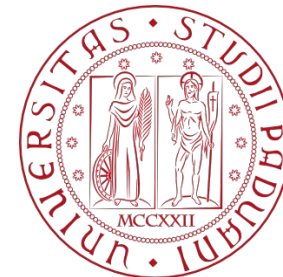


# COMPUTER ENGINEERING LABORATORY

**Luigi Rizzo**

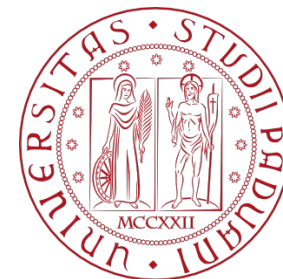
[luigi.rizzo@unipd.it](mailto:luigi.rizzo@unipd.it)

October 2023-January 2024



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

# Miscellaneous



UNIVERSITÀ  
DEGLI STUDI  
DI PADOVA

- Switch-case statement
- Ternary statement
- Goto and labels
- Insights
  - C Preprocessor, Pointers, Arrays, Structures
  - Miscellaneous Functions

C has the following conditional statements:

- Use `if` to specify a block of code to be executed, if a specified condition is true
- Use `else` to specify a block of code to be executed, if the same condition is false
- Use `else if` to specify a new condition to test, if the first condition is false
- **Use `switch` to specify many alternative blocks of code to be executed**
- **Use the ternary operator as a shorthand way to write an if-else statement**

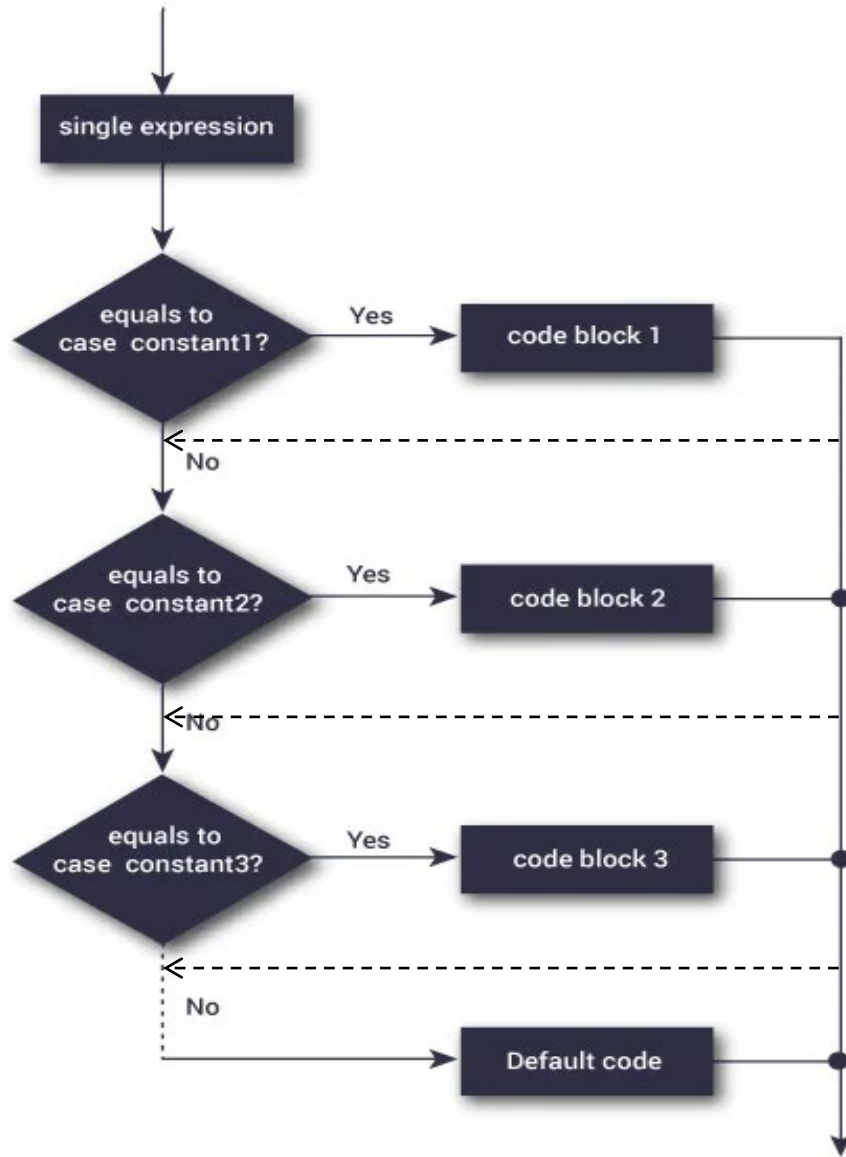
The switch statement is a multi-way decision that tests whether an expression matches one of a number of constant integer values, and branches accordingly. It provides a way to streamline multiple if-else conditions when the same variable or expression is being compared to different values.

```
switch (expression) {  
  case constant1:  
    // Code to be executed if expression matches constant1  
    break;  
  
  case constant2:  
    // Code to be executed if expression matches constant2  
    break;  
  
  // Additional cases as needed  
  
  default:  
    // Code to be executed if expression does not match any constant  
}
```

- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- All case expressions must be different.
- Each case is labeled by one or more integer-valued constants or constant expressions or enumerated types.
- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- If a case matches the expression value, execution starts at that case until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement. This will stop the execution of more code and case testing inside the block.

