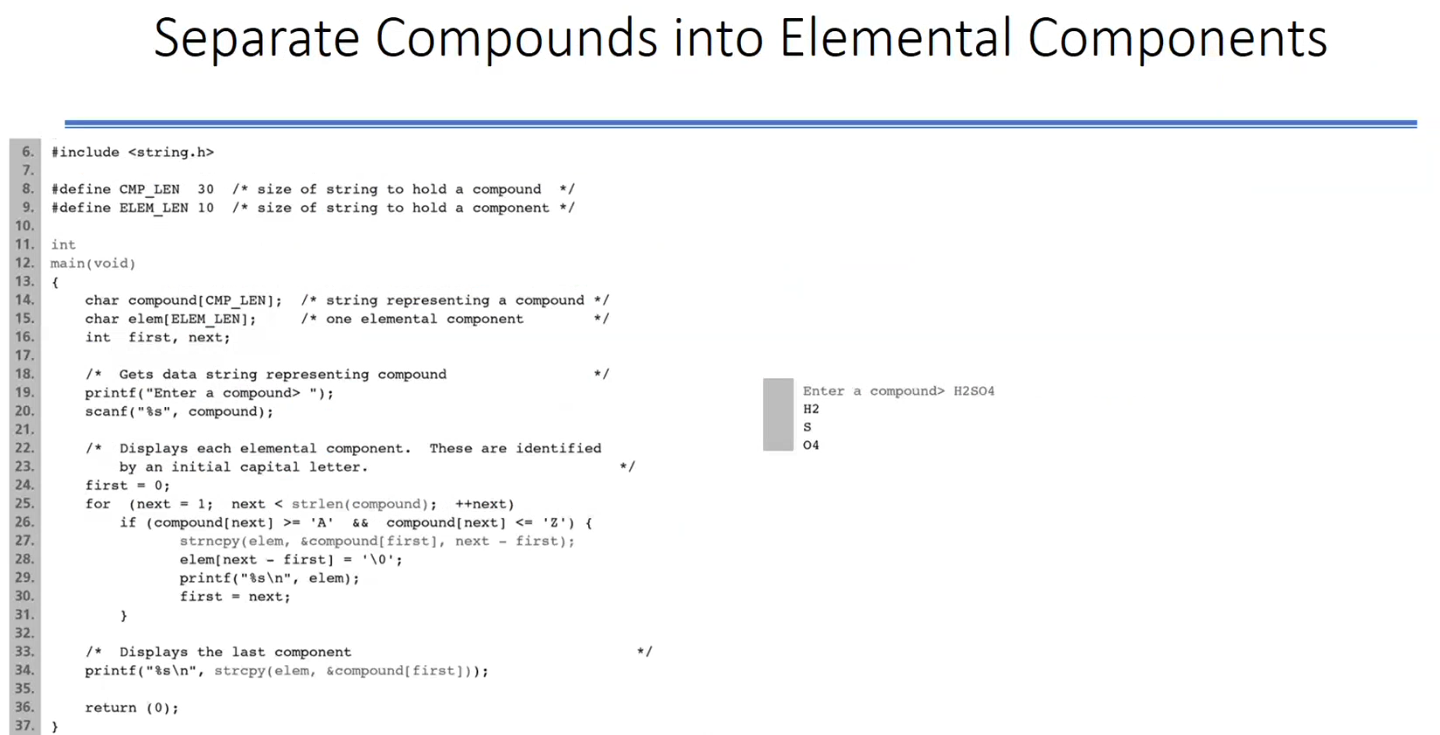
str[19] = ‘\0’;

With strncpy, in last parameter you should have a value that at least 1 bigger than the middle parameter to have a null character. If not, then you must add \0 manually.

char result[10], s1[15] = “Jan.30, 1996”;

strcpy(result, &s1[9]); - - -> You can do like this instead of just writing s1 in the second part. This copies starting from 9th index. We know result will be a string because s1 finishes with \0 as it is stirng.

