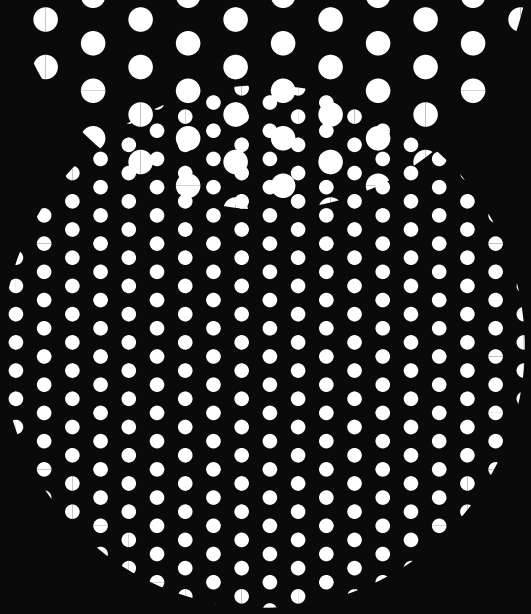


C Programming

8 bits = 1 byte

Integers — int(), 32 bits, 4 bytes (also Long()) — Long long(), 64 bits, 8 bytes

Floating — float(), 4 – 32 bytes → %f
double(), 8 – 64 bytes → %lf

Character — char(), 1 – 8 bits
↳ ASCII

Constants — for Char constants, use 'a' or 97 (ascii)

```
#define CONSTANT_NAME VALUE;
Const TYPE CONSTANT_NAME VALUE;
```

For Conversion → x = (int) my_float

Key words

auto	break	case	char	const	continue
default	do	double	else	enum	extern
float	for	goto	if	int	long
struct	switch	typedef	union	sizeof	static
volatile	while	unsigned	void		

Variable

```
int count = 20;
float temperature, result;
```

Increment Decrement Operators

```
#include <stdio.h>
int main()
{
    int num = 4;
    printf("value of num is %d\n", num);
    num++;  // ++num; i.e., num = num+1;
    printf("value of num is %d\n", num);
    num = 4;
    printf("value of num++ is %d\n", num++);
    printf("value of num is %d\n",num);
    printf("value of ++num is %d\n", ++num);
    printf("value of num is %d\n\n",num);
    return 0;
}
```

Output:
value of num is 4
value of num is 5
value of num++ is 4
value of num is 5
value of ++num is 6
value of num is 6

```
#include <stdio.h>
int main()
{
    int num = 4;
    printf("value of num is %d\n", num);
    num--;  // same as --num;
    printf("value of num is %d\n", num);
    num = 4;
    printf("value of num-- is %d\n", num--);
    printf("value of num is %d\n", num);
    printf("value of --num is %d\n", --num);
    printf("value of num is %d\n", num);
    return 0;
}
```

Output:
value of num is 4
value of num is 3
value of num-- is 4
value of num is 3
value of --num is 2
value of num is 2

Input scanf()

include <stdio.h>

scanf (control-string, argument-list);

```
#include <stdio.h>
int main( )
{
    int   n1, n2;
    float f1;
    double f2;
    printf("Please enter 2 integers:\n");
    scanf("%d %d", &n1, &n2);
    printf("The sum = %d\n", n1+n2);
    printf("Please enter 2 floats:\n");
    scanf("%f %lf", &f1, &f2);
    // Note: use %lf for double data
    printf("The sum = %f\n", f1+f2);
    return 0;
}
```

Output:
Please enter 2 integers:
5 10
The sum = 15
Please enter 2 floats:
5.3 10.5
The sum = 15.800000

d	signed decimal conversion of int
o	unsigned octal conversion of unsigned
x,X	unsigned hexadecimal conversion of unsigned
c	single character conversion
f	signed decimal floating point conversion → %lf for double data
s	string conversion

var = getchar()

d	signed decimal conversion of int
o	unsigned octal conversion of unsigned
x,X	unsigned hexadecimal conversion of unsigned
c	single character conversion
f	signed decimal floating point conversion
s	string conversion

Output printf()

printf (control-string, argument-list);

```
#include <stdio.h>
int main()
{
    int num1 = 1, num2 = 2;
    printf("%d + %d = %d\n", num1, num2, num1+num2);
    return 0;
}
```

conversion specifiers

```
#include <stdio.h>
int main( )
{
    int    num = 10;
    float  i = 10.3;
    double j = 100.3456;
    printf("int num = %d\n", num);
    printf("float i = %f\n", i);
    printf("double j = %f\n", j);
    /* by default, 6 digits are printed
       after the decimal point */
    printf("double j = %.2f\n", j);
    printf("double j = %10.2f\n", j);
    /* formatted output */
    return 0;
}
```

Output:
int num = 10
float i = 10.300000
double j = 100.345600
double j = 100.35
double j = 100.35

putchar()

putchar('\n')
↳ manually put new line

★ for scan Double %lf

Logical Operator

operator	example	meaning
!	!(num < 0)	not
&&	(num1 > num2) && (num2 >num3)	and
||	(ch == '\t') || (ch == ' ')	or

!	not (most imp)
* /	multiply and divide
+ -	add and subtract
< <= > >=	less, less or equal, greater or equal
== !=	equal, not equal
&&	logical and
||	logical or (Last to execute)

★ $(x != 0)\&\&(\frac{2}{x} != 1)$
$(\frac{2}{x} != 1)\&\&(x != 0)$

When x=0, the bottom will short circuit and cause the thing to fail. The first still works

Conditional Operator ?

```
expression_1 ? expression_2 : expression_3
```

The **value** of this expression depends on whether *expression_1* is true or false.

if expression_1 **is true**
    the value of the expression is that of *expression_2*
else
    the value of the expression is that of *expression_3*

or example:

```
max = (x > y) ? x : y;      <==>    if (x > y)
                                        max = x;
                                    else
                                        max = y;
```

Escape Seq

'\0___' = Octal number

\a	Alarm or Beep
\b	Backspace
\f	Form Feed
\n	New Line
\r	Carriage Return
\t	Tab (Horizontal)
\v	Vertical Tab
\\	Backslash
\'	Single Quote
\"	Double Quote
\?	Question Mark
\ooo	octal number
\xhh	hexadecimal number
\0	Null

Looping

if( )

```
if (condition) {
   Statement;
}
else {
   Statement;
}
```

Switch → must be constant

```
switch (expression) {
  case constant_1:
        statement_1;
        break;
  case constant_2:
        statement_2;
        break;
  case constant_3:
        statement_3;
        break;
  default:
        statement_D;
}
```

- **switch, case, break** and **default** - reserved keywords.
- The result of **expression** in ( ) must be of **integral type**.
- **constant_1, constant_2, ...** are called **labels**.
  - must be an **integer constant**, a **character constant** or an integer constant expression, e.g. 3, 'A', 4+'b', 5+7, ..., etc.
  - must deliver a **unique integer value**. Duplicates are not allowed.
  - may also have **multiple labels** for a statement, for example, to allow both lower and upper case selection.

switch: An Example

```
#include <stdio.h>
int main() {
   char choice;   int num1, num2, result;
   printf("Enter your choice (A, S or M) => ");
   scanf("%c", &choice);
   printf("Enter two numbers: ");
   scanf("%d %d", &num1, &num2);
   switch (choice)    {
     case 'a':
     case 'A':  result = num1 + num2;
       printf(" %d + %d = %d\n", num1,num2,result);
       break;
     case 's':
     case 'S':  result = num1 - num2;
       printf(" %d - %d = %d\n", num1,num2,result);
       break;
     case 'm':
     case 'M':  result = num1 * num2;
       printf(" %d * %d = %d\n", num1,num2,result);
       break;
     default :  printf("Not one of the proper choices.\n");
   }
   return 0;
}
```

Output

```
Enter your choice (A, S or M) => S
Enter two numbers: 9 5
9 – 5 = 4
```

while( )

do - while → good for menu interface

```
do {
      /* display menu */
      printf("Input a number >= 1 and <= 5:  ");
      scanf("%d",&input);
      if (input > 5 || input < 1)
         printf("%d is out of range.\n", input);
} while (input > 5 || input < 1);
```

for (initialize; test; update)

```
for (t = 1;     t <= timeLimit;     t++) {
   distance = 0.5 * ACCELERATION * t * t;
   printf("Dist after %d sec is %d \
        feet.\n", t, distance);
}
```

Function

```
float findMax(float x, float y)  // header
{
   // function body
   float maxnum;

   if (x >= y)
      maxnum = x;
   else
      maxnum = y;

   return maxnum;
}
```

```
Return_type Function_name (Parameter_list)
```

- **Function_name**
  - specifies the name given to the function. Try to give a meaningful name to the function.
- **Parameter_list**
  - specifies a list of parameters which contains the data that are passed in by the calling function.
- **Return_type**
  - specifies the type of the data to be returned to the calling function.