- A break can save a lot of execution time because it "ignores" the execution of all the rest of the code in the switch block
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- The case labeled default is executed if none of the other cases are satisfied.
  - A default is optional; if it isn't there and if none of the cases match, no action at all takes place.
- Cases and the default clause can occur in any order.
- Falling through from one case to another is not robust, better avoiding it.
- As a matter of good form, put a break after the last case (usually the default one) even though it's logically unnecessary. If another case gets added at the end, this break could save your program correctness.

# Switch statement flowchart





# Switch statement



Let's consider an example where we want to determine the day of the week based on a numeric code. In this case, the numeric code represents the day of the week, and we'll use a switch statement to handle different cases.

```
#include <stdio.h>
int main() {
    int dayCode = 3; // Example code for Wednesday
    switch (dayCode) {
        case 1:
            printf("Monday\n");
            break;
        case 2:
            printf("Tuesday\n");
            break;
        case 3:
            printf("Wednesday\n");
            break;
        case 4:
            printf("Thursday\n");
            break;
```

```
        case 5:
            printf("Friday\n");
            break;
        case 6:
            printf("Saturday\n");
            break;
        case 7:
            printf("Sunday\n");
            break;
        default:
            printf("Invalid day code\n");
    }
    return 0;
}
```

# Switch statement



```
#include <stdio.h>

int main () {

    /* local variable definition */
    char grade = 'B';

    switch(grade) {
        case 'A' :
            printf("Excellent!\n" );
            break;
        case 'B' :
        case 'C' :
            printf("Well done\n" );
            break;
```

```
        case 'D' :
            printf("You passed\n" );
            break;
        case 'F' :
            printf("Better try again\n" );
            break;
        default :
            printf("Invalid grade\n" );
    }

    printf("Your grade is %c\n", grade );

    return 0;
}
```

# Conditional or ternary operator '?:'



The statements

```
if (a > b)
```

```
z = a;
```

```
else
```

```
z = b;
```

compute in z the maximum of a and b. The conditional expression, written with the ternary operator ``?:'', provides an alternate way to write this and similar constructions. The conditional operator can be in the form

```
variable = expression1 ? expression2 : expression3;
```

or in the form

```
variable = (condition) ? expression2 : expression3;
```

or, simply,

```
expression1 ? expression2 : expression3;
```

# Conditional or ternary operator '?:'



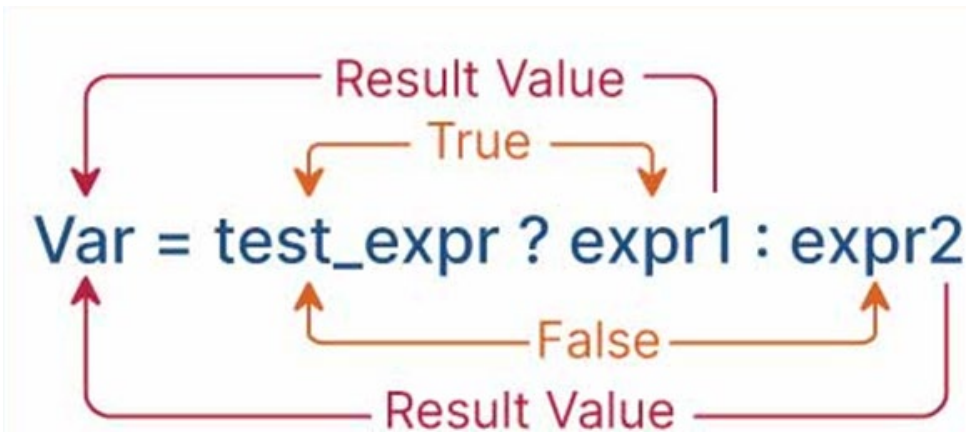
In the form

*expression1 ? expression2 : expression3*

the expression `expression1` is evaluated first. If it is non-zero (true), then the expression `expression2` is evaluated, and that is the value of the conditional expression. Otherwise `expression3` is evaluated, and that is the value. Only one of `expression2` and `expression3` is evaluated.