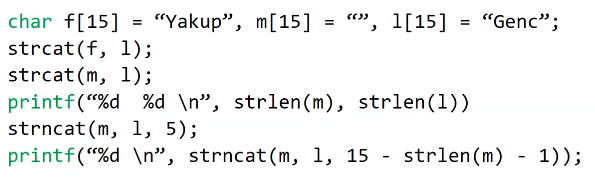
strncpy(result, &s1[4], 2); - - -> This will copy just 30 to result. But now result is not a string. It is just ‘3’ and ‘0’. There is no null character. You have to write one more line: result[2] = ‘\0’;

SEPARATE COMPOUNDS INTO ELEMENTAL COMPONENTS

\*\*strcpy will return pointer to elem in the 34th line. That line is for writing O4.

Concatenation

* Add a string at the end of the other string.
* strcat and strncat
  + Assumes sufficient space available for the concatenated string.



\*\*\* m[0] is ‘\0’

‘\0’ of the f will be overwritten with the first value of the l, and ‘\0’ character will be after the Genc.

Scanning a Full Line

* Input one complete line of data
  + Do not stop at space or tab characters
  + Do not store end-of-line (new-line, return, enter) character
  + Store string you have and null character

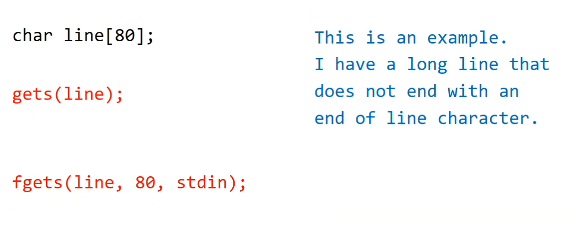
char line[80];

gets(line); - - -> Take what you write and put ‘\0’ to the end. If you write 10 letter including spaces, 11 space from the line array will be used. You must be careful to not to overflow.

* File input, fgets has different format
  + Final character is always ‘\0’
  + Stores ‘\n’ character if the line is not truncated

char \*fgets(char \*str, int n, FILE \*stream);

fgets(line, 80, inp);



gets just reads “This is an example.”, fgets reads all the lines.

String Comparison

* Comparison operators can not be used.
* Strings are implemented as arrays.
* str1 < str2 🡪 Alphabetical order
* strcmp: compares two strings and returns an integer

strcmp(str1, str2);

* + Has negative value if str1 is less than str2
  + Has value 0 if str1 is equal to str2
  + Has positive value if str1 is greater than str2
* strncmp: compares first n characters

strncmp(“Ali”, “Alice”, 3) - - -> returns 0 (both “Ali”)

strncmp(str1, &str2[3], 3) - - -> You can start searching something in your string wherever you want.

**ARRAYS OF POINTERS**

* Swapping 2 strings requires a lot of copying of characters to sort a list of strings.
  + Three copy operations per exchange. - - -> str1 to temp, str2 to str1, temp to str2
* Alternative approach: use array of pointers
  + Pointers to strings (arrays)
  + Sort the pointers not the strings
  + Saves the original order as well

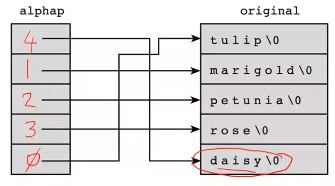
When you want to sort a list, you must do 3 copy operations every time:

* strcpy( temp, s1);
* strcpy( s1, s2);
* strcpy( s2, temp);

This means that, if your string’s size is 10, everytime you change one string with another, you have to change 30 bytes everytime.

If your strings are much bigger, 30 bytes may become sooo many bytes.

With sorting list of strings, this means a lot of work.

You can sort another list that including pointers to strings.

Here you can change just your alphap list that includes indexes of array of strings.

This means everytime you do 3 jobs again and now these 3 jobs become 3 x 4 = 12 bytes because every int number is 4 bytes. Also this doesn’t increase with the length of your string.

Instead of using indexes, we use pointers.

You can still reach the original array from array of pointers with simply alphap[index]

With this, you can do swapping with 3 x 2 (byte of address as example (008A)) = 6 bytes

You can define it with : char\* alphap[5];

For initializing :

for (i = 0; i < 5; ++i){

alphap[i] = original[i]; - - -> copying the address in original[i] to alphap[i]

// DON’T FORGET ORIGINAL IS ARRAY OF STRINGS !!!

