# General Structure of a C Program

Eike Ritter
School of Computer Science
University of Birmingham

# General structure of a C program

```
#include <header file>
foo1(){ // User-defined function
   // Declaration of variables
   // Arithmetic and Logical expressions
}
main(){
   // Declaration of variables
   // Arithmetic and Logical expressions
   foo1(); // function call
   // More arithmetic and Logical expressions
}
```

- Program must contain a **main()** function
- Program execution starts from main()
- Header file contains pre-defined library functions

# Built-in data types in C

| C Basic Data Types | 32-bit CPU | | 64-bit CPU | |
|---|---|---|---|---|
| | Size (bytes) | Range | Size (bytes) | Range |
| char | 1 | -128 to 127 | 1 | -128 to 127 |
| short | 2 | -32,768 to 32,767 | 2 | -32,768 to 32,767 |
| int | 4 | -2,147,483,648 to 2,147,483,647 | 4 | -2,147,483,648 to 2,147,483,647 |
| long | 4 | -2,147,483,648 to 2,147,483,647 | 8 | - -9,223,372,036,854,775,808- 9,223,372,036,854,775,807 |
| long long | 8 | - 9,223,372,036,854,775,808- 9,223,372,036,854,775,807 | 8 | -9,223,372,036,854,775,808- 9,223,372,036,854,775,807 |
| float | 4 | 3.4E +/- 38 | 4 | 3.4E +/- 38 |
| double | 8 | 1.7E +/- 308 | 8 | 1.7E +/- 308 |

# Built-in data types in C

When we deal with **positive numbers only**, we can add **unsigned** before the type and get the range doubled.

```
int a;
// Range is -2,147,483,648 to 2,147,483,647

unsigned int b;
// Range is 0 to 4,294,967,295
```

# Immediate initialization of built-in variable

```
int i = 5, j =6;

char c = 'A', d;

float f = 2.333333333;

unsigned int n = 4294967295;
```

All variables except ‘d’ have been initialized during declaration.

Un-initialized variable contains a ‘garbage’ value initially.

# Constant data

- Constants can be declared as const

```
const float PI = 3.13;
```

- A constant is stored in the read-only segment of the memory
- Any effort to change a const variable will result in compilation error

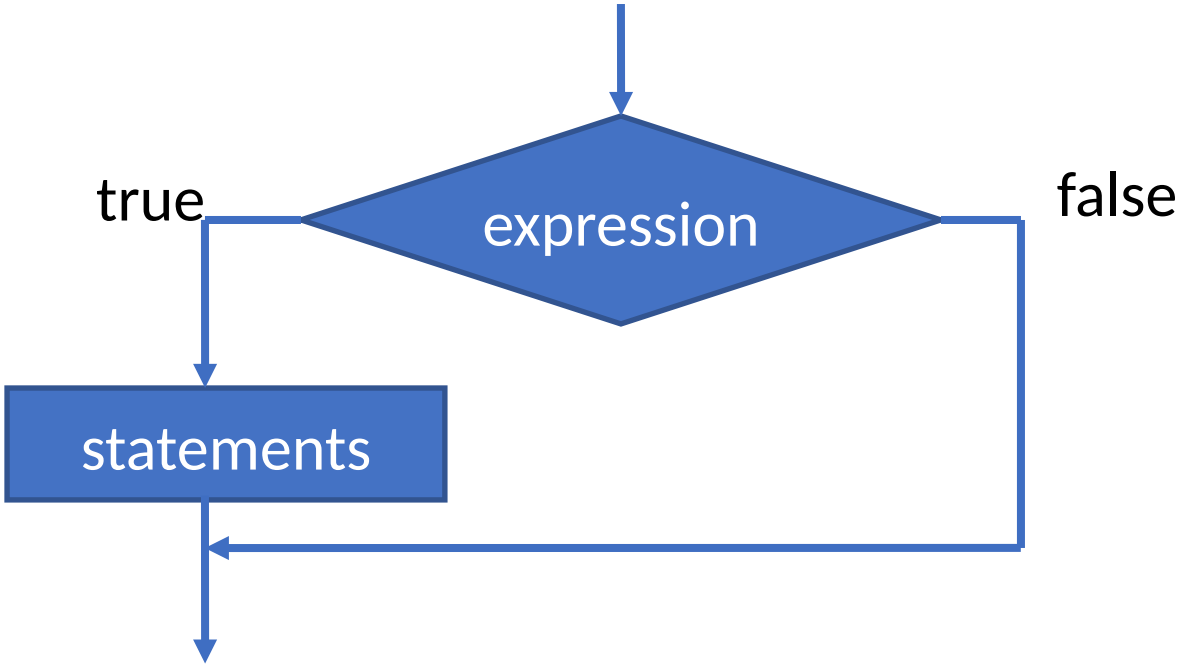
```
const float PI = 3.13;
// … Some code …
PI = PI + 1;
```