Thus to set `z` to the maximum of `a` and `b`, we can use the following statement

```
z = (a > b) ? a : b; /* z = max(a, b) */
```



# Conditional or ternary operator '?:'



```
#include <stdio.h>
```

```
int main() {
```

```
    int n1, n2;
```

```
    printf("Enter two numbers: ");
```

```
    scanf("%d %d", &n1, &n2);
```

```
    // Using the ternary operator to check which number is greater  
    (n1 > n2) ? printf("%d > %d", n1, n2) : printf("%d > %d", n2, n1);
```

```
    return 0;
```

```
}
```

# Conditional or ternary operator '?:'



```
#include <stdio.h>
```

```
int main() {  
    int age;
```

```
    printf("Enter your age: ");  
    scanf("%d", &age);
```

```
    // Using the ternary operator to check if the person is a minor or not  
    const char* status = (age < 18) ? "minor" : "adult";  
    printf("You are: %s", status);
```

```
    return 0;
```

```
}
```

## Advantages of Ternary Operators Over If-Else Statements

Ternary operators offer two main benefits over traditional if-else statements:

- They allow for more compact and efficient code.
- They can improve code readability by reducing the number of lines needed.

However, whether or not to use ternary operators is ultimately a matter of personal preference and coding style.

The goto statement in C is a jump statement that allows the program's control to jump to a labeled statement within the same function or block.

While the use of goto is generally discouraged due to its potential to create complex and less readable code, and indeed the it is never necessary, and in practice it is almost always easy to write code without it, the goto statement can be employed in specific scenarios where other control flow structures are not suitable.

The most common is to abandon processing in some deeply nested structure, such as breaking out of two or more loops at once. The break statement cannot be used directly since it only exits from the innermost loop.



# Goto and labels



While this example might be better solved using other control structures, it demonstrates the basic syntax of goto.

```
#include <stdio.h>
int main() {
    int i, j;
    for (i = 1; i <= 5; ++i) {
        for (j = 1; j <= 5; ++j) {
            if (i == 3 && j == 3) {
                // Use goto to break out of nested loops
                goto endLoop;
            }
            printf("%d %d\n", i, j);
        }
    }
    // Label to jump to
    endLoop:
    printf("Loop ended\n");

    return 0;
}
```

# Goto syntax and flow diagram



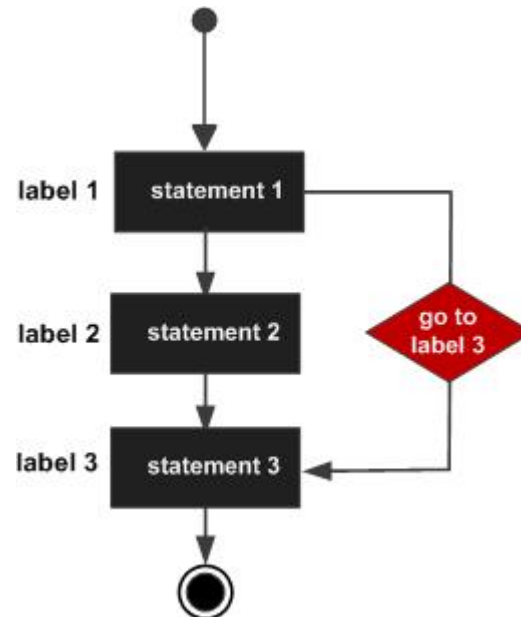
The syntax for a goto statement in C is as follows –

*goto label;*

...

*label: statement;*

Here label can be any plain text except a C keyword and it can be set anywhere in the C program above or below to goto statement.



# Reasons to avoid goto



The use of goto statement may lead to code that is buggy and hard to follow. For example,

*one:*

```
for (i = 0; i < number; ++i)
```

```
{
```

```
    test += i;
```

```
    goto two;
```

```
}
```

*two:*

```
if (test > 5) {
```

```
    goto three;
```

```
}
```

... ..

Should you use goto?

If you think the use of goto statement simplifies your program, you can use it. That being said, goto is rarely useful and you can create any C program without using goto altogether.

Here's a quote from Bjarne Stroustrup, creator of C++, "**The fact that 'goto' can do anything is exactly why we don't use it.**"

C provides certain language facilities by means of a preprocessor, which is conceptually a separate first step in compilation. The two most frequently used features are `#include`, to include the contents of a file during compilation, and `#define`, to replace a token by an arbitrary sequence of characters.

File inclusion makes it easy to handle collections of `#defines` and declarations.

A macro substitution has the form

*`#define name replacement text`*

subsequent occurrences of the token *name* will be replaced by the *replacement text*.

- All preprocessor commands begin with a hash symbol (`#`). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column.

