# Task - Manage Contents Levels

# C Programming

## Compiler version

- MSVC
- GCC

## Language standard

- C++17 / C89/90/11 *github

- GNU11

- For Visual Studio 2019, the version information is given in

  ```
  Project\Properties\Configuration Properties\C/C++\Language\
  *C++ Language Standard
  ```

## References

- *programiz: introduction
- *tutorialspoint: introduction
- *geeksforgeeks: details
- *cprogramming: details

## Visual Studio

### Troubleshooting

- C1041 error: cannot open program database

  > Add `C:\Program Files (x86)\Microsoft Visual Studio\` to the anti-virus exception.

# Custom Notations

- Though these can be used inside code boxes, these are *irrelevant* to C syntax.

| notation | meaning |
|---|---|
| `<==>` | *equivalence* |
| `<=>` | the two items have the *same value*. |
| `<!=>` | the two item are different. |

# C Programming Basics

## Declaration / Definition

- *cprogramming *geeksforgeeks
- *Declaration* can be understood as a subset of *definition*.
- In the C program, no variable can be used without being declared.
- A variable or function can be declared multiple times but it can be defined only once. The following throws an error: '*x*': *redefinition; multiple initialization*.

```
int x = 1;
int x = 1; // error
```

### Function declaration

- For functions, the *function prototype* `int fcn();` declares the function `fcn`. Function declarations are typically organized in a header file.
- The following *defines* and thus *declares* the function `fcn`.

```
int fcn() {
}
```

### Global variable declaration

- Global variables are initialized to the value `0` if they are not otherwise assigned a value.

```
int x; // `x` is a global variable and initialized to 0.

int main() {
    int y; // `y` is only declared because it is not a global variable, but
a local variable.
    return 0;
}
```

  In the above code, `int x;` not only declares `x`, but also defines `x` because `x` is a global variable and hence a storage is allocated to the variable to store a value `0`. (simply speaking, `int x;` includes `x = 0;`.)

- To only declare a variable without defining, use `extern` keyword. Of course, the variable must be defined in another source file. *geeksforgeeks

```
extern int x;
```

- A global variable `int x` defined by `int x;` shares properties of declaration and definition. But it is not equivalent to `int x = 0;` because `int x;` can be written multiple times in a program. That is,

```
int x;
int x; // This is allowed.

int main(){
    return 0;
}
```

## Preprocessor

- *geeksforgeeks *cprogramming
- In a very basic term, preprocessor takes a C program and produces another C program without any `#`.
- **Directives**: In C and C++, the language supports a simple macro preprocessor. Source lines that should be handled by the preprocessor, such as `#define` and `#include` are referred to as *preprocessor directives*.
- The preprocessor is *not* a statement. That is, the statement terminator `;` i not used.

### `#include`

- `< >`: standard folder
- `" "`: current folder (directory)

### `#define` - Constants

- When we use define for a *constant*, the preprocessor produces a C program where the defined constant is searched and matching tokens are *replaced* with the given expression.

```
#define PI 3.141593

int main() {
    printf("%f\n", PI);
    return 0;
}
```

### `#define` - Macros

- *geeksforgeeks

- The macros are functions defined by `#define`.

- The arguments are not checked for data type.

- When a macro is used, the input tokens are treated as a set of strings. They are just replaced to the macro token placeholders.

  ```
  #define MULTIPLY(a, b) a*b
  #define MULTIPLY2(a, b) (a)*(b)

  int main() {
      printf("%d\n", MULTIPLY(1 + 2, 1 + 2)); // This gives 5.
      printf("%d\n", MULTIPLY2(1 + 2, 1 + 2)); // This gives 9.
      return 0;
  }
  ```

  To avoid such problems, enclose the function body with the parentheses as follows:

  ```
  #define MULTIPLY2(a, b) ((a)*(b))
  ```

- When the arguments do not have decimal points, the output is automatically assigned the `int` type.

  ```
  #define MULTIPLY(a, b) a*b

  int main() {
      printf("%d\n", MULTIPLY(1 + 2, 1 + 2)); // This gives 5.
      printf("%f\n", MULTIPLY(1 + 2, 1 + 2)); // This gives 0.000000.
      printf("%f\n", MULTIPLY(1 + 2.0, 1 + 2)); // This gives 5.000000.
      return 0;
  }
  ```