**There will be compilation error.**

# if Statement

if statement is for branching

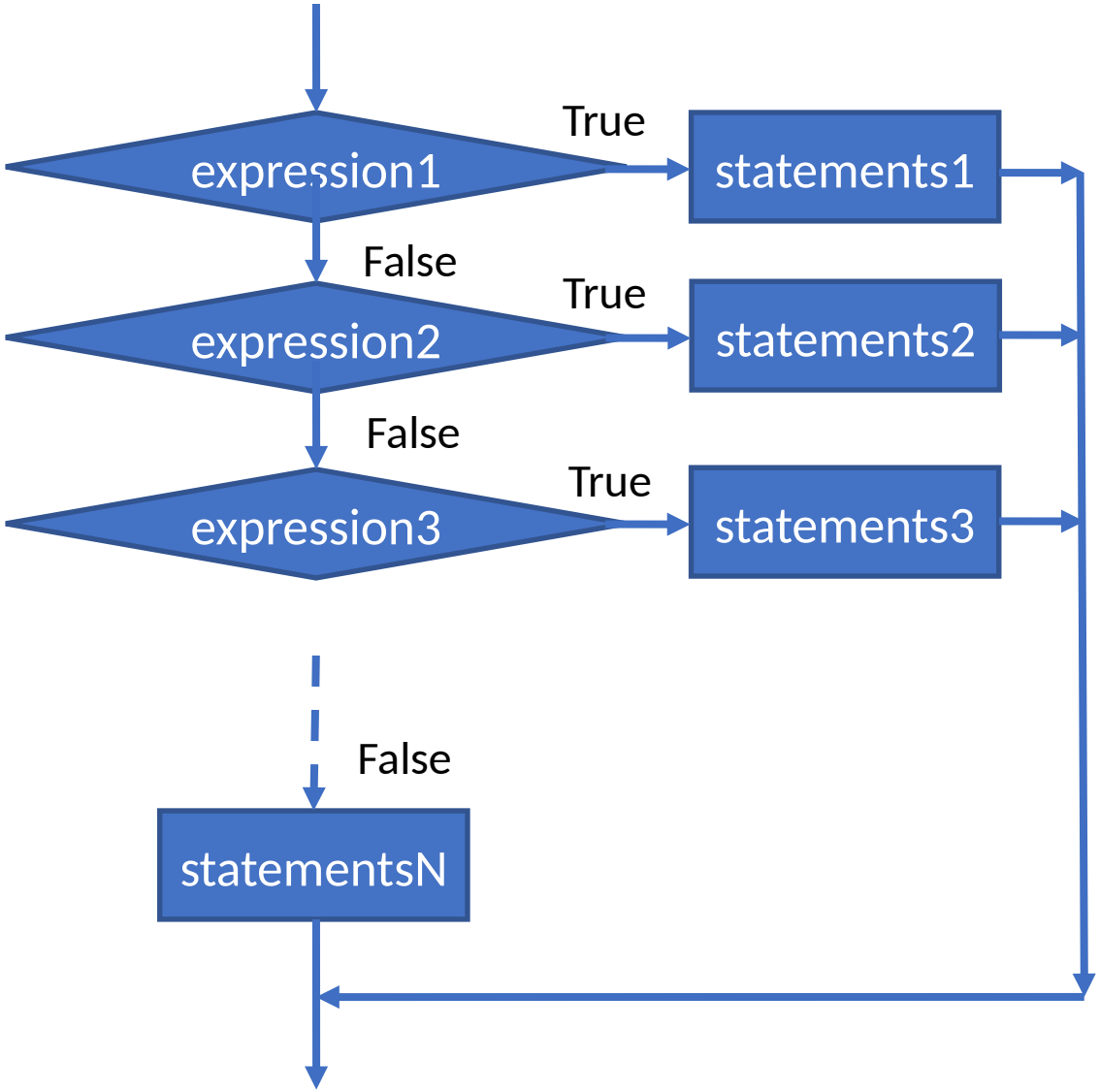
```
if (expression){
    statements
}
```