Parameters define the data passed into the function.
A function can have no parameter, one parameter or many parameters.

```
type parameterName[, type parameterName]
```

Example: float findMax(float x, float y)

Each parameter has:
- parameter name
- data type (such as int, char, etc.) of the parameter

The function assumes that these parameter inputs will be supplied to the function when they are being called.

Return Type is the data type returned from the function, it can be int, float, char, void, or nothing.
The syntax for the **return** statement is **return (expression);**

void – the function will not return any value.

```
void hello_n_times(int n)
{
   int count;
   for (count = 0; count < n; count++)
      printf("hello\n");
   /* no return statement */
}
```

nothing – if defined with no type, the default type is **int**.

```
successor(int num)   /* i.e. int successor(int num) */
{
   return num + 1; /* has a return statement */
}
```

Function Prototypes (make func available)

☆ Before the main()

```
void hello (int n);
double distance(double, double);
```

☆ Put this after main(){

```
}
int func( x, y){
...
}
```

Global vs Local vs Static

```
#include  <stdio.h>
int g_var = 5;                  // global variable –  has file scope
int fn1(int, int);
float fn2(float);

int main( ) {
     char reply;                // local - these two variables are only
     int num;                   // known inside main() function - block scope
     ...
}
int fn1(int x, int y)           //  local x,y - formal parameters  are only
(                               //  known inside this function – block scope
     float fnum;                //  local - these two variables are known
     int temp;                  //  in this function only – block scope
     g_var += 10;
     ...
}
float fn2(float n)  {
     float temp;                //  local and block scope
     ...
}
```

```
#include <stdio.h>
void function();
int main()
{
   int i;
   for (i=0; i<3; i++)     // calling the fn three times
      function();
   return 0;
}
void function()
{
   static int static_var = 0;    /* static variable */
   int local_var = 0;            /* local variable */
   ++static_var;
   ++local_var;
   printf("Static variable: %d\n", static_var);
   printf("Local variable: %d\n", local_var);
}
```

Output

```
Static variable: 1
Local variable: 1
Static variable: 2
Local variable: 1
Static variable: 3
Local variable: 1
```

Note:

Local variable – the variable disappears after each function execution.

Static variable (like global) – the variable stays until the end of program execution.

Static > Global

Static is still a local function that retains information within the function

Address

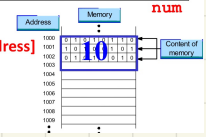
```
#include  <stdio.h>
int main()
{
     int num = 5;

     printf("num = %d, &num = %p\n", num, &num);
     scanf("%d", &num);
     printf("num = %d, &num = %p\n", num, &num);
}
```

Printing the memory address of the variable

Output

```
num = 5, &num = 1000 [address]
10
num = 10, &num = 1000
```




```
int num=5;
```

(1) num

– It is a variable of data type int and 4 bytes of memory are allocated.
– Its memory location is used to store the integer value of the variable.

Example:

(2) &num

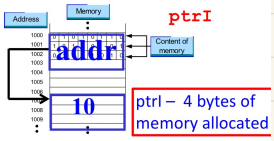
– It refers to the memory address of the variable which is used to store the int value of the variable.

Note: You may also print the address of the variable using the printf() statement.

Pointer Varibles

Pointer variable – different from the primitive variable **num** (variable of primitive data type such as int, float, char) declared earlier, it stores the **address** of memory location of a data object. A **pointer variable** is declared by, for example:

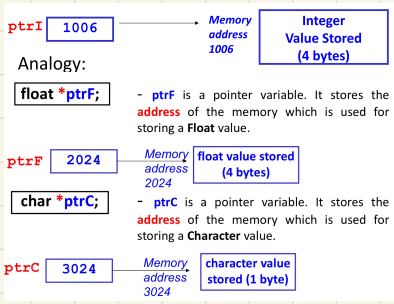
```
int *ptrI;
or  int * ptrI;
or  int* ptrI;
```



ptrI is a pointer variable. It does **not** store the **value** of the variable. It stores the **address** of the memory which is used for storing an Int value.

Pointer to integer

int *a = Address of 4byte var a



☆ Printing pointer address ⇝ %p

☆ Dereferencing ⇝ *ptr

```
#include <stdio.h>
int main()
{
   int num = 3; // integer var
   int *ptr;    // pointer var

   ptr = &num;

   printf("num = %d, &num = %p\n", num, &num);

   printf("ptr = %p, *ptr = %d\n", ptr, *ptr);

   *ptr = 10;
   // What will be the values: *ptr, num, &num?
   printf("num = %d, &num = %p\n", num, &num);
   return 0;
}
```

Statement	Operation
ptr = #	ptr 1024 → num 3 Address = 1024
*ptr = 10;	ptr 1024 → num 10 Address = 1024

Output
num = 3, &num = 1024
ptr = 1024, *ptr = 3
num = 10, &num = 1024
[*ptr = 10]

```
int a = 600;
int * b;
b = &a  → address
*b = 600
```
value of pointer b

Parameter Passing

Call by Value

→ When using return in function, it provides a single value to be passed back. The argument will be replaced by the paremeter inside the function and will take on a new value

- **Call by Value** – The **communication** between a function and the calling body is done through **arguments** and the **return value** of a function.

```
#include <stdio.h>
int add1(int);

int main( )
{
   int num = 5;
   num = add1(num);   // num – called argument
   printf("The value of num is: %d", num);
   return 0;
}

int add1(int value)   // value – called parameter
{
    value++;
    return value;
}
```

num 5 -> 6

value 5 -> 6

Output
The value of num is: 6

Call by function

Call by reference: the parameter in the function holds the address of the argument variable, i.e., the **parameter** is a **pointer variable**. Therefore,

- In the **function header**'s parameter declaration list, the **parameters** must be prefixed by the **indirection operator** *.
  E.g. void distance(double *x, double *y)
- In the **function call**, the **arguments** must be **pointers** (or using the address operator as the prefix).
  E.g. distance(&x1, &y1);

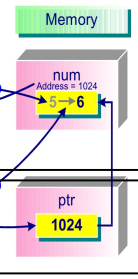
1) Function definition
```
↳ void func(int* a){
    ....
  }
```
2) Calling function
```
↳ int num;
  func(&num);
```
3) Function Prototype
↳

Call by reference

```
#include <stdio.h>
void add2(int *ptr);
 int main()
{
   int num = 5;
  /*passing the address of num*/
   add2(&num);
   printf("Value of num is: %d",
          num);
   return 0;
}

void add2(int *ptr)
{
    ++(*ptr);
}
```

Output
Value of num is 6



Call by Reference: Key Steps

In the **function definition**, the **parameter** must be prefixed by **indirection operator ***:

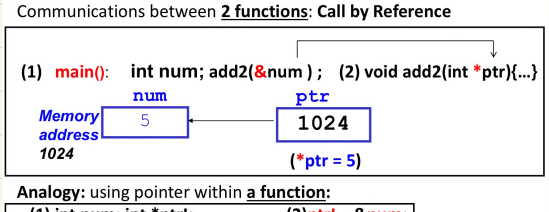
add2( ): void add2(int *ptr) { ...}

In the **calling function**, the **arguments** must be pointers (or using **address** operator as the prefix):

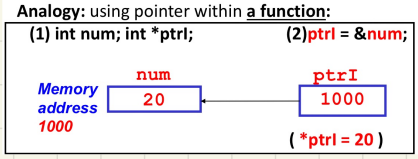
main( ): int num; add2(&num) ;

Call by Reference: Analogy

Communications between **2 functions**: **Call by Reference**



Analogy: using pointer within **a function**:

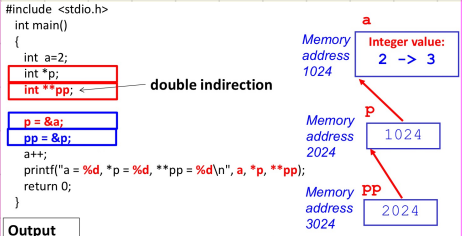


Double Indirection

```
#include <stdio.h>
 int main()
 {
   int  a=2;
   int *p;
   int **pp;

   p = &a;
   pp = &p;
   a++;
   printf("a = %d, *p = %d, **pp = %d\n", a, *p, **pp);
   return 0;
 }
```

double indirection



Output
a = 3
*p = 3
**pp = 3

Note: it could also be int ***ppp; etc. The idea remains the same.

```
#include<stdio.h>
void function1 (int a, int *b); void function2 (int c, int *d);
void function3 (int h, int *k);
int main() {
    int x, y;
    x = 5;  y = 5;                          /* (i) */
    function1(x, &y);   address             /* (x) */
    return 0;
}                                  pointer
void function1(int a, int *b) {             /* (ii) */
    *b = *b + a;                            /* (iii) */
    function2(a, b);                        /* (ix) */
}                                  pointer
void function2(int c, int *d) {             /* (iv) */
    *d = *d * c;                            /* (v) */
    function3(c, d);                        /* (viii) */
}                                  pointer
void function3(int h, int *k) {             /* (vi) */
    *k = *k - h;                            /* (vii) */
}
```