}

For printing strings:

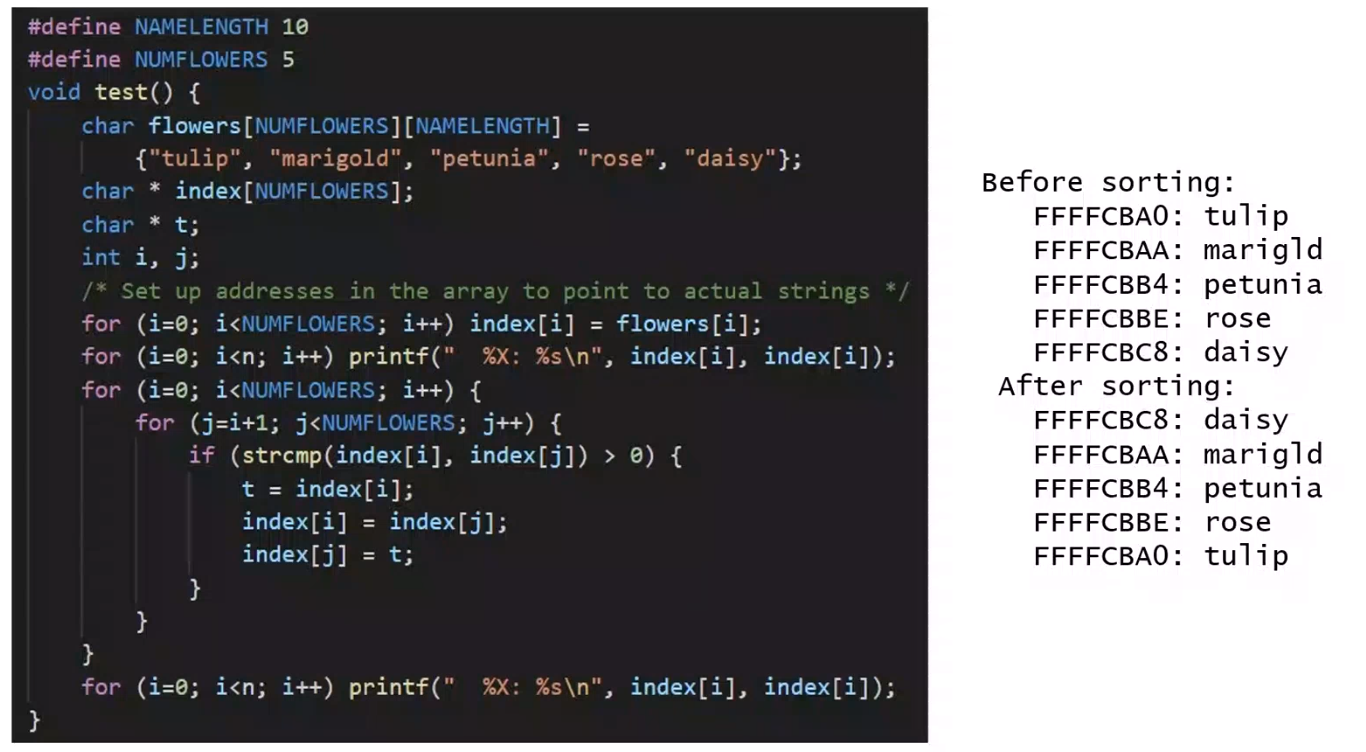
char original[5][10]; - - -> 5 strings, each have length 10

char\* alphap[5];

for( i = 0; i < 5; ++i)

printf(“%s\n”, alphap[i]);