- The operator `##` is used to concatenate the tokens.

  ```
  #define MERGE(a,b) a##b

  int main() {
      printf("%d", MERGE(12, 34));
      return 0;
  }
  ```

- The operator `#` is used to convert the token passed to macro to a *string literal*.

  ```
  #define STRING(a) #a

  int main() {
      printf("%s", STRING(stringLiteral));
      return 0;
  }
  ```

- The macros can be written in multiple lines using `\`. The last line doesn’t need to have `\`.

- Preprocessors also support *if-else directives* which are typically used for conditional compilation.

```
#if MASS <= 0
    printf("Error.\n");
#endif
```

- There are some standard macros which can be used to print program file (`__FILE__`), Date of compilation (`__DATE__`), Time of compilation (`__TIME__`) and Line Number in C code (`__LINE__`)

## Semicolons `;`

- In a C program, the semicolon is a *statement terminator*. That is, each *individual statement* must be ended with a semicolon. It indicates the end of one logical entity.

## Tokens

- A token is the smallest element of a program that is meaningful to the compiler. Tokens can be classified as follows:
    - Keywords
    - Identifiers
    - Constants
    - Strings
    - Special Symbols
    - Operators

## Keywords

- *programiz

- Keywords are predefined, reserved words used in programming that have special meanings to the compiler. Keywords are part of the syntax and they cannot be used as an identifier.

- List of keywords *programiz

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

## Identifiers

- *programiz

- Identifier refers to name given to entities such as variables, functions, structures etc. Identifiers must be unique. They are created to give a unique name to an entity to identify it during the execution of the program. For instance, in `int n;`, `n` is an identifier.
- Identifier names must be different from keywords. `int` cannot be used as an identifier because it is a keyword.
- Rules:
  - The first letter of an identifier should be either a letter or an underscore.
  - The identifier must not exceed 31 characters, or it may cause problems in some compilers.

## Constants

- Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called *literals*.
- Two ways to define constants:
  - By using the keyword `const`

    ```
    const double PI = 3.14159;
    ```

  - By using `#define` *preprocessor* directive

## Special Symbols

| **Symbols** | | |
|---|---|---|
| Brackets | `[]` | Opening and closing brackets are used as array element reference. These indicate single and multidimensional subscripts. |
| Parentheses | `()` | These special symbols are used to indicate function calls and function parameters. |
| Braces | `{}` | These opening and ending curly braces marks the start and end of a block of code containing more than one executable statement. |
| Comma | `,` | It is used to separate more than one statements like for separating parameters in function calls. |
| Semicolon | `;` | It is an operator that essentially invokes something called an initialization list. |
| Asterisk | `*` | It is used to create pointer variable. |

assignment operator: It is used to assign values. pre processor |`#`| The preprocessor is a macro processor that is used automatically by the compiler to transform your program before actual compilation.

## Literals

- *programiz
- A literal is a value (or an identifier) whose value cannot be altered in a program. For example, `1`, `2.5`, `'c'`.

### **Integer** literals

- Example

```
Decimal (base 10): 0, -9, 22 etc
Octal (base 8): 021, 077, 033 etc
Hexadecimal (base 16): 0x7f, 0x2a, 0x521 etc
```

### **Floating-point** literals

- Example

```
-2.0
0.0000234
-0.22E-5
```

### Character literals

- A **character** literal is created by enclosing a *single* character inside *single* quotation marks. For example, `'a'`, `'m'`, `'F'`, `'2'`, `'}'`.

- Escape sequences

| Escape Sequences | Character |
|---|---|
| `\b` | Backspace |
| `\f` | Form feed |
| `\n` | Newline |
| `\r` | Return |
| `\t` | Horizontal tab |
| `\v` | Vertical tab |
| `\\` | Backslash |
| `\'` | Single quotation mark |
| `\"` | Double quotation mark |
| `\?` | Question mark |
| `\0` | Null character |

### String literals

- A **string** literal is a *sequence* of characters enclosed in *double*-quote marks.

```
"good"                  //string constant
""                      //null string constant
"      "                //string constant of six white space
"x"                     //string constant having a single character.
"Earth is round\n"      //prints string with a newline
```

## Variables

- *programiz
- A variable is a container (storage area) to hold data.