Arrays

(Same data type)

* Cannot change size (index start with 0)

One Dimention

- Data are in two forms

1) Primitive Varibles: Store values (int, float, char)

2) Reference for pointer variables

- Declaration of arrays **without initialization**:

```
char name[12];          /* array of 12 characters */
float sales[365];       /* array of 365 floats */
int states[50];         /* array of 50 integers */
int *pointers[5];       /* array of 5 pointers to integers */
```

- When an array is declared, some **consecutive memory** locations are allocated by the compiler for the whole array (**2 or 4** bytes will be allocated for an integer depending on machine):
  total_memory = **sizeof**(type_specifier)*array_size;
  e.g. char name[12]; - total_memory = 1*12 = 12 bytes
- The size of array must be integer constant or constant expression in declaration:

```
e.g.  char name[i];     // i is a variable  ==> illegal
      int states[i*6];  // i is a variable  ==> illegal
```

- **Initialize array variables** at declaration:

```
int days[12]={31,28,31,30,31,30,31,31,30,31,30,31};
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
days	31	28	31	30	31	30	31	31	30	31	30	31

- **Partial array initialization**: E.g. (initialize first 7 elements)

```
int days[12]={31,28,31,30,31,30,31};
/* remaining elements are initialized to zero */
```

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
days	31	28	31	30	31	30	31	0	0	0	0	0

Operating on Arrays

Accessing array elements:

```
sales[0] = 143.50;        // using array index
if (sales[23] == 50.0) ...
```

Subscripting: The element indices range from **0** to **n-1** where n is the declared size of the array.

```
char name[12];
name[12] = 'c';           // index out of range – common error
```

Working on array values:

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
days	31	28	31	30	31	30	31	31	30	31	30	31

(1) days[1] = 29; - OK ??
(2) days[2] = days[2] + 4; - OK ??
(3) days[3] = days[2] + days[3]; - OK ??
(4) days[1] = {2,3,4,5,6}; - OK? NOT OK!! (int Only)

Array for Loop

Example 1: Printing Values

```
#include <stdio.h>
#define MTHS 12          /* define a constant */
int main( )
{
   int i;
   int days[MTHS] = {31,28,31,30,31,30,31,31,30,31,30,31};

   /* print the number of days in each month */
   for (i = 0 ;  i < MTHS ;  i++ )
      printf("Month %d has %d days\n", i+1, days[i]);
   return 0;
}
```

Output
Month 1 has 31 days.
Month 2 has 28 days.
...
Month 12 has 31 days.

condition: i < MTHS

days	31	28	30	31	30	31	30	31	30	31	30	31
	0	1	2	3	4	5	6	7	8	9	10	11

Example 2: Searching for a Value

```
#include <stdio.h>
#define SIZE 5          /* define a constant */
int main ( )
{
   char myChar[SIZE] = {'b', 'a', 'c', 'k', 's'};
   int i;
   char searchChar;
   // Reading in user's input to search
   printf("Enter a char to search: ");
   scanf("%c", &searchChar);
   // Traverse myChar array and output character if found
   for (i = 0;   i < SIZE;   i++) {
      if (myChar[i] == searchChar){
         printf ("Found %c at index %d", myChar[i], i);
         break;     //break out of the loop
      }
   }
   return 0;
}
```

Output
Enter a char to search: a
Found a at index 1

Example 3: Finding the Maximum Value

```
#include <stdio.h>
int main( )
{
   int index, max, numArray[10];
   max = -1; printf("Enter 10 numbers: \n");
   for (index = 0; index < 10; index++)
         scanf("%d", &numArray[index]);
   // Find maximum from array data
   for (index = 0;  index < 10;  index++) {
         if (numArray[index] > max)
               max = numArray[index];
   }
   printf("The max value is %d.\n", max);
   return 0;
}
```

This example shows how to find the largest value in an array of numbers.

Inputting into array here

Output
Enter 10 numbers:
4 3 8 9 15 25 3 6 7 9
The max value is 25.

numArray	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
	4	3	8	9	15	25	3	6	7	9

Memory address : 1021 1023 1025 1027 1029 102B 102D 102F 1031 1033

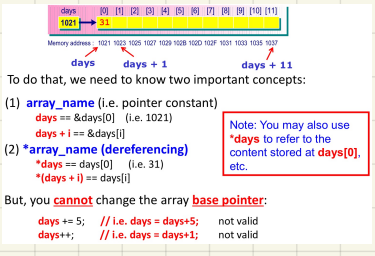
&numArray[index]

Array Number

When define array: int days[12];

Creates pointer to first array value

& days[0] is address to 1st element



To do that, we need to know two important concepts:

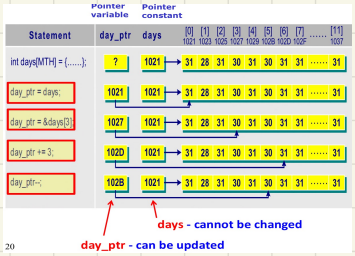
(1) **array_name** (i.e. pointer constant)
   days == &days[0] (i.e. 1021)
   days + i == &days[i]

(2) ***array_name (dereferencing)**
   *days == days[0] (i.e. 31)
   *(days + i) == days[i]

Note: You may also use *days to refer to the content stored at days[0], etc.

But, you **cannot** change the array **base pointer**:

```
days += 5;    // i.e. days = days+5;    not valid
days++;       // i.e. days = days+1;    not valid
```



20

Example:

Finding Maximum: Using Pointer Constants

```
#include <stdio.h>
int main( )                    Pointer constant
{
   int index, max, numArray[10];
   printf("Enter 10 numbers: \n");
   for (index = 0;  index < 10;  index++)
         scanf("%d", numArray + index);
   // Find maximum from array data
   max = *numArray;
   for (index = 1;  index < 10;  index++)
   {
         if (*(numArray + index) > max)
            max = *(numArray + index);
   }
   printf("The max value is %d.\n", max);
   return 0;
}
```

Using index for reading input:

```
for (index = 0; index < 10; index++)
   scanf("%d", &numArray[index]);
```

Using index for processing:

```
max = numArray[0];
for (index = 1;  index < 10;
                 index++)
{
   if (numArray[index] > max)
      max = numArray[index];
}
```

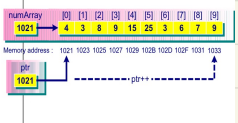
numArray [0] .. .. [9]

	0	1	2	3	4	5	6	7	8	9

Finding Maximum: Using Pointer Variables

```
#include <stdio.h>
int main( ){
   int index, max, numArray[10];
   int *ptr;
   ptr = numArray;
   printf("Enter 10 numbers: \n");
   for (index = 0; index < 10; index++)
         scanf("%d", ptr++);
   // Find maximum from array data
   ptr = numArray;
   max = *ptr;
   for (index = 0; index < 10; index++) {
         if (*ptr > max)
            max = *ptr;
         ptr++;
   }
   printf("max is %d.\n", max);
   return 0;
}
```

Output
Enter 10 numbers:
4 3 8 9 15 25 3 6 7 9
max is 25.



Function for Arrays

Function header

```
void fn1(int table[ ], int size)
{
  ....
}
```

```
or  void fn2(int table[TABLESIZE])
{
  ....
}
```

```
or  void fn3(int *table, int size)
{
  ....
}
```

The prototype of the function:

```
void fn1(int table[ ], int size);
```
or
```
void fn2(int table[TABLESIZE]);
```
or
```
void fn3(int *table, int size);
```

Note: **size** and **TABLESIZE** are the **data size** to be processed in the array

Any dimensional array can be passed as a function argument, e.g. we can **call the function**:

```
fn1(table, n);        /* calling a function */
```

where **fn1()** is a function and **table** is an one-dimensional array, and **n** is the size of the array **table**.

An **array table** is passed in using **call by reference** to a function.

This means the **address** of the **first element** of the array is passed to the function.

Using in function

```
#include <stdio.h>
int maximum(int table[ ], int n);
int main( )
{
    int max, index, n;
    int numArray[10];   // Using index for input

    printf("Enter the number of values: ");
    scanf("%d", &n);
    printf("Enter %d values: ", n);
    for (index = 0; index < n; index++)
        scanf("%d", &numArray[index]);

    // find maximum          // Calling the function
    max = maximum(numArray, n);
    printf("The maximum value is %d\n", max);

    return 0 ;
}
```

Output

```
Enter the number of
values:  10
Enter 10 values: 0 1 2 3 4
5 6 7 8 9
The maximum value is 9
```

Input max value & array name

```
int maximum(int table[ ], int n)
{
    int i, max;                    Using array
                                   indexing
    max = table[0];
    for (i = 1; i < n; i++)
        if (table[i] > max)
            max = table[i];
    return max;
}
```

Two Dimensional Array

Declared as **consecutive** pairs of brackets.

E.g. a **2-dimensional** array is declared as follows:

```
int x[3][5];
```
// a 3-element array of 5-element arrays

E.g. a **3-dimensional** array is declared as follows:

```
char x[3][4][5]; // a 3-element array of 4-element arrays of
                    5-element arrays
```

ANSI C standard requires a minimum of **6 dimensions** to be supported.

Initializing multidimensional arrays: enclose each row in braces.

