

# CSC 4320/6320: Operating Systems

---



## C Review

Spring 2025

Instructor: Dr. Md Mahfuzur Rahman  
Department of Computer Science, GSU

---

# First C Program

---

```
#include <stdio.h>

int main(int argc, char *argv[]) {

    /* my first program in C */
    printf("Hello, World! \n");

    return 0;

}
```

# Structure of a C Program

---

```
/* File: powertab.c.
 *
 * This program generates a table comparing values
 * of the functions nA2 and 2An.
 */
#include <stdio.h>

/* Constants.
 *
 * LowerLimit - Starting value for the table
 * UpperLimit - Final value for the table
 */
#define LowerLimit 0
#define UpperLimit 12

/* Function prototypes*/
int RaiseintPower(int n, int k);
```

---

---

```
/* Main program */
int main()
{
    int n;

    printf("      2      N\n");
    printf("    N      N      2\n");
    printf("----+-----+-----\n");

    for (n = LowerLimit; n <= UpperLimit; n++) {
        printf(" %2d | %3d | %4d\n", n,
               RaiseintPower(n, 2),
               RaiseintPower(2, n) );
    }
}
```

---

```
/*  
 * Function: RaiseintPower  
 * This function returns n to the kth power.  
 */  
  
int RaiseintPower(int n, int k)  
{  
    int i, result;  
  
    result = 1;  
  
    for (i = 0; i < k; i++){  
        result *= n;  
    }  
    return (result);  
}
```

# Integer Types

---

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

# Float Types

---

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

# Formatted Input/Output

---

- Read input: *scanf* function
  - E.g. `scanf("%d", &x);`
- Print output: *printf* function
  - E.g. `printf("%d", x);`
- Both *printf* function and *scanf* function must be supplied with a *format string* as 1<sup>st</sup> argument.



# *printf* Function

---

## ■ Example:

```
int i, j;  
float x, y;
```

```
i = 10;  
j = 20;  
x = 43.2892f;  
y = 5527.0f;
```

```
printf("i = %d, j = %d, x = %f, y = %f\n", i, j,  
x, y+1);
```

Output:

```
i = 10, j = 20, x = 43.289200, y = 5528.000000
```

# *scanf* Function

---

## ■ Example

```
int i, j;
```

```
float x, y;
```

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

## ○ Sample input:

```
1 -20 .3 -4.0e3
```

`scanf` will assign 1, -20, 0.3, and -4000.0 to `i`, `j`, `x`, and `y`, respectively.

# Conversion characters

---

<u>character</u>	<u>type</u>
%c	char
%d	int
%f	float, double (for printf)
%If	double (for scanf)
%Lf	long double (for scanf)

# Other data types

---

The type void:

The type specifier void indicates that no value is available.

Derived types:

They include:

- Pointer types
- Array types
- Structure types
- Union types
- Function types.

# Declaring Pointer Variables

---

- Pointer variable must be preceded by an **asterisk**.
- Examples
  - `int *p;` /\* points only to integers \*/
  - `char *str;` /\* points only to characters \*/
  - `double *q;` /\* points only to doubles \*/
  - `int i , j , a[10], *p, *q ;` /\* p and q point to integers \*/

# Example

---

- Illustrates the relationship between pointers and arrays.

```
int SumintegerArray(int *ip, int n)
{
    int i, sum;

    sum = 0;
    for (i=0; i < n; i++) {
        sum+= *ip++;
    }
    return sum;
}
```

- Assume

```
int sum, list[S];
```

- Function call:

```
sum= SumintegerArray(list, 5);
```

# Example 1

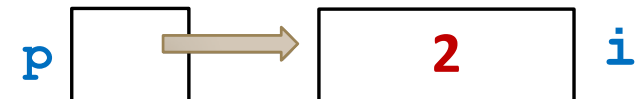
---

```
/* What does this program do? */  
int func (int n)  
{  
    int sum= 0;  
  
    while (n > 0) {  
        sum+= n % 10;  
        n /= 10;  
    }  
  
    return (sum);  
}
```