## Characters

- Printing a character with the `%s` format specifier gives a unintended result. Use `%c` to print a single character.

```c
char chr = 'a';
printf("%c\n", chr); // proper
printf("%s\n", chr); // unintended result
```

## Operators

### Arithmetic operators

- Operators

| Operator | Meaning of Operator |
|---|---|
| `+` | addition or unary plus |
| `-` | subtraction or unary minus |
| `*` | multiplication |
| `/` | division |
| `%` | remainder after division (modulo division) |

### Increment and decrement operators

- C programming has two operators increment `++` and decrement `--` to change the value of an operand (constant or variable) by `1`.

### Assignment operators

- An assignment operator is used for assigning a value to a variable. The most common assignment operator is `=`.

| Operator | Example | Same as |
|---|---|---|

| Operator | Example | Same as |
|---|---|---|
| `=` | `a = b` | `a = b` |
| `+=` | `a += b` | `a = a+b` |
| `-=` | `a -= b` | `a = a-b` |
| `*=` | `a *= b` | `a = a*b` |
| `/=` | `a /= b` | `a = a/b` |
| `%=` | `a %= b` | `a = a%b` |

### Relation operators

- A relational operator checks the relationship between two operands. If the relation is true, it returns `1`; if the relation is false, it returns value `0`.

| Operator | Meaning of Operator | Example |
|---|---|---|
| `==` | Equal to | `5 == 3` is evaluated to `0` |
| `>` | Greater than | `5 > 3` is evaluated to `1` |
| `<` | Less than | `5 < 3` is evaluated to `0` |
| `!=` | Not equal to | `5 != 3` is evaluated to `1` |
| `>=` | Greater than or equal to | `5 >= 3` is evaluated to `1` |
| `<=` | Less than or equal to | `5 <= 3` is evaluated to `0` |

### Logical operators

- Operators

| Operator | Meaning | Example |
|---|---|---|
| `&&` | Logical AND. True only if all operands are true | If `c = 5` and `d = 2` then, expression `((c==5) && (d>5))` equals to `0`. |
| `\|\|` | Logical OR. True only if either one operand is true | If `c = 5` and `d = 2` then, expression `((c==5)\|\| (d>5))` equals to `1`. |
| `!` | Logical NOT. True only if the operand is 0 | If `c = 5` then, expression `!(c==5)` equals to `0`. |

### Bitwise Operators

- During computation, mathematical operations like: addition, subtraction, multiplication, division, etc are converted to bit-level which makes processing faster and saves power.

| Operators | Meaning of operators |
|---|---|

| Operators | Meaning of operators |
|---|---|
| `&` | Bitwise AND |
| `\|` | Bitwise OR |
| `^` | Bitwise exclusive OR |
| `~` | Bitwise complement |
| `<<` | Shift left |
| `>>` | Shift right |

## Increment / decrement operators

- *wikipedia
- `i++` and `++i` are different operations.
  In languages that support both versions of the operators:
  - The *pre*-increment and *pre*-decrement operators increment (or decrement) their operand by 1, and the value of the expression is the *resulting* incremented (or decremented) value.
  - The *post*-increment and *post*-decrement operators increase (or decrease) the value of their operand by 1, but the value of the expression is the operand's *original* value prior to the increment (or decrement) operation.