```
int x[2][2] = { { 1, 2},   /* row 0 */
                { 6, 7} }; /* row 1 */
```
or
```
int x[2][2] = { 1, 2, 6, 7};
```

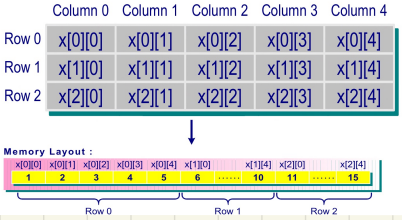
Partial initialization:

```
int exam[3][3] = { {1,2}, {4}, {5,7} };
int exam[3][3] = { 1,2,4,5,7 };
        i.e. = { {1,2,4}, {5,7}};
```

int x[3][5];

Row-major order i.e. x[row][column]

	Column 0	Column 1	Column 2	Column 3	Column 4
Row 0	x[0][0]	x[0][1]	x[0][2]	x[0][3]	x[0][4]
Row 1	x[1][0]	x[1][1]	x[1][2]	x[1][3]	x[1][4]
Row 2	x[2][0]	x[2][1]	x[2][2]	x[2][3]	x[2][4]



Example: sum of row

```
#include <stdio.h>
int main()
{   // declare an array with initialization
    int array[3][3]={                    column
            row     {5, 10, 15},
                    {10, 20, 30},
                    {20, 40, 60}
                    };
    int row, column, sum;
    /* compute sum of row - traverse each row first */
    for (row = 0; row < 3; row++)              // nested loop
    {
        /* for each row – compute the sum */
        sum = 0;
        for (column = 0;  column < 3;  column++)
            sum += array[row][column];
        printf("Sum of row %d is %d\n", row, sum);
    }
    return 0;
}
```

Output

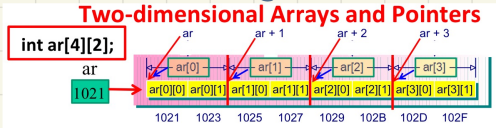
```
Sum of row 0 is 30
Sum of row 1 is 60
Sum of row 2 is 120
```

Nested Loop

2D Array Pointer

Two-dimensional Arrays and Pointers

int ar[4][2];



- **ar** - the address of the **1st element** of the array. In this case, the 1st element is an **array of 2 ints**. So, **ar** is the address of a **two**-int-sized object.

  ar == &ar[0]
  ar + 1 == &ar[1]
  ar + 2 == &ar[2]
  ar + 3 == &ar[3]

  Note: Adding 1 to a pointer or address yields a value larger by the size of the referred-to object.
  e.g. **ar** has the same address value as **ar[0]**
  ar+1 has the same address value as **ar[1]**, etc.

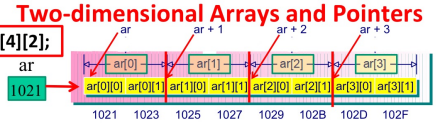
- **ar[0]** is an array of **2** integers, so **ar[0]** is the **address** of **int-sized object**.

  ar[0] == &ar[0][0]
  ar[1] == &ar[1][0]
  ar[2] == &ar[2][0]
  ar[3] == &ar[3][0]

  Note:
  - **ar[0]** has the same address as **ar[0][0]**;
  - **ar[0]+1** refers to the address of **ar[0][1]** (i.e. 1023)

Two-dimensional Arrays and Pointers

int ar[4][2];



- **Dereferencing** a pointer or an address (apply * **operator**) yields the **value** represented by the referred-to object.

  ar == &ar[0] *ar == ar[0] **(by dereferencing)**
  ar + 1 == &ar[1] ***(ar + 1) == ar[1]**
  ar + 2 == &ar[2] ***(ar + 2) == ar[2]**
  ar + 3 == &ar[3] ***(ar + 3) == ar[3]**

- Similarly

  ar[0] == &ar[0][0] *ar[0] == ar[0][0] **(dereferencing)**
  ar[1] == &ar[1][0] ***ar[1] == ar[1][0]**
  ar[2] == &ar[2][0] ***ar[2] == ar[2][0]**
  ar[3] == &ar[3][0] ***ar[3] == ar[3][0]**


```
#include <stdio.h>
int main() {
  int ar[3][3]= {
    {5, 10, 15},
    {10, 20, 30},
    {20, 40, 60}
  };
  int i, j;
    // (1) using indexing approach
  for (i=0; i<3; i++)
    for (j=0; j<3; j++)
      printf("%d ", ar[i][j]);
  printf("\n");
    // (2) using the pointer formula
  for (i=0; i<3; i++)
    for (j=0; j<3; j++)
      printf("%d ", *(*(ar+i)+j));
  return 0;
}
```

Output

```
5 10 15 10 20 30 20 40 60
5 10 15 10 20 30 20 40 60
```

*ar = ar[0]
*ar[0] = ar[0][0]
**ar = ar[0][0]

*a[m][n] == *(*(ar+m)+n)

2D Function

```
void fn(int array[2][4])
{
    ....
}
```
or
```
void fn(int array[ ][4])
{
    ....
}
```
/*note that the first dimension can be excluded*/

In the above definition, the **first dimension can be excluded** because the C compiler does not need the information of the first dimension.

Why the First Dimension can be Omitted?

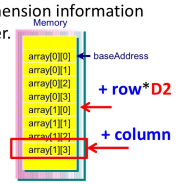
- For example, in the assignment operation: array[1][3] = 100; requests the compiler to compute the address of array[1][3] and then place 100 to that address.
- In order to compute the address, the dimension information of the array must be given to the compiler.
- Let's redefine array as

int array[D1][D2]; // with D1=2, D2=4

The address of array[1][3] is computed as:

baseAddress + row * D2 + column
==> baseAddress + 1 * 4 + 3
==> baseAddress + 7

The baseAddress is the address pointing to the beginning of array.



Example:

```
#include <stdio.h>
int sum_all_rows(int array[ ][3]);
int sum_all_columns(int array[ ][3]);
int main()
{
    int ar[3][3]= {
                   {5, 10, 15},
                   {10, 20, 30},
                   {20, 40, 60}
    };
    int total_row, total_column;
    total_row = sum_all_rows(ar);  // sum of all rows
    total_column = sum_all_columns(ar); //all columns
    printf("The sum of all elements in rows is %d\n", total_row);
    printf("The sum of all elements in columns is %d\n", total_column);
    return 0;
}
```

ar → 5 | 10 | 15 | 10 | 20 | 30 | 20 | 40 | 60

Output
The sum of all elements in rows is 210
The sum of all elements in columns is 210

```
int sum_all_rows(int array[ ][3]){
    int row, column;
    int sum=0;
    for (row = 0; row < 3; row++)
    {
        for (column = 0; column < 3; column++)
            sum += array[row][column];
    }
    return sum;
}
int sum_all_columns(int array[ ][3]){
    int row, column;
    int sum=0;
    for (column = 0; column < 3; column++)
    {
        for (row = 0; row < 3; row++)
            sum += array[row][column];
    }
    return sum;
}
```

ar, main(): 5 | 10 | 15 | 10 | 20 | 30 | 20 | 40 | 60
array
array

Pointers

Applying 1-D Array to Process 2-D Arrays in Functions: Using Pointers

```
#include <stdio.h>
void display1(int *ptr, int size);
void display2(int ar[ ], int size);
int main()
{
    int array[2][4] = { 0, 1, 2, 3, 4, 5, 6, 7 };
    int i;

    for (i=0; i<2; i++) { /* as 2-D Array */
        display1(array[i], 4);
    }

    display1(array, 8);  /* as 1-D array */

    return 0;
}
```

Row: array[0] array[1]