# Preprocessor directives



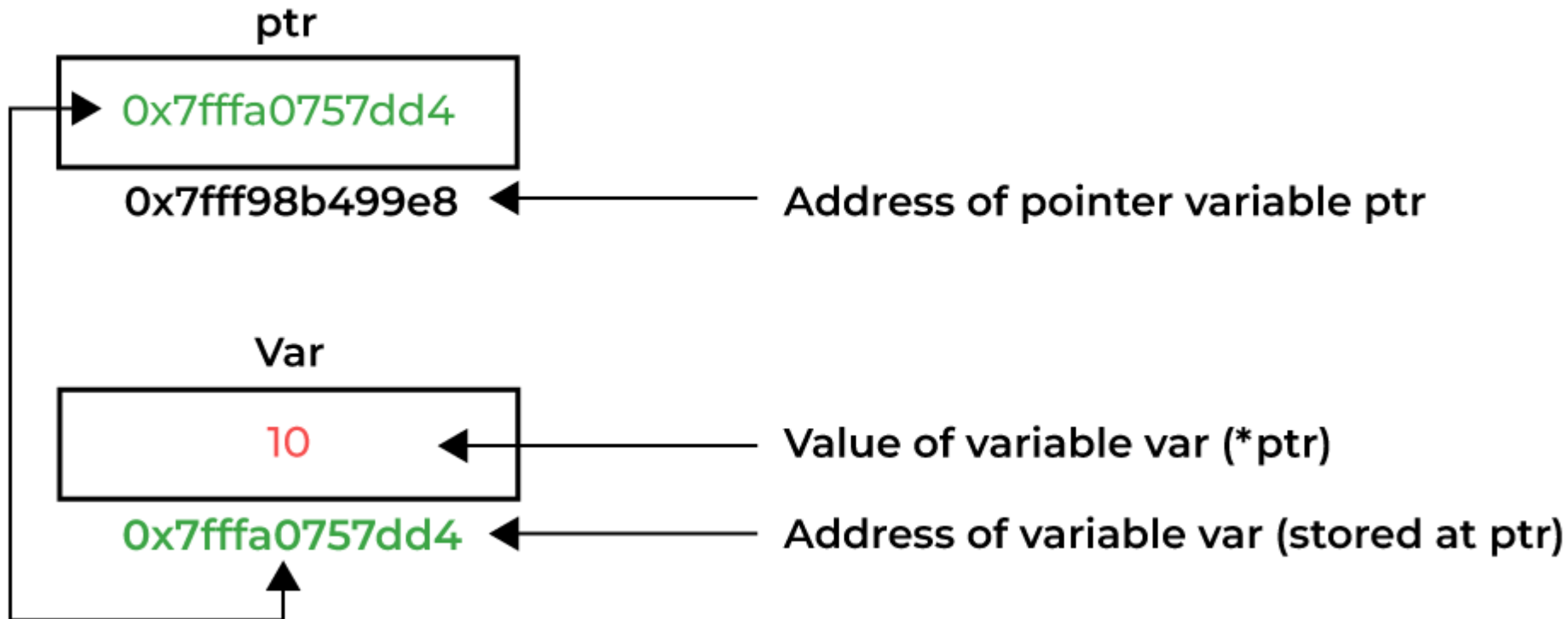
Directive	Description
<code>#define</code>	Substitutes a preprocessor macro
<code>#include</code>	Inserts a particular header from another file
<code>#undef</code>	Undefines a preprocessor macro
<code>#ifdef</code>	Returns true if this macro is defined
<code>#ifndef</code>	Returns true if this macro is not defined
<code>#if</code>	Tests if a compile time condition is true
<code>#else</code>	The alternative for <code>#if</code>
<code>#elif</code>	<code>#else</code> and <code>#if</code> in one statement
<code>#endif</code>	Ends preprocessor conditional
<code>#error</code>	Prints error message on stderr

A pointer is a variable that holds the memory address of another variable. Pointers can be used for storing addresses of dynamically allocated arrays and for arrays that are passed as arguments to functions.

This size of the pointer is fixed and only depends upon the architecture of the system.

A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups. One common situation is that any byte can be a char, a pair of one-byte cells can be treated as a short integer, and four adjacent bytes form a long. A pointer is a group of cells (often two or four) that can hold an address.

So, if var is an int and p is a pointer that points to it, we could represent the situation this way:



The unary operator **&** gives the address of an object, so the statement

```
ptr = &var;
```

assigns the address of *var* to the variable *ptr*, and *ptr* is said to “point to” *var*.

The **&** operator only applies to objects in memory: variables and array elements. It cannot be applied to expressions, constants, or register variables.