# if-else Statement

Multiple branching

```
if (expression1){
    statements1
}
else if (expression2){
    statements2
}
else if (expression3){
    statements3
}
...
else {
    statementsN
}
```



# if statement

```
if (value < 0) {
   sign = '-';
}
```

Used for conditional computation

# if-else statements

```
if (testscore >= 90) {
   grade = 'A';
}
else if (testscore >= 80) {
   grade = 'B';
}
else if (testscore >= 70) {
   grade = 'C';
}
else if (testscore >= 60) {
   grade = 'D';
}
else {
   grade = 'F';
}
```

Used for multiple branching

# switch Statement

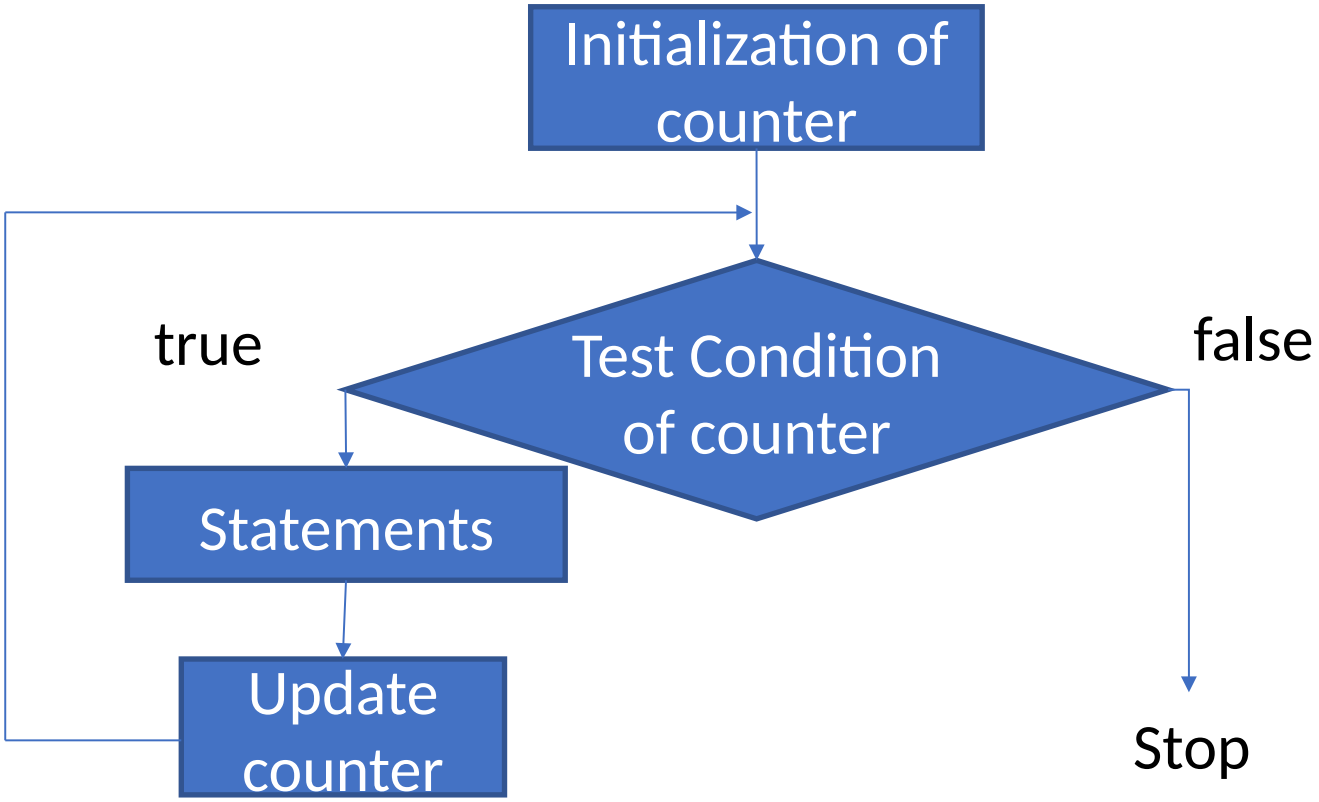
```
switch(expression){
  case constant-expression1:
    Code Block1;
    break;
  case constant-expression1:
    Code Block2;
    break;
  // ... More case blocks
  case constant-expressionN:
    Code BlockN;
    break;
  default:
    Code BlockDefault;
    break;
}
```