## Data types

- *programiz *geeksforgeeks

- The typical size and string specifier of each type on **32 bit gcc compiler** *geeksforgeeks

| Type | Size (bytes) | Format Specifier |
|---|---|---|
| `char` | 1 | `%c` |
| `int` | 4 | `%d` |
| `float` | 4 | `%f` |
| `double` | 8 | `%lf` |
| `short int` | 2 | `%hd` |
| `unsigned int` | 4 | `%u` |
| `long int` | 4 | `%ld` |
| `long long int` | 8 | `%lld` |
| `unsigned long int` | 4 | `%lu` |
| `unsigned long long int` | 8 | `%llu` |
| `signed char` | 1 | `%c` |
| `unsigned char` | 1 | `%c` |

| Type | Size (bytes) | Format Specifier |
|---|---|---|
| `long double` | 12 | `%Lf` |

> In 32-bit and 64-bit operating systems, the Microsoft C compiler maps long double to type double. Therefore, `sizeof(long double)` yields 8 bytes in *MSVC*. *microsoft

- Variables can be declared at once.

```
int x1, x2;
```

- The size of `int` is 4 bytes (32 bits). It can take $2^{32}$ distinct states from `-2^31 (= -2147483648)` to `2^31-1 (= 2147483647)`.
- `float` and `double` are used to hold real numbers. The size of `float` (single precision float data type) is 4 bytes. And the size of `double` (double precision float data type) is 8 bytes.
- Keyword `char` is used for declaring character type variables.

```
char s = 'a';
```

  The size of `char` is 1 byte.

- `void` is an incomplete type. It means "nothing" or "no type". A variable can have `void` type. Only functions that return nothing can have `void` as its return type.
- `short` and `long` are equivalent to `short int` and `long int`, respectively.
- `signed` and `unsigned` are *type modifiers* that can change the data storage.

```
unsigned int x;
```

  The sizes of `int` and `unsigned int` are both 4 bytes but the values of `unsigned int` range from `0` to `2^31-1`.

## If-else statement

- `if-else` ladder template

```
if (x > 1) {
}
else if(y > 0) {
}
else if (a && b) {
}
.
```

```
.
else {
}
```

## `for` loop

- Syntax

```
for (initializationStatement; testExpression; updateStatement) {
    // statements
}
```

- Template

```
int i;
for (i = 0; i < 4; i++) {
    print("for: %d\n", i);
}
```

or

```
for (int i = 0; i < 4; i++) {
    printf("%d\n", i);
}
```

`i` in the latter template is a local variable, i.e., it cannot be referenced outside the `for` loop.

- In `for` loop argument `i++` and `++i` are equivalent because the counter `i` is incremented after the statements are executed.

## `while` loop

- Syntax

```
while (testExpression) {
    // statements
}
```

- Template

```
unsigned int i = 0;
while (i < 4) {
    printf("while: %d\n", i);
    i++;
}
```

- `do-while` loop

```
do {
// statements
}
while (testExpression);
```

  This executes the statement first regardless of the *test value*. Only then, the test expression comes into action afterwards.

## `break` / `continue`

- The break statement *ends the loop immediately* when it is encountered.

```
break;
```

- The continue statement *skips the current iteration* of the loop and continues with the next iteration.

```
continue;
```

## `switch` statement

- Syntax

```
switch (expression) {
    case constant1:
    // statements
    break;
    case constant2:
    // statements
    break;
    .
    .
    default:
    // default statements
    break;
}
```

- Template

```
num = 0;
switch (num) {
case 1:
    printf("switch: %d\n", num);
    break;
case 2:
    printf("switch: %d\n", num);
    break;
default:
    printf("switch: %d\n", num);
    break;
}
```

- Without `break;`, all statements after the matching label are executed.
- The `default` clause inside the `switch` statement is optional.

### `switch` and `break`

- Each `case` must be closed by `break;`.

```
case 1:
    // statements
    break;
```

## Functions

- A **function declaration** tells the compiler about a function's name, return type, and parameters. A **function definition** provides the actual body of the function.
- A function must be at least declared **above** the lines of the function call.

### Function prototype (declaration)

- A function prototype is simply the **declaration** of a function that specifies function's *name*, *parameters* and *return type*. It doesn't contain *function body*.
- Parameter names are not important in function declaration *only their type* is required.
- Syntax

```
returnType functionName(type1 argument1, type2 argument2, ...);
// or
returnType functionName(type1, type2);
```

- Example

```
int fcn(int a, int b);
// or
int fcn(int, int);
```

### Calling a function

- Syntax

```
functionName(argument1, argument2, ...);
```

### Function definition

- A function definition contains a function body as well as the function declaration.
- Syntax

```
returnType functionName(type1 argument1, type2 argument2, ...) {
    //body of the function
} // `;` is an option.
```

### Passing arguments

- In programming, *argument* refers to the variable passed to the function.
- If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function. Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit. *tutorialspoint
- Actual parameters vs. formal parameters *crasseux *csc

### Return statement

- Syntax

```
return (expression);
```

- The type of value *returned from the function* and the return type specified in *the function prototype and function definition* must match.