The unary operator **\*** is the indirection or dereferencing operator; when applied to a pointer, it accesses the object the pointer points to, so the statement

```
newVar = *ptr;
```

assigns the value of the variable (*var*) pointed by *ptr* to the variable *newVar*.



The declaration of the pointer `ptr`,

```
int *ptr;
```

is intended as a mnemonic; it says that the expression `*ptr` is an `int`.

If `ptr` points to the integer `var`, then `*ptr` can occur in any context where `var` can, so

```
*ptr = *ptr + 10;
```

increments `*ptr` by 10.

The unary operators `*` and `&` bind more tightly than arithmetic operators, so the assignment

```
newVar = *ptr + 1;
```

takes whatever `ptr` points at, adds 1, and assigns the result to `newVar`.

```
*ptr += 1;
```

increments what ptr points to, as do

```
++*ptr;
```

and

```
(*ptr)++;
```

The parentheses are necessary in this last example; without them, the expression would increment ptr instead of what it points to, because unary operators like \* and ++ associate right to left.

Finally, since pointers are variables, they can be used without dereferencing. For example, if ptrNew is another pointer to int,

```
ptrNew = ptr;
```

copies the contents of ptr into ptrNew, thus making ptrNew point to whatever ptr pointed to.

# Pointers and functions arguments

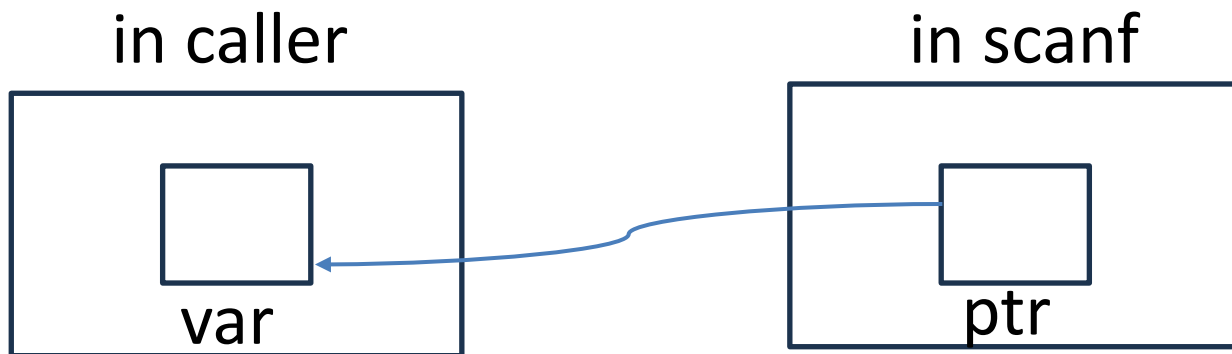


Since C passes arguments to functions by value, there is no direct way for the called function to alter a variable in the calling function.

The way to obtain the desired effect is for the calling program to pass pointers to the values to be changed as in `scanf` function arguments

```
scanf("%d", &var);
```

Pointer arguments enable a function to access and change objects in the function that called it.



# Pointers and arrays



In C, there is a strong relationship between pointers and arrays, strong enough that pointers and arrays should be discussed simultaneously. Any operation that can be achieved by array subscripting can also be done with pointers. The pointer version will in general be faster but somewhat harder to understand.

The declaration

```
int a[10];
```

defines an array of size 10, that is, a block of 10 consecutive objects named `a[0]`, `a[1]`, ..., `a[9]`.



# Pointers and arrays



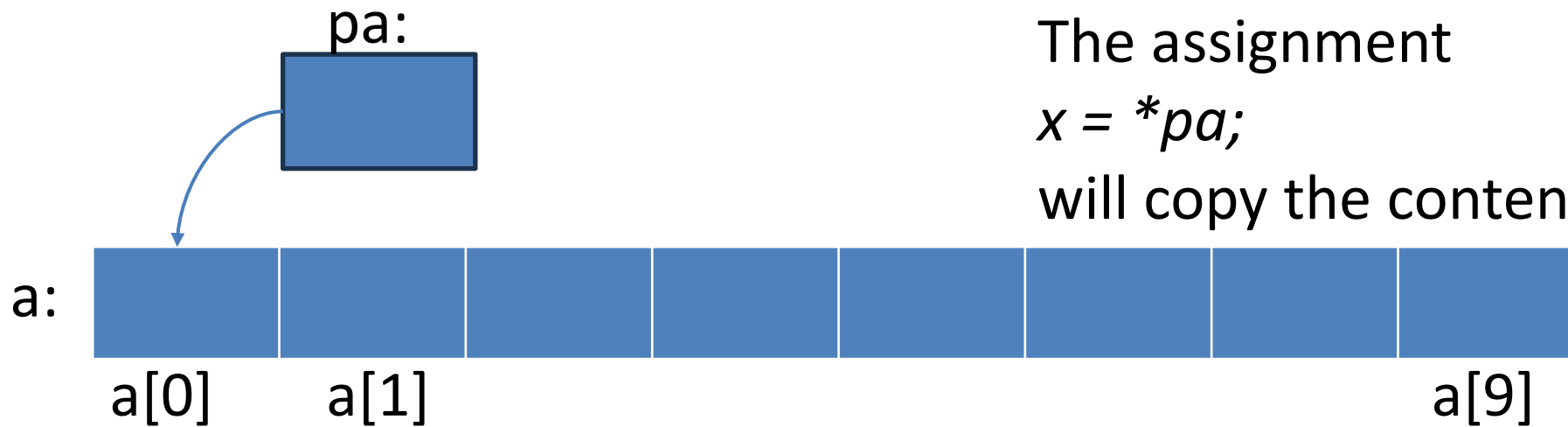
The notation  $a[i]$  refers to the  $i$ -th element of the array. If  $pa$  is a pointer to an integer, declared as

```
int *pa;
```