# The Address and Indirect Operators

- **& (address) operator** : get the address of a variable
- **\* (indirection) operator** : gain the access to the object that a pointer points to.
- Example:

```
int  i=2, j, *p;  /* p points no where */  
p = &i;    /* p points to integer variable i */  
j = *p; /* same as j=i */
```





# Pointer as Arguments

---

```
main() {  
    int x = 1, y = 2;  
    swap(&x, &y);  
    printf("x= %d , y= %d", x, y);  
}  
  
void swap(int *p, int *q) {  
    int temp=0;  
    temp = *p;  
    *p = *q;  
    *q = temp;  
}
```

Output:    **x=   2,   y   =1**

# Compare to “Value as Arguments”

---

```
main() {  
    int x = 1, y = 2;  
    swap(x, y);  
    printf("x= %d , y= %d", x, y);  
}  
  
void swap(int p, int q) {  
    int temp=0;  
    temp = p;  
    p = q;  
    q = temp;  
}
```

Output:    **x= 1, y =2**



# Pass by value or pass by pointer

---

- Useful read:

<https://denniskubes.com/2012/08/20/is-c-pass-by-value-or-reference//>

# Pointer as Return Values

---

- Return the location of an answer instead of returning its value.

```
main() {  
    int *p, x=2, y=5;  
    p = max(&x, &y);  
}  
  
int *max(int *a, int *b)  
{  
    if(*a > *b)  
        return a;  
    else  
        return b;  
}
```

**If  $x > y$ , return the address of  $x$   
otherwise, return the address of  $y$**

# Arrays

---

## ■ Declare an array

e.g. `int a[8];`

**In Java we should write**

`int[] a=new int[8];`

e.g. `#define N 10`

.....

`int a[N];`

**N will be replaced by 10 when  
program is executed**

e.g. `int m[5][9]; // 5 rows 9 columns`



# Arrays

---

- Note: Be careful for its indexing
  - C doesn't require that **subscript bounds** be checked.
  - If a subscript goes out of range, the program's behavior is undefined.
    - No compiling error but the result may be unexpected.

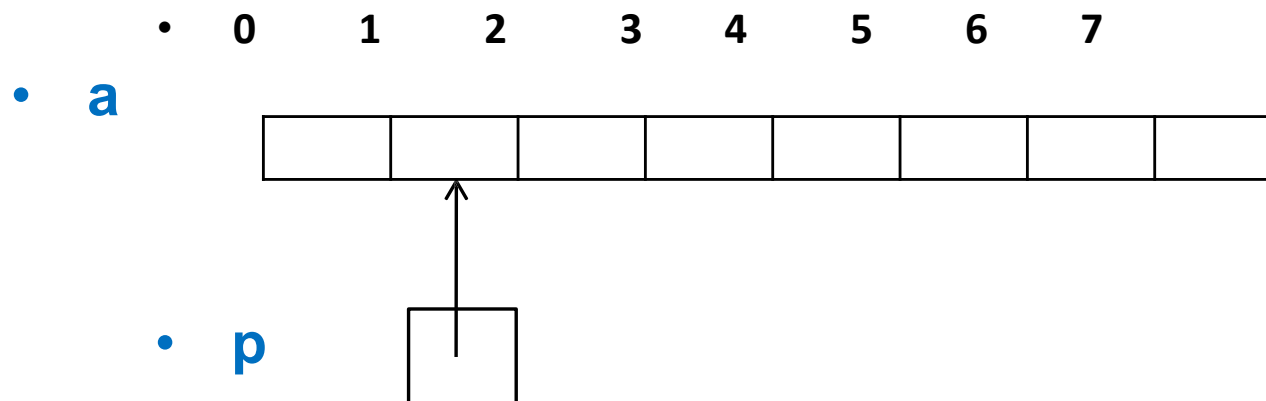
```
int a[10], i;  
for(i = 1; i <= 10; i++)  
    a[i] = 0;
```