```
// Using pointers
void display1(int *ptr, int size)
{
    int j;

    printf("Display1 result: ");
    for (j=0; j<size; j++)
        printf("%d ", *ptr++);
    putchar('\n');
}
```

ptr

Output:
Display1 result: 0 1 2 3
Display1 result: 4 5 6 7
Display1 result: 0 1 2 3 4 5 6 7

Starts at main address then increment upwards by 1, treat as 1D array

Indexing

Applying 1-D Array to Process 2-D Arrays in Functions: Using Indexing

```
#include <stdio.h>
void display1(int *ptr, int size);
void display2(int ar[ ], int size);
int main()
{
    int array[2][4] = { 0, 1, 2, 3, 4, 5, 6, 7 };
    int i;

    for (i=0; i<2; i++) { /* as 2-D Array */
        display2(array[i], 4);
    }

    display2(array, 8);  /* as 1-D array */

    return 0;
}
```

array[0] array[1]

```
// Using indexes
void display2(int ar[ ], int size)
{
    int k;

    printf("Display2 result: ");
    for (k=0; k<size; k++)
        printf("%d ", ar[k]*5);
    putchar('\n');
}
```

ar

Basically you only consider the mini arrays

Output:
Display2 result: 0 5 10 15
Display2 result: 20 25 30 35
Display2 result: 0 5 10 15 20 25 30 35

Starts at main address then increment upwards by 1, treat as 1D array

Example: minMax()

Write a C function minMax() that takes a 5x5 two-dimensional array of integers *a* as a parameter. The function returns the minimum and maximum numbers of the array to the caller through the two parameters *min* and *max* respectively. [using call by reference]

```
#include <stdio.h>
void minMax(int a[5][5], int *min, int *max);
int main()
{
    int A[5][5];
    int i, j;
    int min, max;

    printf("Enter your matrix data (5x5): \n");
    // nested loop
    for (i=0; i<5; i++)
        for (j=0; j<5; j++)
            scanf("%d", &A[i][j]);
    minMax(A, &min, &max);
    printf("min = %d; max = %d", min, max);
    return 0;
}
```

```
void minMax(int a[5][5], int *min,
            int *max)
{
    int i, j;
    /* add your code here */
}
```

Q: Using indexing?

Q: Using pointer?

minMax: Using Pointer Variable Approach

Using pointer variable:

```
void minMax(int a[5][5], int *min, int
*max)
{
    int i;
    int *p;
    p=*a;

    *max = *p;
    *min = *p;
    for (i=0; i<25; i++) {
        if ( *p > *max )
            *max = *p;
        else if ( *p < *min )
            *min = *p;
        p++;
    }
}
```

Point P towards a[0], *p = a[0][0]

a
p

Using pointer variable to process 2D arrays

```
main():
int A[5][5]= {
                {5, 10, 15, 20, 25},
                {10, 20, 30, 40, 50},
                {20, 40, 60, 80, 100},
                {1, 3, 5, 7, 9},
                {2, 4, 6, 8, 10}
};
```

Consecutive & sequential memory

5 | 10 | … | … | … | … | … | 6 | 8 | 10

p++

Sizeof Operand

- Gives the number of bytes

Sizeof Operator and Array: Example

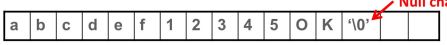
```
#include <stdio.h>
int sum(int  a[ ], int n);
int main(){
   int ar[6] = {1,2,3,4,5,6};
   int total;
   printf("Array size is %d\n",
       sizeof(ar)/sizeof(ar[0]));
   total = sum (ar, 6);
   return 0;
}
int sum ( int a[ ], int n ) {
   int i, total=0;
   printf("Size of a = %d\n", sizeof(a));
   for ( i=0;  i<n ;  i++)
        total += a[i];
   return total;
}
```

Output
Array size is **6**
(i.e. 24/4=6)
Size of a = **4**

Apply ***sizeof*** to a **pointer variable (e.g. a)** yields the size of the pointer.

String

- A **string** is **an array of characters** terminated by a **NULL** character ('\0').



- **String constant** is a set of characters in double quotes:
  e.g. **"C Programming"** - is an array of characters and automatically terminated with the **null** character **'\0'**
- Using #define to define a string constant:
  e.g. **#define NTU "Nanyang Technological University"**
- String constants can be used in function arguments, e.g. printf() and puts(): e.g. **printf("Hello, how are you?");**

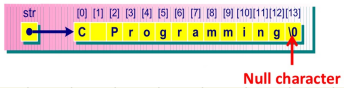
Note: Character Constant '**X**' vs String Constant "**X**":
- The character constant '**X**' consists of a single character of type **char**, while the character string constant "**X**" is an array of **char** that consists of two characters (i.e. the character '**X**' and the null character '**\0**').

```
char Str [] = "some text"
char Str [5] = "yesu"   "4 char + \0"
char Str[4]="yesu"  X   "4 char w/o \0"
char Str[] = {'1', '2', '3', '\0'}
```

(1) str = = &str[0]
(2) ***str** = = 'a'
(3) ***(str+1)** = = str[1] = = 'b'

String Variables: Declaration using Pointer Notation

- **String variable** can also be declared using the **pointer notation.**
- When declaring a string variable using the pointer notation, we can assign a **string constant** to a **pointer** that points to the data type char:
  e.g. char *str = "C Programming";
- When a **string constant** is assigned to a **pointer variable**, C compiler will:
  1. Allocate **memory space** to hold the string constant.
  2. Store the **starting address** of the string in the pointer variable.
  3. Terminate the string with **null** ('\0') character.



7

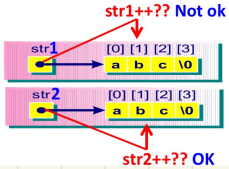
String Variables: Array vs Pointer Declaration

- As can be seen earlier, there are two ways to declare a string:

(1) char **str1[ ]** = "How are you?"; //with **array** notation
(2) char ***str2** = "How are you?"; //with **pointer** notation

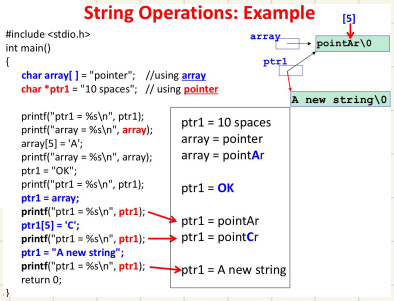
Q: What is the difference between the two declarations?
str1: **pointer constant**, **str2**: **pointer variable**.
Therefore,

```
++str1;      // not OK
++str2;      // OK
str1 = str2; // not OK
str2 = str1; // OK
```



★ Can manupilate String ptr but not String var

String Operations: Example



String Manupilation

- fgets( ) - puts( )
- scanf( ) - printf( )

String Input: fgets()

- **fgets() returns Null** if it fails, otherwise a pointer to the string is returned.
- Make sure **enough memory space** is allocated to hold the input string.

```
#include <stdio.h>
#include <string.h>
int main()
{
  char name[80], *p;   // allocate memory
  /*read name*/
  printf("Hi, what is your name?\n");
  fgets(name, 80, stdin);
  if ( p=strchr(name,'\n') )
    *p = '\0';   // replace '\n' character in name
  /*display name*/
  printf("Nice name, %s.\n", name);
  return 0;
}
```

name → Hui Siu Cheung\n '\0'

Output:
Hi, what is your name?
Hui Siu Cheung<\n> '\0'
Nice name, Hui Siu Cheung.

if: char *name;
Ok or not? Why?
name → **?? Not OK!**

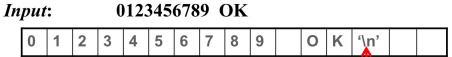
Only gives 4 bytes when char *name;

record 80 bytes

String Output: puts()

```
#include <stdio.h>
#include <string.h>
int main( )
{
   char str[80], *p;        // string with allocated memory
   printf("Enter a line of string: ");
   if (fgets(str, 80, stdin) == NULL) {
        printf("Error\n");
   }
   if ( p=strchr(str,'\n') )  *p = '\0';
   puts(str);
   return 0;
}
```

***Input*:** 0123456789 OK



***Output*:** 0123456789 OK '\0'

14

scanf() and printf(): Example

```
#include <stdio.h>
int main( )
{
   char name1[20], name2[20], name3[20];
   int count;
   printf("Please enter your strings.\n");
   count = scanf("%s %s %s", name1, name2, name3);
   printf("I read the %d strings: %s %s %s\n", count, name1,
         name2, name3);
   return 0;
}
```

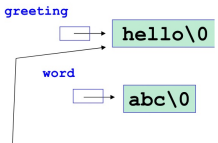
Output
Please enter your strings.
Hui Siu Cheung
I read the 3 strings: Hui Siu Cheung

Separated by space

String Processing – Using Indexes