## Function Types

- When there's no return value, the return type of the function is set to `void`.

## Function call types

- *tutorialspoint

### Call by value

- This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
- By default, C uses *call by value* to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

### Call by reference

- This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

## Recursion

- A function that calls itself is known as a *recursive function*. And, this technique is known as *recursion*.

- In recursion, the body of a function contains a code calling the function.

- The recursion continues until the body of a function does not call itself. That is, the `return` statement does not contain the function evaluation.

- Example

```
int fibonacci(int i) {
    if (i == 0) {
        return 0;
    }
    if (i == 1) {
        return 1;
    }
    return fibonacci(i-1) + fibonacci(i-2);
}
```

## Scope

- *tutorialspoint
- A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language:
    - Inside a function or a block which is called local variables.
    - Outside of all functions which is called global variables.
    - In the definition of function parameters which are called formal parameters.

## Storage class

- Every variable in C programming has two properties: **type** and **storage class**. Type refers to the data type of a variable. And, storage class determines the scope, visibility and lifetime of a variable. *programiz

- Storage Classes are used to describe the features of a variable/function. These features basically include the *scope*, *visibility* and *life-time* which help us to trace the existence of a particular variable during the runtime of a program. *geeksforgeeks

| storage specifier | storage | intial value | scope | life |
|---|---|---|---|---|
| auto | stack | garbage | within block | end of block |
| extern | data segment | zero | global multiple files | till end of program |
| static | data segment | zero | within block | till end of program |
| register | CPU register | garbage | within block | end of block |

### Local variable

- The variables declared inside a block are *automatic* or *local variables*. (An automatic variable is a local variable.) The local variables exist only inside the block in which it is declared.
- There is no initial value of an auto variable (garbage).

### Global variable

- Variables that are declared outside of all functions are known as *external* or *global* variables. They are accessible from any function inside the program.
- The initial value of a global variable is *zero*.

### Register variable

### Static variable

- The value of a *static variable* persists until the end of the program. Once it is initialized, its value is stored throughout the program even in a new scope. *geeksforgeeks

  ```
  static int var;
  ```

- The initial value of a global variable is *zero*.

## Extern keyword

- `extern` keyword is used to use a *global variable* defined in another source file. By using `extern`, a variable shared in multiple source codes needs not be declared in a header file.
- Example
  A variable `int a` is defined only in *source2.c*. Then, *source1.c* can use the variable by declaring `int a` with `extern` in the `main()` block without explicitly declaring `int a` in a common header.

  - source1.c

```c
int main() {
    extern int a;
    printf("%d\n", a); // This prints '2'.
    return 0;
}
```

- source2.c

```c
int a = 2;
```

## Variable initialization (incomplete)

- *geeksforgeeks
- Global and static variables are initialized to `0`.
- Local variables must be initialized before it is used. For instance,

```c
int x;
printf("%d\n", x); // error
```

  throws an error in contrast with

```c
int x = 2;
printf("%d\n", x); // This prints 2.
```

- `for`-loops requires only declaration of a counter variable. (definition is redundant because a `for`-loop initializes the counter at the beginning.)

### Multiple local variable declaration

- Example:

```c
int x1, x2 = 2, arr[] = { 0, 1, 2, 3};
```

  `x1` is only declared, whereas `x2` and `arr` are defined.

## Array

### One dimensional array

- Declaration

```
dataType arrayName[arrSize];
```

`arrayName` has arrSize *elements*. The *index* starts from `0` and ends with `arrSize-1`.