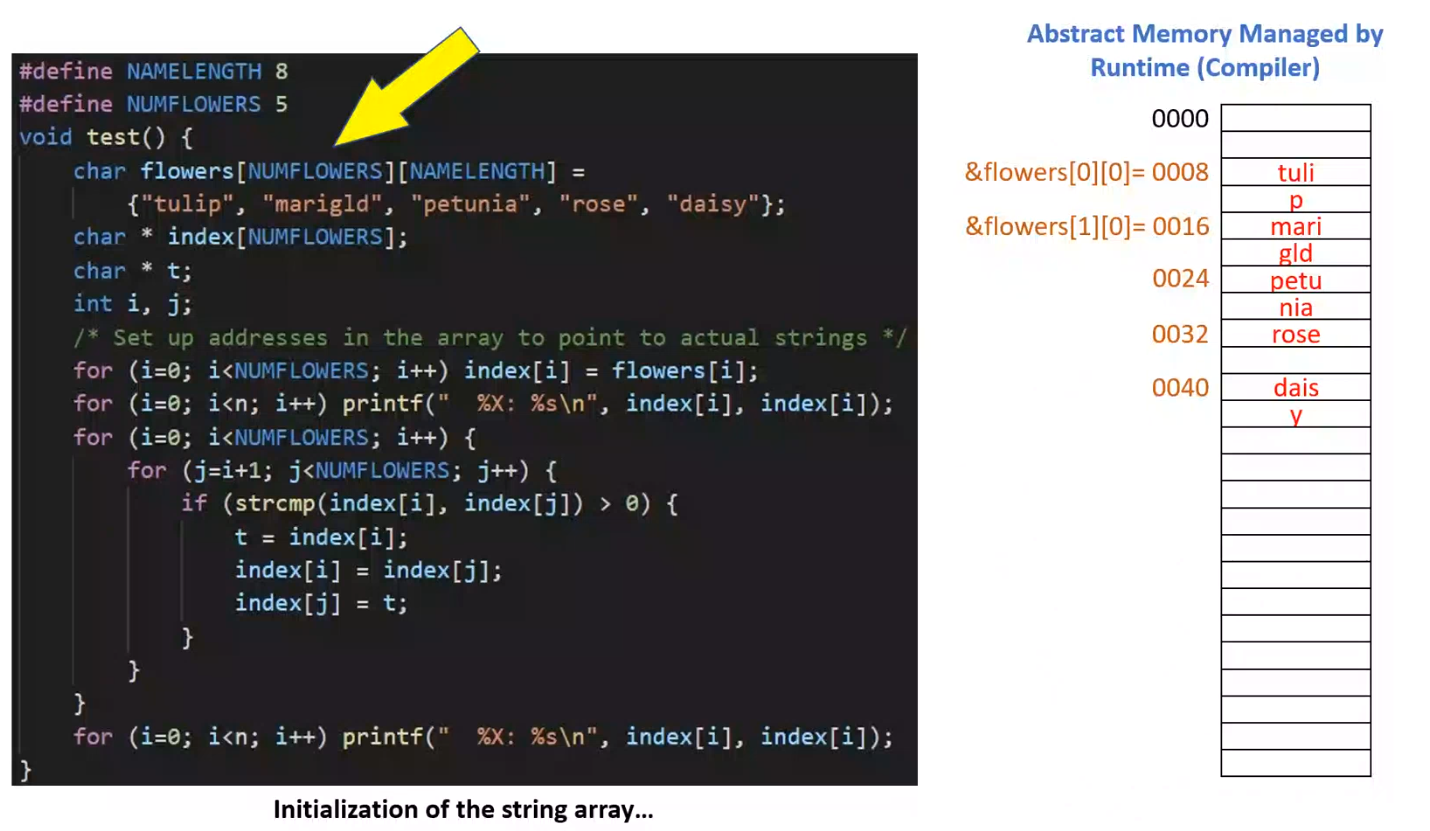