**Note: actually `a[10]` does not exist.**  
**With some compilers, this for**  
**statement causes an infinite loop!**

# Pointer and Array

---

- Pointers can point to array elements, not just ordinary variables.
- Example
  - `int a[8], *p;`
  - `p = &a[1];`



# Function Declarations

*return-type* ***function-name*** (*parameters*) ;

```
double average(double x, double y); Declaration
```

```
int main()  
{
```

```
.....
```

```
average(x, y);
```

```
}
```

Or you can skip the name of parameters in declaration, and just write  
**double average (double, double)**

Function call

```
double average(double x, double y)  
{  
.....  
}
```

Definition



# String

---

- Define a string using pointer

```
char    *str="HelloWorld";
```

- Print out a string

```
printf ("%s", str);
```

- Get input for a string

```
scanf ("%s", str);
```

# C String library

---

- Get the length of a string

```
int len = strlen("abc"); /* return 3 */
```

- Compare two strings

```
strcpy(str1, "abc");  
strcat(str2, "def");  
if(strcmp(str1, str2) < 0)  
    printf("less than\n");
```

# Structure

---

- C has no class concept, but C has structure.
- A **structure** is a collection of values(members), possibly of different types.

# Declaring Structure Variable

---

```
struct {  
    int number;  
    char name[NAME_LEN + 1];  
    int on_hand;  
} part1, part2;
```

- `struct{...}` specifies a type
- `part1` and `part2` are variables of that type
- `number`, `name` array and `on_hand` are members of this structure

# Structure Initialization

---

```
struct {  
    int number;  
    char name[NW4E_LEN+1];  
    int on_hand;  
} part1 = {528, "Disk drive", 10}, part2 =  
{914, "Printer cable", 5};
```

# Structure Type

---

- Declaring a structure tag

```
struct part {  
    int number;  
    char name[NAME_LEN + 1];  
    int on_hand;  
}
```

```
struct part part1, part2;
```

- Declaring a structure type

```
typedef struct {  
    int number;  
    char name[NAME_LEN + 1];  
    int on_hand;  
} Part;
```

```
Part part1, part2;
```

# Operations on Structures

---

- Use period '.' to access a member within a structure

- `printf("Part number : %d \n", part1.number);`

- `part1.number = 258;`  
`part1.on_hand++;`

`scanf("%d", &part1.on_hand); // &(part1.on_hand)`

# Structures as Arguments

- Pass by values

```
void print_part (struct part p)
{
    printf("Part number: %d\n", p.number);
}
```

- Pass by addresses

```
void update_part (struct part *p)
{
    p->number = 123;
}
```



For pointers to a structure,  
use '->' to access to the  
members instead of period '.'



# Structure Definitions

---

- **Structures can be nested. One representation of a rectangle is a pair of points that denote the diagonally opposite corners.**

```
struct rect {  
    struct point pt1;  
    struct point pt2;  
};
```

```
struct rect screen;
```

```
/* Print the pt1 field of screen*/  
printf("%d, %d", screen.pt1.x, screen.pt1.y);
```

```
/* Print the pt2 field of screen*/  
printf("%d, %d", screen.pt2.x, screen.pt2.y);
```

# Structures and Pointers

---

- Both `.` and `->` associate from left to right
- Consider

```
struct rect r, *rp = &r;
```

- The following 4 expressions are equivalent.

```
r.pt1.x
```

```
rp -> pt1.x
```

```
(r.pt1).x
```

```
(rp->pt1).x
```