```
#include <stdio.h>
int length1(char []);
int main( )
{
   char  *greeting = "hello", word[] = "abc";
   printf("The length is %d\n",
            length1(greeting),
   return 0;
}
```

Output
The length is 5



using index notation

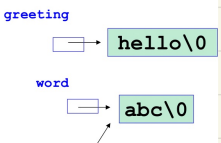
```
int length1(char string[])  // or int length1(char *string)
{
   int count = 0;
   while (string[count] != '\0')
      count++;
   return(count);
}
```

string

String Processing – Using Pointers

```
#include <stdio.h>
int length2(char *);
int main( )
{
   char  *greeting = "hello", word[] = "abc";
   printf("The length is %d\n",
            length2(word));
   return 0;
}
```

Output
The length is 3



using pointer notation

```
int length2(char *string)   // or int length2(char string[])
{
   int count = 0;
   while ( *(string+count) != '\0')
      count++;
   return(count);
}
```

string

String function

String Functions

- Must include the header file: **#include <string.h>**
- Some standard string functions are:

strcat()	appends one string to another
strncat()	appends a portion of a string to another string
strchr()	finds the first occurrence of a specified character in a string
strrchr()	finds the last occurrence of a specified characters in a string
strcmp()	compares two strings
strncmp()	compares two strings up to a specified number of characters
strcpy()	copies a string to an array
strncpy()	copies a portion of a string to an array
strcspn()	computes the length of a string that does not contain specified characters
strstr()	searches for a substring
strlen()	computes the length of a string
strpbrk()	finds the first occurrence of any specified characters in a string
strtok()	breaks a string into a sequence of tokens

The strlen() Function

- The function prototype of **strlen** is
  `size_t strlen(const char *str);`
  strlen computes and **returns the length of the string** pointed to by *str*, i.e. the number of characters that precede the terminating null character.
- Example:

```
#include <stdio.h>
#include <string.h>
int main()
{
   char line[81] = "This is a string";
   printf("The length of the string is %d.\n", strlen(line));
   return 0;
}
```

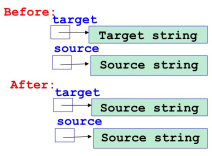
Output
The length of the string is 16.

The strcpy() Function

- The function prototype of **strcpy** is
  `char *strcpy(char *str1, const char *str2);`
  strcpy copies the string pointed to by str2 into the array pointed to by str1. It **returns** the value of **str1** (i.e. string).
- Example

```
#include <stdio.h>
#include <string.h>
int main(){
   char target[40] = "Target string";
   char *source = "Source string.";
   puts(target); puts(source);
   strcpy(target, source);
   puts(target); puts(source);
   return 0;
}
```

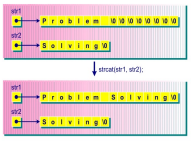
Output
Target string
Source string
Source string
Source string



The strcat() Function

- The function prototype of **strcat** is
  `char *strcat(char *str1, const char *str2);`
  strcat **appends** a copy of the string pointed to by str2 to the end of the string pointed to by str1. The initial character of str2 overwrites the null character at the end of str1. strcat **returns** the value of **str1** (i.e. string).
- **Example**

```
#include <stdio.h>
#include <string.h>
int main() {
   char str1[40] = "Problem ";
   char *str2 = "Solving";
   printf("The first string: %s\n", str1);
   printf("The second string: %s\n", str2);
   strcat(str1, str2);
   printf("The combined string: %s\n", str1);
   return 0;
}
```



Output
The first string: Problem
The second string: Solving
The combined string: Problem Solving

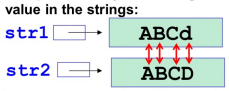
The strcmp() Function

- The function prototype of **strcmp** is
  `int strcmp(const char *str1, const char *str2);`
  strcmp compares the string pointed to by str1 to the string pointed to by str2.
- It **returns** an integer >, =, or < zero, accordingly if the string pointed to by str1 is >, =, or < the string pointed to by str2:
  - **0**: if the two strings are equal
  - **> 0 (the value could be the difference or 1 depending on system)**: if the first string follows the second string alphabetically, i.e. first string is larger (based on **ASCII values**)
  - **< 0 (the value could be the difference or -1 depending on system)**: if the first string comes first alphabetically, i.e. the first string is smaller (based on **ASCII values**)

The strcmp() Function: Example 1

```
#include <stdio.h>
#include <string.h>
int main()
{
   char str1[81], str2[81], *p;
   int result;
   printf("String Comparison:\n");
   printf("Enter the first string: ");
   fgets(str1, 81, stdin);
   if ( p=strchr(str1,'\n') ) *p = '\0';
   printf("Enter the second string: ");
   fgets(str2, 81, stdin);
   if ( p=strchr(str2,'\n') ) *p = '\0';
   result = strcmp(str1, str2);
   printf("The result of the comparison is
         %d\n\n", result);
   return 0;
}
```

Compare char by char using ASCII value in the strings:



Output
String Comparison:
Enter the first string: *ABCd* Larger
Enter the second string: *ABCD* Smaller
The result of the comparison is **1**

String Comparison:
Enter the first string: *A* Smaller
Enter the second string: *AF* Larger
The result of the comparison is **-1**

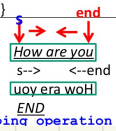
ABCd > ABCD
A < AF

Here, in this example, only 1, 0 or -1 is returned, it could also be the difference in ASCII values depending on the system.

The strcmp() Function: Example 2

```
/* Read a few lines from standard input &
write each line to standard output with
the characters reversed. The input
terminates with the line "END"*/
#include<stdio.h>
void reverse(char *);
int main()
{
   char line[132], *p;
   fgets(line, 132, stdin);
   if ( p=strchr(line,'\n') ) *p = '\0';
   while (strcmp(line, "END") !=0) {
      reverse(line);
      printf("%s\n", line);
      fgets(line, 132, stdin);
      if ( p=strchr(line,'\n') ) *p = '\0';
   }
}
```

```
void reverse(char *s)
{
   char c, *end;
   end = s + strlen(s) - 1;
   while (s < end) {
      /* 2 ends approaching centre */
      /* swapping operation */
      c = *s;
      *s++ = *end;   /*postfix op*/
            // i.e. *s = *end; s++;
      *end-- = c;
            // i.e. *end = c; end--;
   }
}
```

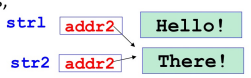


Swapping operation

Common Errors in Manipulating Strings

- When **copying** strings,

str1 = str2;



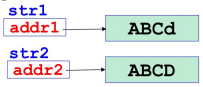
Changing Str2 will change Str1

Address is the same

is **incorrect**, we should use: **strcpy(str1, str2);**

- When **comparing** two strings,

if (str1 == str2) .....



Comparing the address

is **incorrect**, we should use
if (strcmp(str1,str2) == 0) ...

String Manupilation (U I)

Name	True If Argument is
Isalnum	Alphanumeric (alphabetic or numeric)
isalpha	Alphabetic
iscntrl	A control character, e.g. Control-B
isdigit	A digit
isgraph	Any printing character other than a space
islower	A lowercase character
isprint	A printing character
ispunct	A punctuation character (any printing character other than a space or an alphanumeric character)
isspace	A whitespace character: space, newline, formfeed, carriage return, etc.
isupper	An uppercase character
Isxdigt	A hexadecimal-digit character

ctype.h Functions

- These functions are used to test the nature of a character.
- Return **true (non-zero)** if the character belongs to a particular class, and return **false (zero)** otherwise.
- Must include the header file: **#include <ctype.h>**

ctype.h: Character Conversion Functions

- **toupper()** - converts lowercase character to uppercase;
  tolower() - converts uppercase character to lowercase;

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
void modify(char* str);
int main()  {
    char str[80], *p;            // allocate memory
    printf("Enter a string of text: \n");
    fgets(str,80,stdin); if ( p=strchr(str,'\n') ) *p = '\0';
    modify(str); puts(str);
    return 0;
}
void modify(char* str) {
    while (*str != '\0') {
        if (isupper(*str))
            *str = tolower(*str);
        else if (islower(*str))
            *str = toupper(*str);
        str++;
    }
}
```

Output

This is a test

t H...

tHIS IS A TEST

USER CHECKING

String to Number Conversions

- There are two ways to store a number. It can be stored as strings or in numeric form. Sometimes, it is convenient to read in the numerical data as a string and convert it into the numeric form. To do this, C provides the functions: **atoi()** and **atof()**.
- Must include the header file: **#include <stdlib.h>**

atof( )

- Prototype: *double **atof** (**const** char *ptr);*
- Functionality: converts the **string** pointed to by the pointer ***ptr*** into a double precision ***floating point number***.
- Return value: converted value.

atoi( )

- Prototype: *int **atoi** (**const** char *ptr);*
- Functionality: *converts* the **string** pointed to by the pointer ***ptr*** *into an **integer**.*
- Return value: converted value.