- *expression* is first evaluated
- It is compared with *constant-expression1, constant-expression2, …*
- If it matches anyone, then all statements inside that case are executed
- Statements in the **`default`** case are executed if no match is found

# for loop

Syntax is

```
for(init; condition test; increment or decrement)
{
   //Statements to be executed in loop
}
```



# for loop

Example: for-loop for computing the sum of N natural numbers.

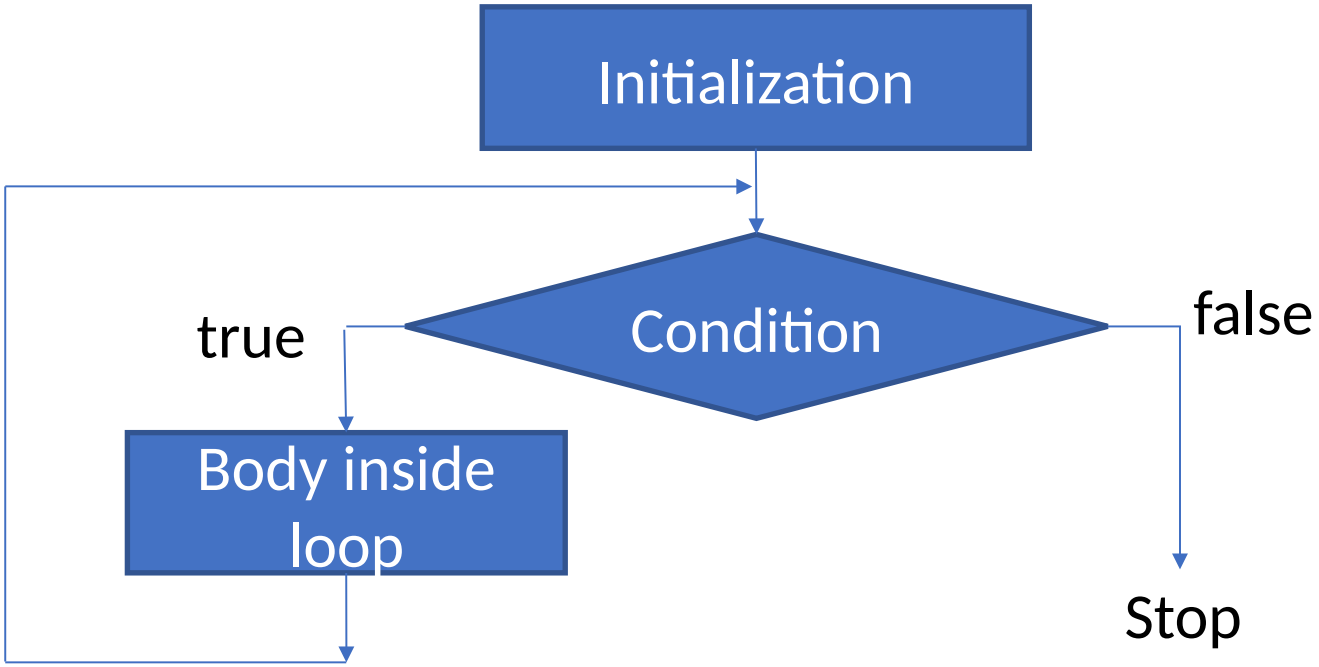
```
int sum=0, i;

for(i=1; i<=N; i++){
   sum = sum + i;
}
```