then the assignment

```
pa = &a[0];
```

sets  $pa$  to point to element zero of  $a$ ; that is,  $pa$  contains the address of  $a[0]$ .



The assignment

```
x = *pa;
```

will copy the contents of  $a[0]$  into  $x$ .

# Pointers and arrays



If  $pa$  points to a particular element of an array, then by definition  $pa+1$  points to the next element,  $pa+i$  points  $i$  elements after  $pa$ , and  $pa-i$  points  $i$  elements before. Thus, if  $pa$  points to  $a[0]$ ,

$*(pa+1)$

refers to the contents of  $a[1]$ ,  $pa+i$  is the address of  $a[i]$ , and  $*(pa+i)$  is the contents of  $a[i]$ .



These remarks are true regardless of the type or size of the variables in the array *a*.

The meaning of "adding 1 to a pointer," and by extension, all pointer arithmetic, is that *pa+1* points to the next object, and *pa+i* points to the *i*-th object beyond *pa* whichever is the data type of the element pointed by *pa*. By definition, the value of a variable or expression of type array is the address of element zero of the array. Thus, after the assignment

$$pa = \&a[0];$$

*pa* and *a* have identical values.

Since the name of an array is a synonym for the location of the initial element, the assignment *pa=&a[0]* can also be written as

$$pa = a;$$

Rather more surprising is the fact that a reference to  $a[i]$  can also be written as  $*(a+i)$ .

In evaluating  $a[i]$ , C converts it to  $*(a+i)$ ; the two forms are equivalent, that's  $a[i] == *(a+i)$

Applying the operator  $\&$  to both parts of this equivalence, it follows that  $\&a[i]$  and  $a+i$  are also identical, therefore  $a+i$  is the address of the  $i$ -th element beyond  $a$ .

As a consequence of this equivalence, if  $pa$  is a pointer, expressions might use it with a subscript;  $pa[i]$  is identical to  $*(pa+i)$ .

**In short, an array-and-index expression is equivalent to one written as pointer and offset.**



There is one main difference between an array name and a pointer that must be kept in mind.

- A pointer is a variable, so `pa=a` and `pa++` are legal.
- But an array name is not a variable, therefore, constructions like `a=pa` and `a++` are illegal.

When an array name is passed to a function, what is passed is the location of the initial element. Within the called function, this argument is a local variable, and so an array name parameter is a pointer, that is, a variable containing an address.

When an array name is passed to a function, the function can at its convenience believe that it has been handed either an array or a pointer, and manipulate it accordingly. It can even use both notations if it seems appropriate and clear.

It is possible to pass part of an array to a function, by passing a pointer to the beginning of the subarray.

For example,

```
f(&a[5]);
```

or

```
f(a+5);
```

both pass to the function *f* the address of the subarray that starts at *a[5]*.

Within `f`, the parameter declaration can be

```
f(int arr[]) { ... }
```

or

```
f(int *arr) { ... }
```

So as far as `f` is concerned, the fact that the parameter refers to part of a larger array is of no consequence.

Moreover, if one is sure that the elements exist, it is also possible to index backwards in an array; `p[-1]`, `p[-2]`, and so on are syntactically legal, and refer to the elements that immediately precede `p[0]`. But it is illegal to refer to objects that are not within the array bounds.

C is consistent and regular in its approach to address arithmetic; its integration of pointers, arrays, and address arithmetic is one of the strengths of the language.

If  $p$  is a pointer to some element of an array, then  $p++$  increments  $p$  to point to the next element, and  $p+=i$  increments it to point  $i$  elements beyond where it currently does.

In general a pointer can be initialized just as any other variable can, though normally the only meaningful values are zero or an expression involving the address of previously defined data of appropriate type.