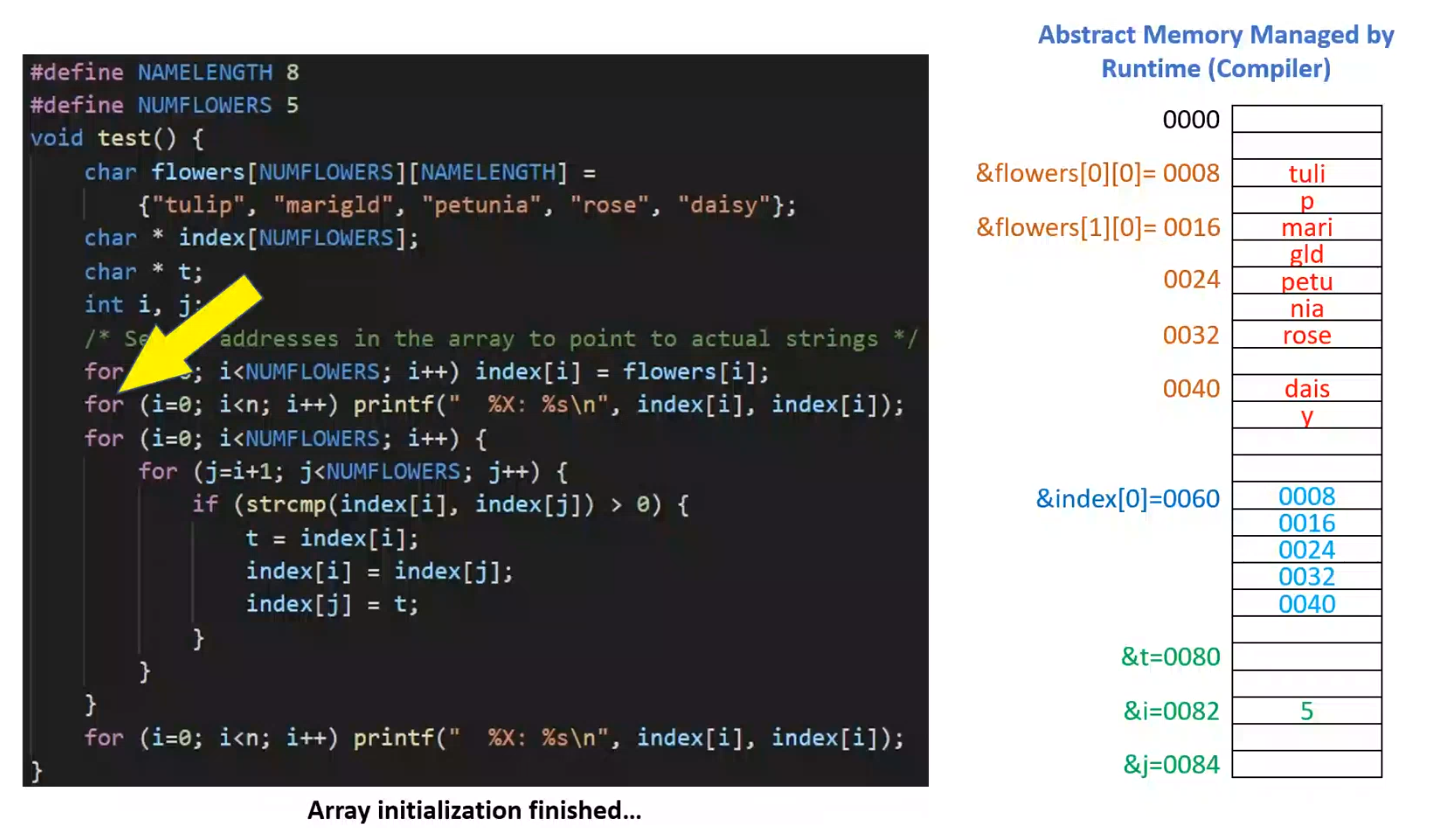
for( i = 0; i < 5; ++i)