# while loop

Syntax is

```
while (condition test)
{
   // Statements to be executed in loop
   // Update 'condition'
}
```



# while loop

Example: while-loop for computing the sum of N natural numbers.

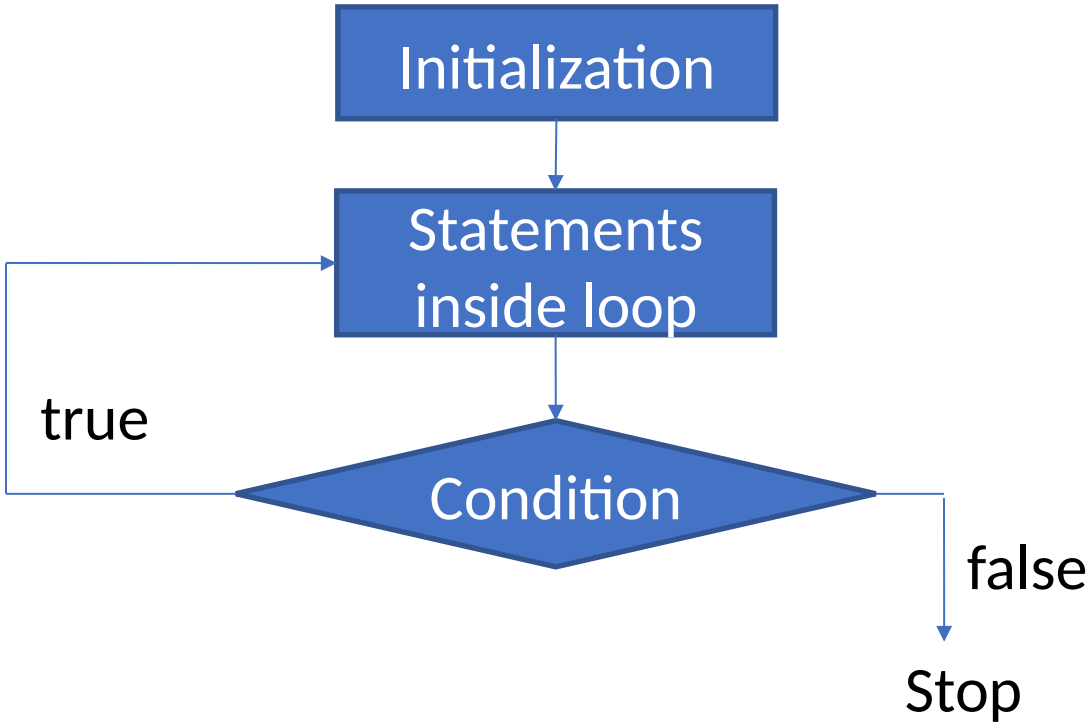
```
int sum=0, i=1;

while(i<=N){
   sum = sum + i;
   i++;
}
```

# do-while loop

Syntax is

```
do
{
   // Statements to be executed in loop
   // Update 'condition'
}while(condition test);
```



# do-while loop

Example: do-while-loop for computing the sum of N natural numbers.

```
int sum=0, i=1;

do{
   sum = sum + i;
   i++;
}while(i<N);
```

# continue statement

- `continue` is used inside a loop
- When a `continue` statement is encountered inside a loop, control skips the statements inside the loop for the current iteration
- and jumps to the beginning of the next iteration

```
sum = 0;
for(i=1; i<=5; i++){
   if(i==3)
      continue;
   sum = sum + i;
}
```

Computes 1 + 2 + 4 + 5 (addition of 3 is skipped)