Being a an array of char of a certain size, the declaration

```
static char *pa = a;
```

defines pa to be a character pointer and initializes it to point to the beginning of a. This could also have been written

```
static char *pa = &a[0];
```

since the array name is the address of the zeroth element.

C guarantees that zero is never a valid address for data, so a return value of zero can be used to signal an abnormal event occurred in functions returning address values that's pointers.

Pointers and integers are not interchangeable. Zero is the sole exception: the constant zero may be assigned to a pointer, and a pointer may be compared with the constant zero.

The symbolic constant `NULL` is often used in place of zero, as a mnemonic to indicate more clearly that this is a special value for a pointer. `NULL` is defined in `<stdio.h>`.

If `pa` and `qa` point to members of the same array, then relations like `==`, `!=`, `<`, `>=`, etc., work properly. For example,

`p < q`

is true if `p` points to an earlier element of the array than `q` does. Any pointer can be meaningfully compared for equality or inequality with zero.

The behavior is undefined for arithmetic or comparisons with pointers that do not point to members of the same array.

(There is one exception: the address of the first element past the end of an array can be used in pointer arithmetic.)

We have already observed that a pointer and an integer may be added or subtracted. The construction

$pa + n$

means the address of the  $n$ -th object beyond the one  $pa$  currently points to.

This is true regardless of the kind of object  $pa$  points to;  $n$  is scaled according to the size of the objects  $pa$  points to, which is determined by the declaration of  $pa$ . If an `int` is four bytes, for example, the `int` will be scaled by four.

Pointer subtraction is also valid: if  $p$  and  $q$  point to elements of the same array, and  $p < q$ , then  $q - p + 1$  is the number of elements from  $p$  to  $q$  inclusive. This fact can be used to write a simple version of `strlen`:

```
/* strlen: return length of string s */  
int strlen(char *s)  
{  
    char *p = s;  
    while (*p != '\0')  
        p++;  
    return p - s;  
}
```



Pointer subtraction is also valid: if  $p$  and  $q$  point to elements of the same array, and  $p < q$ , then  $q - p + 1$  is the number of elements from  $p$  to  $q$  inclusive. This fact can be used to write a simple version of `strlen`:

```
/* strlen: return length of string s */  
int strlen(char *s)  
{  
    char *p = s;  
    while (*p != '\0')  
        p++;  
    return p - s;  
}
```

The valid pointer operations are

- assignment of pointers of the same type,
- adding or subtracting a pointer and an integer,
- subtracting or comparing two pointers to members of the same array,
- assigning or comparing to zero.

All other pointer arithmetic is illegal.

It is not legal to add two pointers, or to multiply or divide or shift or mask them, or to add float or double to them, or even, except for void \*, to assign a pointer of one type to a pointer of another type without a cast.

A string constant, written as

*"I am a string"*

is an array of characters. The length in storage is one more than the number of characters between the double quotes, because in the internal representation, the array is terminated with the null character '\0'.

The most common occurrence of string constants is as arguments to functions *printf("hello, world\n");*

When a character string like this appears in a program, access to it is through a character pointer; indeed printf receives a pointer to the beginning of the character array.

Therefore, a string constant is accessed by a pointer to its first element.

# Character pointers and functions



If `pmessage` is declared as

```
char *pmessage;
```

then the statement

```
pmessage = "I am a string";
```

assigns to `pmessage` a pointer to the character array.

This is not a string copy; only pointers are involved.

There is an important difference between these definitions:

```
char amessage[] = "I am a string"; /* an array */
```

```
char *pmessage = "I am a string"; /* a pointer */
```



`amessage` is an array, just big enough to hold the sequence of characters and `'\0'` that initializes it. Individual characters within the array may be changed but `amessage` will always refer to the same storage.

# Character pointers and functions



On the other hand, `pmessage` is a pointer, initialized to point to a string constant; the pointer may subsequently be modified to point elsewhere, but the result is undefined if you try to modify the string contents.

`amessage:` 

`pmessage:`  

# Character pointers and functions



Let's consider the function `strcpy(s,t)`, which copies the string `t` to the string `s`. It would be nice just to say `s=t` but this copies the pointer, not the characters. To copy the characters, we need a loop.

```
/* strcpy: copy t to s; array subscript version */  
void strcpy(char *s, char *t)  
{  
    int i = 0;  
  
    while ((s[i] = t[i]) != '\0')  
        i++;  
}
```

# Character pointers and functions



Because arguments are passed by value, strcpy can use the parameters s and t in any way it prefers. In the next code they are conveniently initialized pointers, which are marched along the arrays a character at a time, until the '\0' that terminates t has been copied into s.

```
/* strcpy: copy t to s; pointer version */  
void strcpy(char *s, char *t)  
{  
    while ((*s = *t) != '\0') {  
        s++;  
        t++;  
    }  
}
```