- Initialization

```
int arr[5] = {0, 1, 2, 3, 4};
```

```
int arr1[] = {0, 1, 2, 3, 4}; // declaration without specifying the size
int arr2[]; // This throws an error because the length is not specified.
```

- Change value

```
arr[0] = 1;
```

### Multidimensional array

- Declaration

```
dataType arrayName[arrSize1][arrSize2]...;
```

- Initialization

```
int arr[2][3] = {{1, 3, 0}, {-1, 5, 9}}; // or
int arr[][3] = {{1, 3, 0}, {-1, 5, 9}}; // or
int arr[2][3] = {1, 3, 0, -1, 5, 9}; // or
int arr2[][2][2] = { {{0, 1}, {2, 3}}, {{4, 5}, {6, 7}} };
```

### Passing arrays to a function

- To pass arrays to a function, *only the name* of the array is passed to the function. Array names are *decayed to pointers* in the function.
  - One dimensional array: `[]` informs that the array is one dimensional.

  ```
  int fcn(int arr[]) { // optionally, arr[2]
      // `arr` is a pointer.
      printf("%d\n", arr[0]);
  }
  ```

```
int main() {
    int arr[] = {0, 1};
    fcn(arr); // This prints 0.
    // Only the name of the array is passed.
    return 0;
}
```

- Multidimensional array

```
int fcn(int arr[][2]) { // optionally, arr[2][2]
    // `arr` is a pointer.
    printf("%d\n", arr[1][1]);
}

int main() {
    int arr[][2] = { {0, 1}, {2, 3} };
    fcn(arr); // This prints 3.
    // Only the name of the array is passed.
    return 0;
}
```

## ASCII

- ASCII code table: *genuinecoder
- To find ASCII value of a character,

```
char chr = 'S';
printf("%d\n", chr);
```

## Strings

- A string is *a sequence of characters* terminated with a *null* character `'\0'`. The *string terminator* `'\0'` is automatically added at the end of the string whenever a sequence of characters are enclosed with the *double quotation marks* `""`. Therefore, the length of an array must be *larger* than the number of characters in a string to assign: *programiz

```
char str1[4] = "test"; // incorrect. There is no room for the null `\0`.
char str2[5] = "test"; // correct
char str3[6] = "test"; // correct, the exceeding element is assigned zero.
printf("str1 = %s\n", str1); // This prints `str2` with some garbage strings
added.
printf("str2 = %s\n", str2);
printf("str3 = %s\n", str3);
printf("str2[4] = %d\n", str2[4]); // This prints 0.
printf("str3[4] = %d\n", str3[4]); // This prints 0.
printf("str3[5] = %d\n", str3[5]); // This prints 0.
```

### String functions in *string.h*

- `strlen()` - calculates the length of a string
- `strcpy()` - copies a string to another
- `strcmp()` - compares two strings
- `strcat()` - concatenates two strings
- `strlwr()` - converts string to lowercase
- `strupr()` - converts string to uppercase

### String functions in *stdio.h*

- `gets()` - takes string input from user
- `puts()` - displays a string

## Pointers

### Declaration

- Example

```
int* p;
int *p; // preferred
int * p;
int* p, x; // `p` is a pointer and `x` is an integer.
```

### Assigning an address to a pointer

- `&` operator returns the address of the variable.

```
int* p, x = 5; // Valid.
p = &x;
```

```
int x = 5;
int* p = &x; // Valid. This is not `*p = &x;`, but `int *p; p = &x;`.
```

### To get the value

- `*` operator is used to get the value of the thing pointed by the pointers.

```
int* p, x, y;
x = 2;
p = &x;
printf("pointer: p = %p, &x = %p\n", p, &x);
printf("value: *p = %d, x = %d\n", *p, x);
```

- `*` is called the *dereference operator* when working with pointers. It operates on a pointer and gives the value stored in that pointer.

### To change the value

- Example

```
*p = 3;
```

### Pointer to an array

- `*arr[0]`, `arr[0]`, and `&arr[0]` are equivalent to `**arr`, `*arr`, and `arr`, respectively.
- Multidimensional array indexing
  For an `n`-dimensional array `arr`,
  - Value of array

    ```
    arr[ind1][ind2]...[ind_n]
    <==> *(*(...*(*(arr+ind1)+ind2)+...)+ind_n) // value of element
    ```

  - Address of the first element of array

    ```
    arr[ind1][ind2]...[ind_k] // k < n
    <==> &(*arr[ind1][ind2]...[ind_k])
    <==> &arr[ind1][ind2]...[ind_k][0]
    ```

    - `*arr[ind1][ind2]...[ind_k]` and `arr[ind1][ind2]...[ind_k][0]` are *equivalent*.