# break statement

- **break** is used to come out of the loop instantly.
- After **break** control directly comes out of loop and the loop gets terminated.

```
sum = 0;
for(i=1; i<=5; i++){
   if(i==3)
      break;
   sum = sum + i;
}
```

Computes 1 + 2
Loop terminates at i=3

# Arrays in C

Array is a data structure

1. stores a fixed-size
2. sequential collection of elements
3. of the same type.

```
int   a[4] = {2, 5, 3, 7};
float b[3] = {2.34, 11.2, 0.12};
char c[8] = {'h','e','l','l','o'};
```

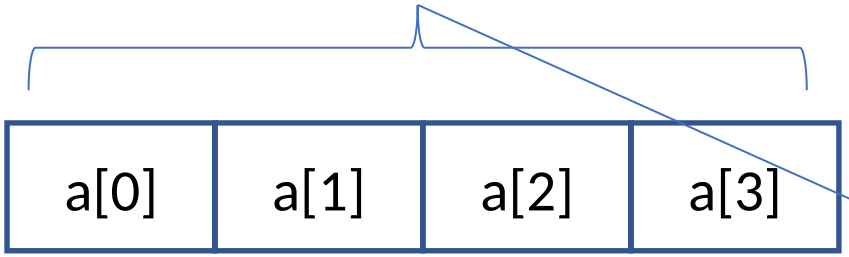
- Array `a[ ]` is of length 4 and contains only `int`

- Index of an array starts from 0, and then increments by 1
  - `a[0]` is the first element in `a[ ]` and it is 2
  - `a[3]` is the last element in `a[ ]` and it is 7

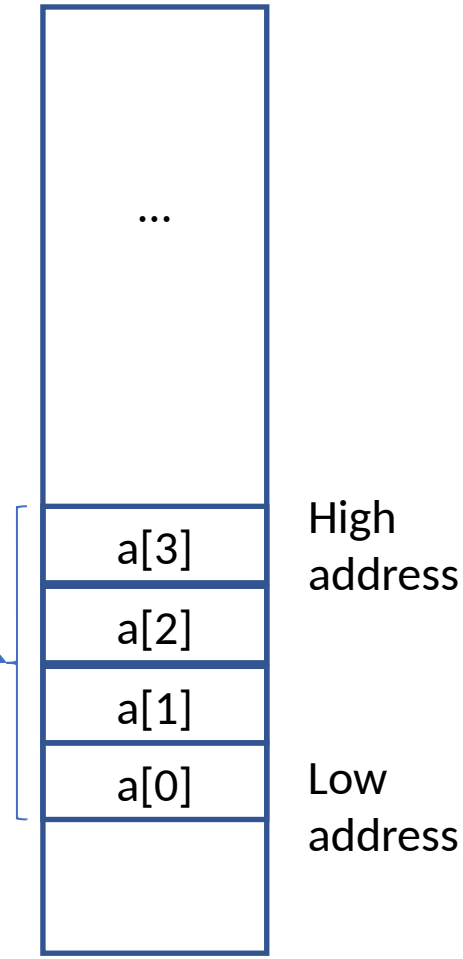
# Memory layout of a[ ]

Array is a data structure

1. stores a fixed-size
2. **sequential collection** of elements
3. of the same type.




Logical view of array a[4]




Memory layout

# Array and Loop

Example of computing the sum of an array

```
// Compute sum of an int array
int  a[4] = {2, 5, 3, 7};
int sum=0;

for(i=0; i<4; i++)
   sum = sum + a[i];
```

# Arrays in C: common mistakes(1)

Array is a data structure

1. stores a **fixed-size**
2. sequential collection of elements
3. of the same type.

```
int array_size; //user input
...
int a[array_size];
```

**This code causes malfunction.**

Array size must be a known constant.
Example: `int array_size = 4;`

# Arrays in C: common mistakes(2)

- C compiler does not check array limits

- If memory protection is violated, then the program crashes due to segmentation fault

```
// Compute sum of an int array
int  a[4] = {2, 5, 3, 7};
int sum=0;

for(i=0; i<500; i++) // Beyond array limit
   sum = sum + a[i];
```