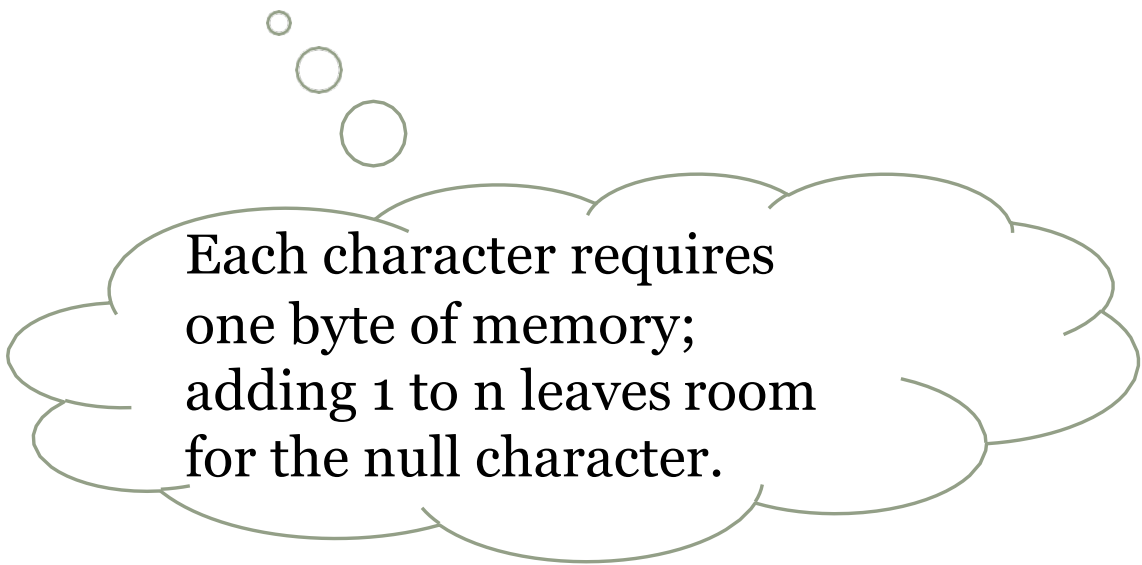
# Dynamic Memory Allocation

---

- Using **malloc** to allocate memory for a string

```
char *p; // p is not initialized
```

```
p = (char *) malloc(n + 1);
```



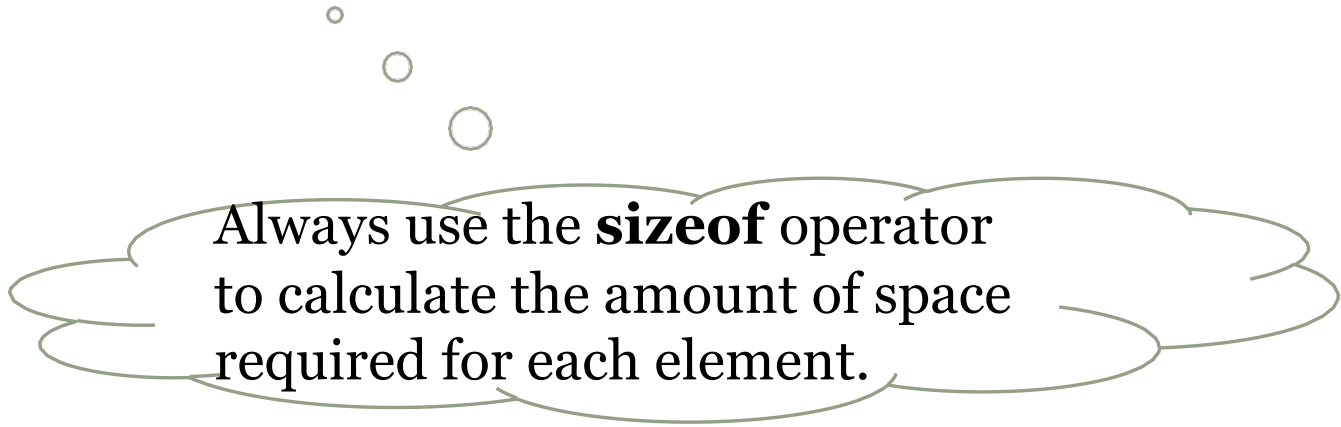
Each character requires  
one byte of memory;  
adding 1 to n leaves room  
for the null character.

# Dynamic Memory Allocation

---

- Using `malloc` to allocate storage for an array

```
int *a; // a is not initialized  
a = malloc(n * sizeof(int));
```



Always use the **sizeof** operator  
to calculate the amount of space  
required for each element.