  - Index level

    `*arr[ind1][ind2]...[ind_k]`, `arr[ind1][ind2]...[ind_k]`, and `&arr[ind1][ind2]...[ind_k]` have the *same value* where `k` < `n` but they are *not equivalent*. They are different in *index level* where the index levels of the the three pointers are `ind_{k+2}`, `ind_{k+1}`, and `ind_k`, respectively.

    ```
    &arr[l] + n <==> (arr + l) + n <==> &arr[l+n] // line 1
    arr[l] + n <==> *(arr + l) + n <==> &arr[l][n] // line 2
    ```

```
*arr[l] + n <==> *(*(arr + l)) + n <==> &arr[l][0][n] // line 3
**arr[l] + n <==> ***(arr + l) + n <==> arr[l][0][0] + n // line 4
// If i!=0, all the three lines represent different values.
```

```
*(arr[l] + m) + n <==> *(*(arr + l) + m) + n <==> *(&arr[l][m]) + n
<==> arr[l][m] + n <==> &arr[l][m][0] + n <==> &arr[l][m][n]
```

- Example
  First of all, `<==>` indicates *equivalence* and `<=>` means that the two items have the *same value*. These are irrelevant to C syntax.
  For a three-dimensional array `arr`,

```
arr[1] <==> &arr[1][0] <=> *arr[1] <==> arr[1][0] // These are addresses.
arr[1]+2 <==> &arr[1][2] <=> arr[1][2] // These are addresses.
*(arr[1][2]+1) <=> arr[1][2][1] // Values. identical address, hence value
```

- Memorizing rules

  > - `&` and the last index `[n]` form a pair that they can be removed together to add an non-negative integer `n`. Equivalently, adding a non-negative integer `n` adds `&` and `[n]`. That is, `&arr[n] <==> arr + n`.
  > - Removing `*` adds `[n]` at the end of the array indexing: `*(arr + n) <==> arr[n]`.

  - Example

```
*(arr + 1) + 2 <==> arr[1] + 2 <==> &arr[1][2]
```

- Pointer operation
- *geeksforgeeks *stackoverflow *tutorialspoint

## Pointers as function arguments (call by reference)

- Example

```
int swap(int* var1, int* var2) {
    int temp = *var1;
    *var1 = *var2;
    *var2 = temp;
    return 0;
}
```

## Dynamic allocation of two dimensional array

- *geeksforgeeks *dyclassroom

### Double pointer

- `*(*p) <==> **p`
- *geeksforgeeks

## `NULL` pointer in C

- In C11 standard

> An integer constant expression with the value 0, or such an expression cast to type void *, is called a null pointer constant. If a null pointer constant is converted to a pointer type, the resulting pointer, called a null pointer, is guaranteed to compare unequal to a pointer to any object or function

- `NULL` is a constant defined by

```
#define NULL ((void *)0)
```

- Some of the most common use cases for NULL are *geeksforgeeks
  - To initialize a pointer variable when that pointer variable isn't assigned any valid memory address yet.
  - To check for a null pointer before accessing any pointer variable. By doing so, we can perform error handling in pointer related code, e.g., dereference pointer variable *only if* it's not `NULL`.
  - To pass a null pointer to a function argument when we don't want to pass any valid memory address.

## Structures

### Declaration / definition of structures

- Declaration of a structure

```
struct structureTag {
    type member1; // member declaration
    type member2;
    .
    .
}; // A storage is not allocated in the declaration.
```

- Several ways of defining a structure variable
  - Method 1 (data-type-like definition)

```c
struct data {
    int x;
    int arr[2];
}; // structure declaration

struct data data1; // global structure variable definition. A storage
is allocated.

int main() {
    struct data data2; // structure variable definition. A storage is
allocated.
    data2.x = 0; // initialization
    data2.arr[0] = 0;
    data2.arr[1] = 0;
    return 0;
}
```

The members of a *global* structure variable are initialized to `0`'s.