String to Number Conversions: Example

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
int main()
{
  char  ar[80];
  int  i,  num;

  scanf("%s", ar);          // read input string
  i=0;
  while (isdigit(ar[i]))    // check digit in string
     i++;                   // until not a digit
  if (ar[i] != '\0')        // if not a null character
     printf("The input is not a number\n");
     /* for example, "1a2" */
  else {
     num = atoi(ar);
     printf("Input is %d\n", num);
  }
}
```

ar → 1 2 3 \0 (indices [0][1][2][3])

Output

123

Input is 123

Note:

- **atof()** and **atoi()** are useful when the program reads in a string and then converts the string into the corresponding number representation for further processing.
- Sometimes it is more convenient to read in a string instead of reading in a number directly.

Formatted String I/O

The C standard I/O library provides two functions for performing formatted input and output **to strings**: **sscanf()** and **sprintf()**.

sscanf( )

- The function **sscanf()** is similar to scanf(). The only difference is that **sscanf()** takes input characters from a **string** instead of from the keyboard.
- The function **sscanf()** can be used to **transform numbers represented in strings**, e.g. the string "123" can be transformed into numbers 123 or 123.0 of data type int or double respectively.
- Function prototype: **sscanf(string_ptr, control-string, argument-list);**

sprintf( )

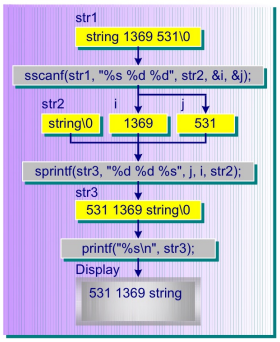
- The function **sprintf()** is similar to printf(). The only difference is that **sprintf()** prints output to a **string**.
- **sprintf()** can be used to **transform numbers into strings**.
- Function prototype:
  sprintf(string_ptr, control-string, argument-list);

Formatted String I/O - Example

```
#include <stdio.h>
#define  MAX_CHAR 80

int main()
{
    char  str1[MAX_CHAR] = "string 1369 531";
    char  str2[MAX_CHAR], str3[MAX_CHAR];
    int   i, j;

    sscanf(str1, "%s %d %d", str2, &i, &j);
    sprintf(str3, "%d %d %s", j, i, str2);
    printf("%s\n", str3);
    return 0;
}
```



Output

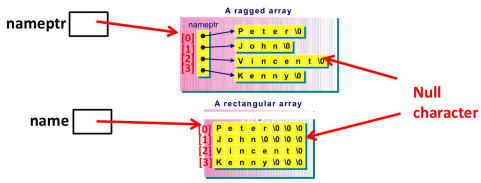
531 1369 string

Array of Character Strings

- **Arrays of Character Strings [declared as array of pointer variables]**

```
char *nameptr[4] = {"Peter", "John", "Vincent", "Kenny"};
```

nameptr is a ***ragged array***, an *array of pointers* (save storage)



- **Arrays of Character Strings [declared as 2-D arrays]**

```
char name[4][8]={"Peter","John","Vincent","Kenny"};
```

name is a ***rectangular array***.

Array of Character Strings: Example

```
#include <stdio.h>
int main()
{
   char *nameptr[4] = {"Peter", "John", "Vincent", "Kenny"};
   char name[4][10] = {"Peter", "John", "Vincent", "Kenny"};
   int  i, j;

   printf("Ragged Array: \n");          Using for loop
   for (i=0; i<4; i++)
     printf("nameptr[%d] = %s\n", i,
         nameptr[i]);

   printf("Rectangular Array: \n");
   for (j=0; j<4; j++)
     printf("name[%d] = %s\n", j,
         name[j]);
   return 0;
}
```

Output

Ragged Array:
nameptr[0] = Peter
nameptr[1] = John
nameptr[2] = Vincent
nameptr[3] = Kenny
Rectangular Array:
name[0] = Peter
name[1] = John
name[2] = Vincent
name[3] = Kenny

Structure

Defining a Structure Template

- A structure template is the master plan that describes how a structure is put together. To set up a structure template, e.g.

```
struct book {            /*template of book*/
  char title[40];
  char author[20];       /* members */
  float value;
};
```

  - struct: reserved keyword to introduce a structure
  - book: an optional tag name which follows the keyword struct to name the structure declared.
  - title, author, value: the member of the structure book.

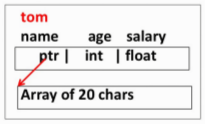
Note - The above declaration just declares a template, not a variable. No memory space is allocated.

Declaring Structure Variable: with Tag Name

- With tag name: separate the definition of structure template from the definition of structure variable.

```
struct person {
  char name[20];
  int age;
  float salary;
};

struct person tom, mary;
```



- With tag name – we can use the structure type subsequently in the program.

Declaring Structure Variable: without Tag Name

- Without tag name: combine the definition of structure template with that of structure variable.

```
struct {
  char name[20];
  int age;
  float salary;
} tom, mary;          /* no tag – person is not used */
```

- Without tag name – we cannot use the structure type elsewhere in the program.

Accessing Structure Members

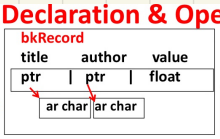
- The notation required to reference the members of a structure is

  structureVariableName.memberName

- The "." (dot notation) is a member access operator known as the member operator.
- For example, to access the member **age** of the variable **tom** from the struct person, we have **tom.age**.

Structure Declaration & Operation: Example

```
#include <stdio.h>
#include <string.h>
struct book {
    char title[40];
    char author[20];
    float value;
};
int main()
{
    char *p;
    struct book bkRecord;
    printf("Please enter the book title: \n");
    fgets(bkRecord.title, 40, stdin);   /* to access member, using . notation */
    if ( p=strchr(bkRecord.title,'\n') ) *p = '\0';
    printf("Please enter the author: \n");
    fgets(bkRecord.author, 20, stdin);
    if ( p=strchr(bkRecord.author,'\n') ) *p = '\0';
    printf("Please enter the value: \n");
    scanf("%f", &bkRecord.value);  /*note: & is needed here*/
    printf("%s by %s: $%.2f\n", bkRecord.title, bkRecord.author,
            bkRecord.value);
    return 0;
}
```



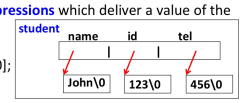
Variable name

```
Output
Please enter the book title:
C Programming
Please enter the author:
SC Hui
Please enter the value:
30.00
C Programming by SC Hui: $30.00
```

Structure Variable: Initialization

- Syntax for initializing structure variable is similar to that for initializing array variable.
- When there are insufficient values assigned to all members of the structure, remaining members are assigned with zero by default.
- Initialization of variables can only be performed with constant values or constant expressions which deliver a value of the required type.

```
struct personTag{
        char    name[20];
        char    id[20];
        char    tel[20];
}student = {"John", "123", "456"};
printf("%s %s %s\n", student.name, student.id,
        student.tel);
```



using . notation

```
Output
John 123 456
```

Structure Assignment

- The values in one structure can be assigned to another:

```
struct personTag newmember;
newmember = student;
```

- This has the effect of copying the entire contents of the structure variable **student** to the structure variable **newmember**. Each member of the **newmember** variable is assigned with the value of the corresponding member in the **student** variable.

Analogy (using primitive data type):

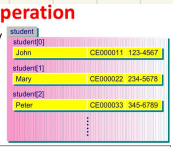
```
int num=10;
int member;
member = num;
```

Array of Struc ★

Arrays of Structures: Operation

```
/* Define a database with up to 10 student records */
struct personTag {
  char  name[40], id[20], tel[20];
};
struct personTag student[10] = {
    { "John", "CE000011", "123-4567"},
    { "Mary", "CE000022", "234-5678"},
        .....
};
int main( ) {
   int   i;
   for (i=0; i<10; i++)
      printf("Name: %s,  ID: %s, Tel: %s\n",
         student[i].name, student[i].id, student[i].tel);
}
```

using array index and . operator

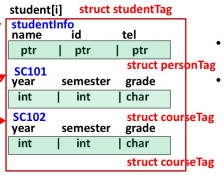