# Dynamic Memory Allocation

---

- Using `malloc` to allocate storage for a structure

```
struct part {  
    int number;  
    char name[NAME_LEN + 1];  
    int on_hand;  
}
```

```
part *part1;  
part1=(struct part *) malloc ( sizeof(struct part) );
```

# Macro

(<https://gcc.gnu.org/onlinedocs/cpp/Macros.html>)

- A *macro* is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro. There are two kinds of macros.
- They differ mostly in what they look like when they are used.
  - *Object-like* macros resemble data objects when used,
  - *function-like* macros resemble function calls.

# Object-like macro

---

(<https://gcc.gnu.org/onlinedocs/cpp/Macros.html>)

- An object-like macro is a simple identifier which will be replaced by a code fragment. It is called object-like because it looks like a data object in code that uses it. They are most commonly used to give symbolic names to numeric constants.
- You create macros with the ‘#define’ directive. ‘#define’ is followed by the name of the macro and then the token sequence it should be an abbreviation for, which is variously referred to as the macro's body, expansion or replacement list. For example,
  - `#define BUFFER_SIZE 1024`
  - defines a macro named `BUFFER_SIZE` as an abbreviation for the token `1024`. If somewhere after this ‘#define’ directive there comes a C statement of the form
    - `foo = (char *) malloc (BUFFER_SIZE);`
  - then the C preprocessor will recognize and expand the macro `BUFFER_SIZE`. The C compiler will see the same tokens as it would if you had written
    - `foo = (char *) malloc (1024);`

# Function-like macro

(<https://gcc.gnu.org/onlinedocs/cpp/Macros.html>)

- You can also define macros whose use looks like a function call. These are called function-like macros. To define a function-like macro, you use the same ‘#define’ directive, but you put a pair of parentheses immediately after the macro name. For example,
  - `#define lang_init() c_init()`
  - `lang_init() ==> c_init()`
- Function-like macros can take arguments, just like true functions. As an example, here is a macro that computes the minimum of two numeric values, as it is defined in many C programs, and some uses. Below is a ternary operator.
  - `#define min(X, Y) ((X) < (Y) ? (X) : (Y))`
  - `x = min(a, b);        ==> x = ((a) < (b) ? (a) : (b));`
  - `y = min(1, 2);        ==> y = ((1) < (2) ? (1) : (2));`
  - `z = min(a + 28, *p);   ==> z = ((a + 28) < (*p) ? (a + 28) : (*p));`



# Linked List

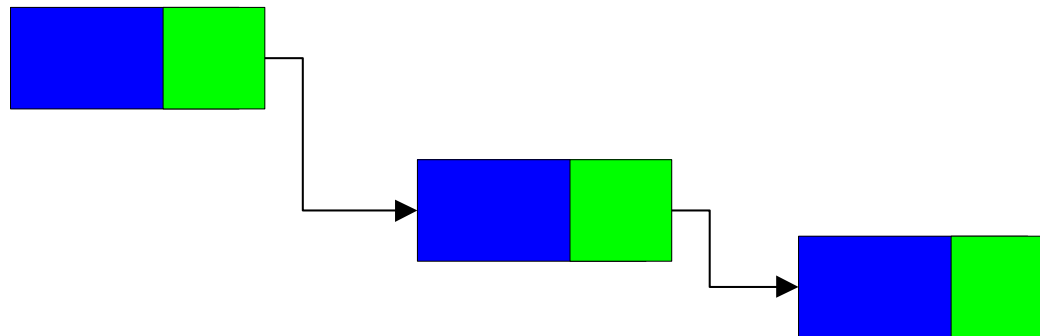
---

- A linked list is a basic data structure.
- For easier understanding divide the linked list into two parts.
  - **Nodes** make up linked lists.
  - Nodes are structures made up of **data** and a **pointer** to another node.
  - Usually the pointer is called *next*.

# Nodes

---

```
struct node  
{  
    struct rec    r;  
    struct node *next;  
};
```



# Creating a New Node

---

- Allocate space for a new node.
- Set the next pointer to the value of NULL
- Set the data value for the node.