Program crashes

# Two-Dimensional Array

```
// Declaration of array
// with 3 rows and 4 columns
int a[3][4];
```

Logical view of a[3][4]

| | | | |
|---|---|---|---|
| a[0][0] | a[0][1] | a[0][2] | a[0][3] |
| a[1][0] | a[1][1] | a[1][2] | a[1][3] |
| a[2][0] | a[2][1] | a[2][2] | a[2][3] |

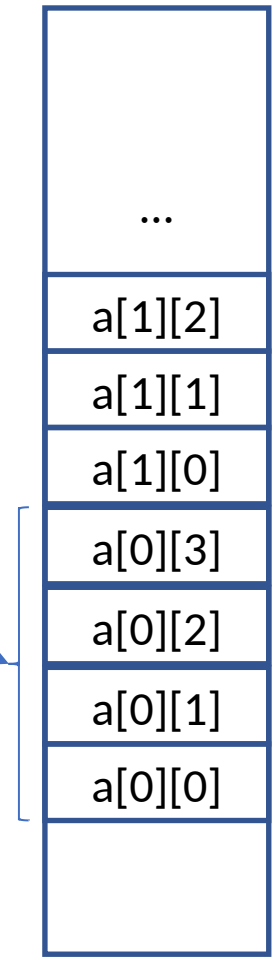
# Memory layout of two-dimensional array

C compiler stores 2D array in **row-major** order

- All elements of Row #0 are stored
- then all elements of Row #1 are stored
- and so on

| | | | |
|---|---|---|---|
| a[0][0] | a[0][1] | a[0][2] | a[0][3] |
| a[1][0] | a[1][1] | a[1][2] | a[1][3] |
| a[2][0] | a[2][1] | a[2][2] | a[2][3] |

Logical view of array a[3][4]




Memory layout

# Functions in C

- A function is a block of statements that together perform a task.

- Function definition in C has
  - a name,
  - a list of arguments (optional)
  - type of the value it returns (if any),
  - local variable declarations (if any), and
  - a sequence of statements (if any)

```
return_type function_name (argument list)
{
   // Local variables
      // Statements
}
```

# Example of function call

```
// max() function computes the maximum of two
// input integers and returns the maximum
int max(int num1, int num2){
  int temp;
  if(num1>num2)
    temp = num1;
  else
    temp = num2;
  return temp;
}

int main(){
  int a=5, b=11, c;
  c = max(a, b); // c gets the maximum of a and b
  // [some code here]
  return 0;
}
```

## scanf() function

scanf() function can be used to receive inputs from keyboard.

It is defined in `stdio.h` library and prototype is:

```
scanf("%format", &variable_name);
```

`format` is a place holder which depends on data-type.

```
Examples
scanf("%d", &var1); // var1 is an int
scanf("%c", &var2); // var2 is a char
scanf("%f", &var3); // var3 is a float

// To read var1, var2 and var3 together:
scanf("%d%c%f", &var1, &var2, &var3);
```

# Data types and formats

| Data Type | Format |
|---|---|
| signed char | %c |
| unsigned char | %c |
| short signed int | %d |
| short unsigned int | %u |
| signed int | %d |
| unsigned int | %u |
| long signed int | %ld |
| long unsigned int | %lu |
| float | %f |
| double | %lf |
| long double | %Lf |

# printf() function

printf() function can be used to print outputs.

It is defined in `stdio.h` library and prototype is:

```
printf("%format", variable_name);
```

`format` is a place holder which depends on data-type.

```
Examples
printf("%d", var1); // var1 is an int
printf("The value is = %d", var1);

// To print multiple variables together:
printf("%d %c %f", var1, var2, var3);
```

# Factorial program in Java

```
class Factorial {
 static int factorial(int n){
  if (n == 0)
    return 1;
  else
    return(n * factorial(n-1));
 }
 public static void main(String args[]){
  int i,fact=1;
  int number=4;
  fact = factorial(number);
  System.out.println("Factorial is: "+fact);

 }
}
```

We will port it to C

# Demo: Factorial program in C