 printf(“%s\n”, original[i];

You can also write index[i] = &flowers[i][0]; in the first for loop.

FOR TWO DIMENSIONAL ARRAY, IF YOU TYPE LIKE :

* ARRAY[0] - - - > THIS IS AN ADDRESS
* ARRAY[0][1] - - - > THIS IS WHAT YOU HAVE

Every cell has 4 bytes.

In second for, first we are printing address in hexadecimal form and then with %s you give a memory which is supposed to be a character. In this case, I am expecting it to be holding an array of characters terminated by a null character at some point.

CHARACTER OPERATIONS

Strings processing usually requires character manipulation

Character library provides several functions

* include ctype.h

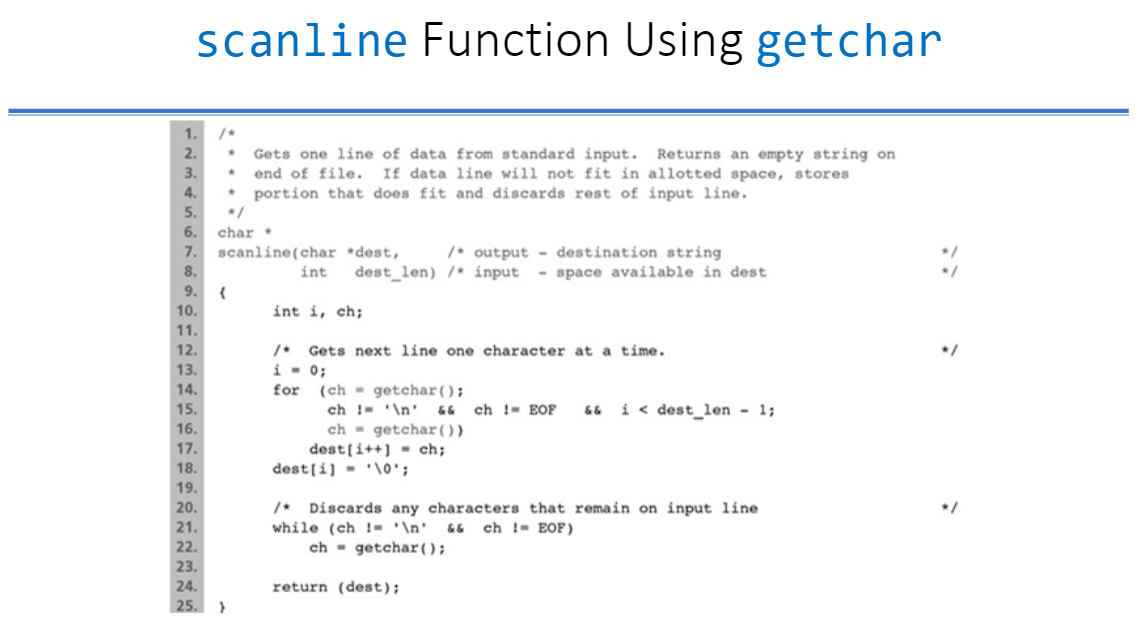
Character I/O

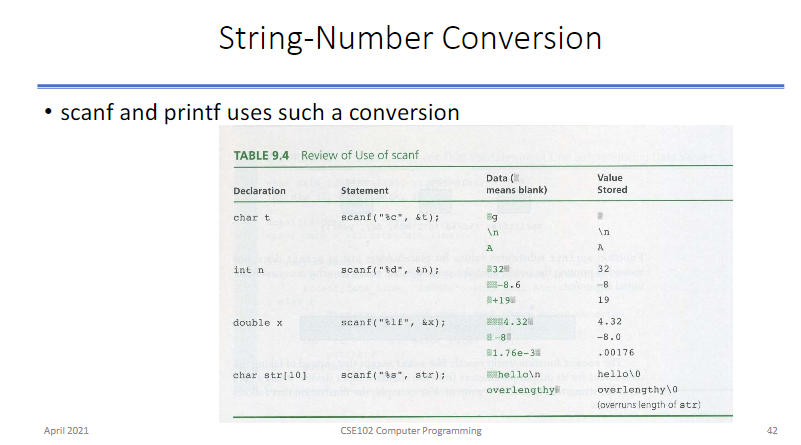
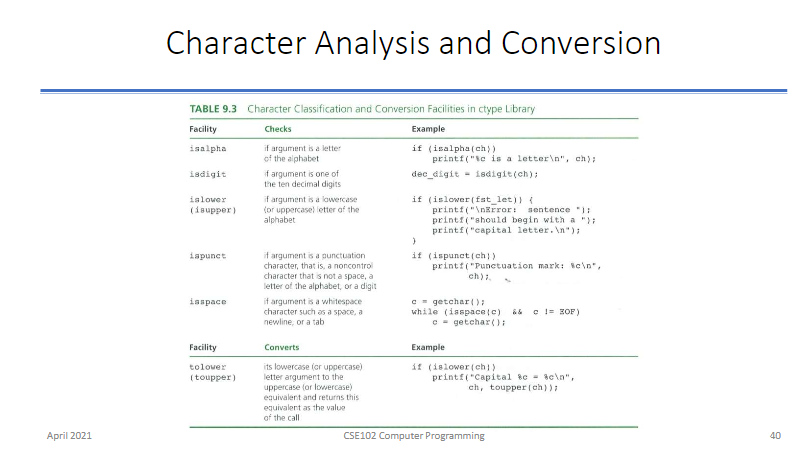
* getchar (and getc)
  + returns the next character from standard input (or file)
  + Return value is an integer
    - Return EOF if getchar end-of-file is reached.
    - The value of EOF is -1 which is not of type char.