# Adding Nodes to the List

---

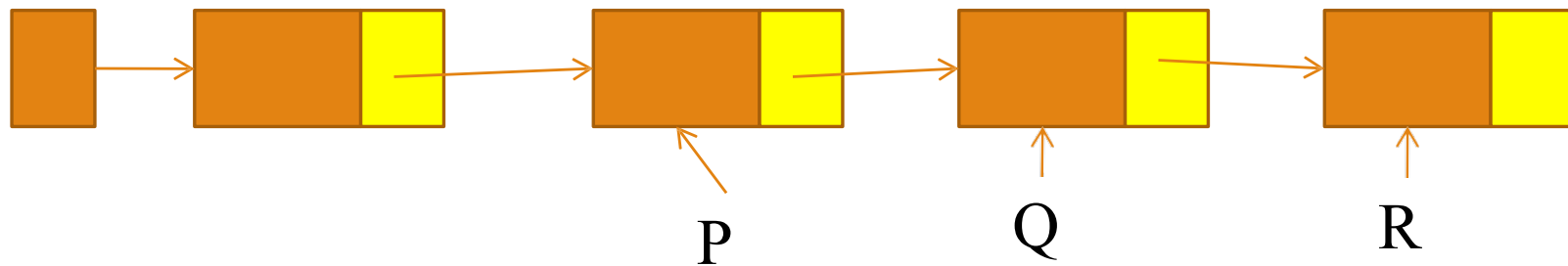
- Read the reference

<https://www.geeksforgeeks.org/linked-list-set-2-inserting-a-node/>

- Add node at the beginning
- Add node at the end
- Add node in the middle

# Deleting a node

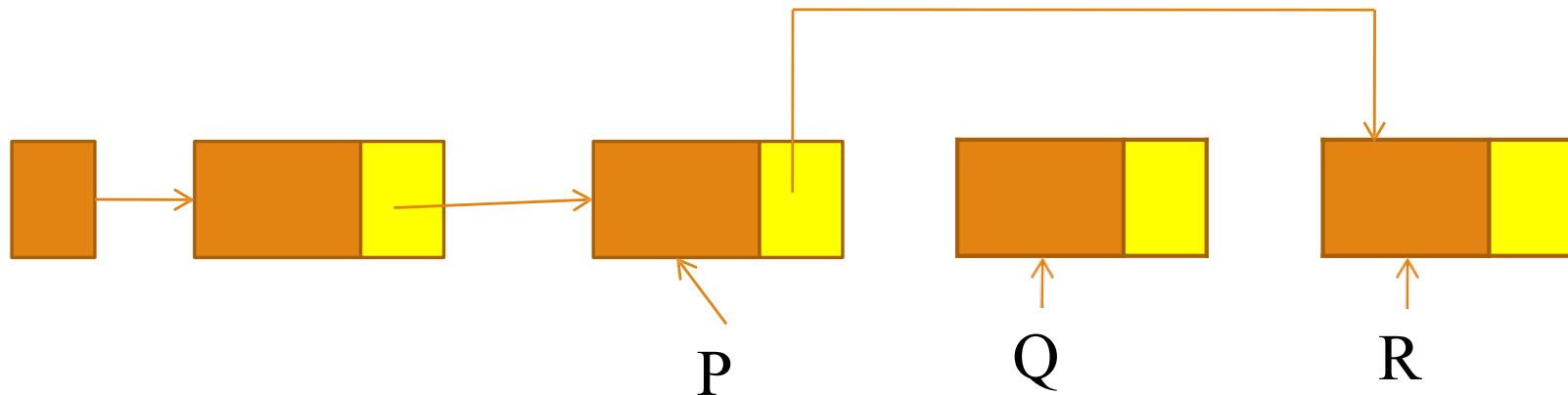
---



I want to delete Q

# Deleting a node

---



```
P->next = Q->next;  
Q->next = NULL;  
Free (Q) ;
```