- Method 2 (declaration with definition)

```c
struct { // The structure tag is optional.
    int x;
    int arr[2];
} data1; // global structure declaration
// The members of data1 are initialized to 0.

int main() {
    return 0;
}
```

- Method 3 (definition with initialization, *designated initialization*)

```c
struct data {
    int x;
    int arr[2];
};

int main() {
    struct data data1 = { 0, {1, 2} }; // definition and initialization
    struct data data2 = { .x = 0, .arr = {1, 2} }; // designated
initialization
    struct data data3 = { .x = -1 }; // designated partial
initialization
    return 0;
}
```

The initialization of a structure variable is only possible during the definition.

#### Member access

- A dot operator `.` is used to access the members of a structure variable.
- A arrow operator `->` is used to access the members of a structure variable through a *pointer*.

#### Array of structures

- Like other primitive data types, an array of structures can be created.

```c
struct data {
    int x;
    int arr[2];
};

int main() {
    struct data data1[4]; // definition of an array of structures
    data1[0].x = 2;
    data1[0].arr[0] = 3;
    data1[1].x = 4;
    data1[1].arr[1] = 5;
    return 0;
}
```

#### Pointer of a structure

- Example

```c
struct data {
    int x;
    int arr[2];
};

struct data *ptr, data1, data2[2] = { {.x = -1}, {.x = 1, .arr = {2, 3}} };

int main() {
    ptr = &data1;
    printf("data1: %d, %d\n", ptr->x, ptr->arr[0]); // member access
    printf("data2: %d, %d, %d\n", \
        data2->x, (data2 + 1)->arr[0], *((data2 + 1)->arr + 1));
    return 0;
}
```

### Structure member alignment / padding / data packing

- *geeksforgeeks

### `printf` format string (% placeholder)

- *cplusplus *wikipedia

### Storage class specifier

- *microsoft
- auto, register, static, extern, typedef

### C Type Specifiers

- *microsoft

### Header

- A header file is a file with extension .h which contains C function *declarations* and *macro definitions* to be shared between several source files.
- Do not *define* variables or functions in a header file. Do only *declaration*.

### Underscores in C

# C Programming Tips

## `sizeof`

- `sizeof` operator can be used to check the size of a variable in *byte*.
- For arrays, `sizeof` returns the sum of all elements in byte. For example,

  ```
  int arr[] = {0, 1, 2, 3};
  printf("%d\n", sizeof(arr)); // This prints 16. (the size of int is 4.)
  ```

- To calculate the length of an array,

  ```
  int arr[] = {0, 1, 2, 3};
  int length = sizeof(arr)/sizeof(arr[0]);
  ```

  However, when an array is put into a function as input, the length of the array cannot be calculated inside the function because the formal parameters corresponding to an array is a pointer. To get the length, the size of the array must be passed separately. Refer to *stackoverflow.

  ```
  int length(int arr[], int arrSize) {
      int len = arrSize / sizeof(arr[0]);
      return len;
  }
  ```

```
int main() {
    int arr[] = {0, 1, 2, 3};
    int len = length(arr, sizeof(arr));
    printf("length(arr4) = %d\n", len); // This prints 4
    return 0;
}
```

### `scanf`

- Template

  ```
  double value1;
  int value2;
  scanf_s("%lf%d", &value1, &value2);
  printf("%lf, %d\n", value1, value2);
  ```

  Use `scanf_s` in MSVC.

- Reading a string with scanf *stackoverflow.

### When to use `;`

#### Mandatory (dec = declaration, def = definition)

- `struct` dec/def, function dec

#### Optional

- function def

### Header inclusion

- A header file may be included more than one time directly or indirectly, this leads to problems of redeclaration of same variables/functions. To avoid this problem, *directives* like `#defined`, `#ifdef`, and `#ifndef` are used.

### The uses of `void`

- For `main()` function, *geeksforgeeks
- For user-defined functions, *stackoverflow