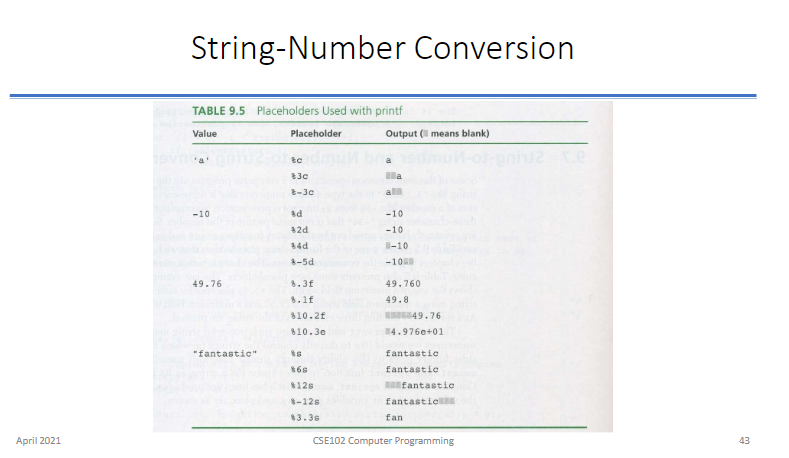
ch = getchar(); scanf(“%c”, &ch); ch = getc(stdin);

* putchar and putc are used to display a character

putchar(‘a’); putc(‘a’, stdout);







String - Number Conversion

* sscanf and sprint similar to scanf and printf
  + They perform the operation on a string.
  + sscanf : reads input from the parameter string
  + sprint : outputs into the parameter string

char s[100];

sprintf(s, “%d/%d/%d”, mon, day, year);

sscanf(“ 85 96.5 hello”, “%d %lf %s”, &n, &f, w);

* You can read the entire data as a line of input, verify its format and convert to correct values using sscanf.

For printf, we have also fprintf (for files) and sprintf.

In sprintf, result of entire thing will be printed in a string that is first parameter.

%c should have variable,

%s should have pointer/address.

ARRAYS AS ARGUMENTS

|  |
| --- |
| {x} |

|  |
| --- |
| {a} |

int sum\_p(int \*a, int n){

int i, sum = 0;

|  |
| --- |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |

for (i = 0; i < n; ++i) sum += a[i];

return sum;

}

int sum\_a(int a[], int n){

int i, sum = 0;

for (i = 0; i < n; ++i) sum += a[i];

return sum;

}

|  |
| --- |
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |

|  |
| --- |
| {x} |

int sum\_pa(int (\*a)[], int n){ /\* int \*\*a da olabilir \*/

int i, sum = 0;

|  |
| --- |
| {a} |

for (i = 0; i < n; ++i) sum += (\*a)[i];

return sum;

}

int main(){

int x[] = {1, 2, 3, 4, 5, 6};

printf(“Sum Pointer : %d\n”, sum\_p(x, 6));

printf(“Sum Array : %d\n”, sum\_a(x, 6));

printf(“Sum ArrayPtr : %d\n”, sum\_pa(&x, 6));

}

For the sum\_pa function, if you use “(\*a)[]” it is pointer to an integer pointer,

if you use “\*a[]” it is array of int pointers.

int \*ptr[5]; 🡪 Array of pointers