```
Output
Name: John ID: CE000011 Tel:
123-4567
Name: Mary ID: CE000022
Tel: 234-5678
```

Nested Structures

- Alternatively, struct studentTag can be defined in a more elegant manner using nested structures:

```
struct personTag {
        char    name[40];
        char    id[20];
        char    tel[20];
};
struct courseTag {
        int     year, semester;
        char    grade;
};
struct studentTag {
        struct personTag        studentInfo;
        struct courseTag        SC101, SC102;
};  // Nested structure
struct studentTag student[1000];
```



Nested Structures: Initialization

- In the program, after defining the nested structure **studentTag**, the array of structures variable **student** can be declared and initialized with initial data.
- The initialization is very similar to that of initializing multi-dimensional arrays.
- In the following example code:

```
/* Array variable initialization */
struct studentTag student[3] = {
    { {"John","CE000011","123-4567"},          // for student[0]
      {2002,1,'B'},
      {2002,1,'A'} },
    { {"Mary","CE000022","234-5678"},          // for student[1]
      {2002,1,'C'},
      {2002,1,'A'} },
    { {"Peter","CE000033","345-6789"},
      {2002,1,'B'},                            // for student[2]
      {2002,1,'A'} }
};
```

Nested Structures: Operation

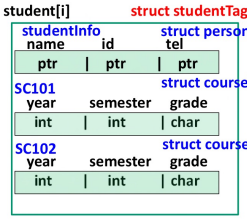
```
/* To print individual elements of the array of structures*/
int i;
for (i=0; i<=2; i++) {
      printf("Name:%s, ID: %s, Tel: %s\n",
      student[i].studentInfo.name,
      student[i].studentInfo.id,
      student[i].studentInfo.tel);

      printf("SC101 in year %d semester %d : %c\n",
      student[i].SC101.year,
      student[i].SC101.semester,
      student[i].SC101.grade);
      printf("SC102 in year %d semester %d : %c\n",
      student[i].SC102.year,
      student[i].SC102.semester,
      student[i].SC102.grade);
}
```

Note: Using dot (member operator) to access members of structures.

Nested Structures: Notations

- **student[i]** denotes the *i+1th* array record. It consists of three members: studentInfo, SC101, SC102.
- **student[i].studentInfo** denotes the personal information in the *i+1th* record. It consists of three members: name, id, tel.
- **student[i].studentInfo.name** denotes the student name in this record.
- **student[i].studentInfo.name[j]** denotes a single character value.
- **student[i].SC101, student[i].SC102** denote the course information in the *i+1th* record. Each consists of three members: year, semester, grade.

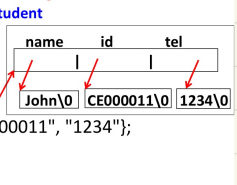


Pointers to Structures: Operation

```
/* Using pointers to structure */
struct personTag {
   char name[40], id[20], tel[20];
};
struct personTag student = {"John", "CE000011", "1234"};
struct personTag *ptr;
...
printf("%s %s %s\n", student.name, student.id, student.tel);
ptr = &student;

printf("%s %s %s\n", (*ptr).name, (*ptr).id, (*ptr).tel );

/* Why is the round brackets around *ptr needed?
  – op precedence */
```



Pointers to Structures: Operation

- To access a structure member via pointer, dereferencing is used as illustrated in the previous example:

  printf("%s %s %s\n", (*ptr).name, (*ptr).id, (*ptr).tel);

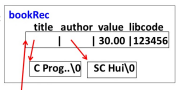
- Instead, we can use the **structure pointer operator (->)** for a pointer pointing to a structure:

  printf("%s %s %s\n", ptr->name, ptr->id, ptr->tel);

- Note that it is quite common to use the structure pointer operator (->) instead of the indirection operator (*) in pointers to structures.

Pointers to Structures: Example

```
#include <stdio.h>
struct book {
  char title[40];
  char author[20];
  float value;
  int libcode;
};
int main()
{
  struct book bookRec = {
    "C Programming", "SC Hui",
    30.00, 123456
  };
  struct book *ptr;
  ptr = &bookRec;
  printf("The book %s (%d) by %s: $%.2f.\n",
    ptr->title, ptr->libcode, ptr->author, ptr->value);
  return 0;
}
```



Output
The book C Programming (123456) by SC Hui: $30.00.

Struc to Function

(1) Passing Structure Members as Argument

```
#include <stdio.h>
float sum(float, float);
struct account {
   char  bank[20];
   float current;
   float saving;
};
int main( )
{
  struct account john={"OCBC Bank",1000.43, 4000.87};
  printf("The account has a total of %.2f.\n",
        sum(john.current, john.saving));     // pass by value
  return 0;
}
float sum(float x, float y)
{
  return (x+y);
}
```

Output
The account has a total of 5001.30.

- Using call by value
- struct members are used as arguments

(2) Passing Structure as Argument: Call by Value

```
#include <stdio.h>
struct account{
   char  bank[20];
   float current;
   float saving;
};
float sum(struct account);          /* argument - structure */
int main( )
{
   struct account john = {"OCBC Bank", 1000.43, 4000.87};
   printf("The account has a total of %.2f.\n", sum(john));  // pass by value
   return 0;
}
float sum( struct account money)
{
   return(money.current + money.saving);
   /* not money->current */
}
```

Output
The account has a total of 5001.30.

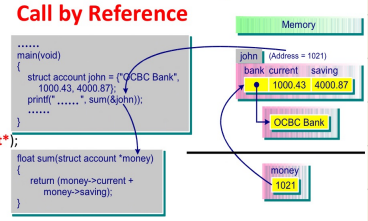
- Call by value
- struct account money is used as parameter

(3) Pointer

Passing Structure Address as Argument: Call by Reference

```
#include <stdio.h>
struct account{
   char  bank[20];
   float current;
   float saving;
};
float sum(struct account*);

int main()
{
   struct account john={"OCBC Bank",1000.43, 4000.87};
   printf("The account has a total of %.2f.\n",
         sum(&john));
   return 0;
}
float sum(struct account *money){
   return( money->current + money->saving);
}
```



- Call by reference

(4) Input into Struc

Passing by Returning a Structure

```
#include <stdio.h>
#include <string.h>
struct nameTag {
  char fname[20], lname[20];
};
struct nameTag getname();
int main()
{
  struct nameTag name;
  name = getname();
  printf("Your name is %s %s\n", name.fname, name.lname);
  return 0;
}
struct nameTag getname () {
  struct nameTag newname;
  char *p;
  printf("Enter first name: ");
  fgets(newname.fname, 20, stdin);
  if (p=strchr(newname.fname,'\n')) *p = '\0';
  printf("Enter last name: ");
  fgets(newname.lname, 20, stdin);
  if (p=strchr(newname.lname,'\n')) *p = '\0';
  return newname;
}
```

Output
Enter first name: *Siu Cheung*
Enter last name: *Hui*
Your name is Siu Cheung Hui

- Call by value (mainly)
- Returning the structure to the calling function
- Similar to returning a variable value in basic data type

The typedef Construct

- **typedef** provides an elegant way in structure declaration. For example, after defining the structure template:

```
struct date { int day, month, year; };
```

- We can define a **new data type** **Date** as

```
typedef struct date Date;
```

We use typedef to define a new data type Date with the structure template.

- Then, variables can be declared either as

```
struct date   today, yesterday;   or
Date          today, yesterday;
```

- Alternatively, when **typedef** is used, **tag name** is **redundant**, thus:

```
typedef struct {
        int  day,month,year;
} Date;
Date  today, yesterday;
```

No tag name – date

Define variables

The typedef Construct: Example

```
#define  CARRIER    1
#define  SUBMARINE    2
  typedef struct {
     int   shipClass;     char  *name;
     int   speed,crew;
  } warShip;
  void printShipReport(warShip);
  int main()   {
    warShip ship[2];          int i;
    ship[0].shipClass = CARRIER;
    ship[0].name = "Washington";
    ship[0].speed = 40;
    ship[0].crew = 800;
    ship[1].shipClass = SUBMARINE;
    ship[1].name = "Rogers";
    ship[1].speed = 100;
    ship[1].crew = 800;
    for (i=0; i<2; i++)
       printShipReport(ship[i]);
    return 0;   }
```

```
   /* Printing each record */
  void printShipReport(warShip ship)
  {
    if (ship.shipClass == CARRIER)
       printf("Carrier:\n");
    else
       printf("Submarine:\n");
    printf("\tname = %s\n", ship.name);
    printf("\tspeed = %d\n", ship.speed);
    printf("\tcrew = %d\n", ship.crew);
  }
```

Output

```
Carrier:
    name: Washington
    speed = 40
    crew = 800
Submarine:
    name = Rogers
    speed = 100
    crew = 800
```