# Character pointers and functions



Really, the function would likely be written as

```
/* strcpy: copy t to s; pointer version */  
void strcpy(char *s, char *t)  
{  
    while (*s++ = *t++)  
        ;  
}
```

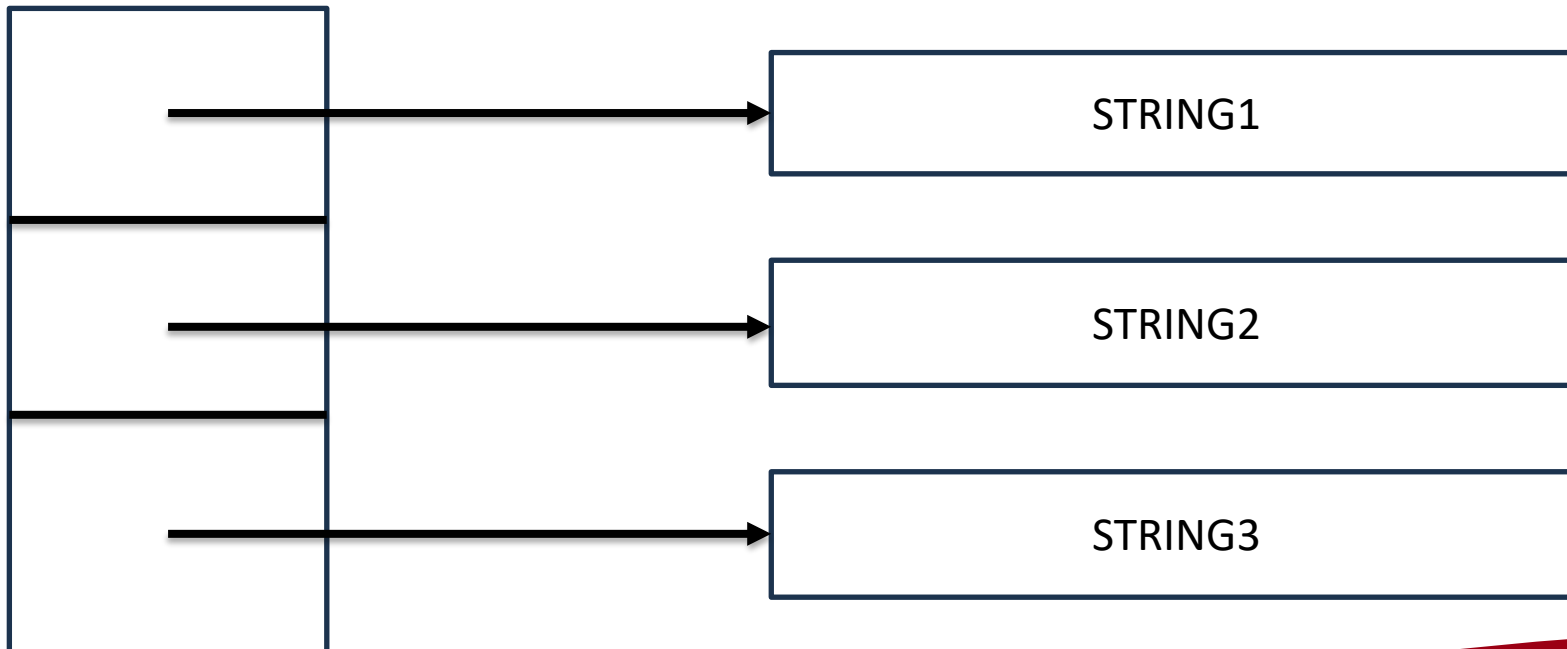


# Pointer Arrays; Pointers to Pointers



Since pointers are variables themselves, they can be stored in arrays just as other variables can.

Therefore we can define arrays of pointers, that are equivalent to pointers to pointers.



# Multi-dimensional Arrays



C provides rectangular multi-dimensional arrays, although in practice they are much less used than arrays of pointers.

In C, a two-dimensional array is really a one-dimensional array, each of whose elements is an array. Hence subscripts are written as

```
vector[i][j] /* [row][col] */
```

rather than

```
vector[i,j] /* WRONG */
```

Elements are stored by rows, so the rightmost subscript, or column, varies fastest as elements are accessed in storage order.

An array is initialized by a list of initializers in braces; each row of a two-dimensional array is initialized by a corresponding sub-list.

```
static char daytab[2][13] = {  
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},  
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}}
```

If a two-dimensional array is to be passed to a function, the parameter declaration in the function must include the number of columns; the number of rows is irrelevant, since what is passed is, as before, a pointer to an array of rows, where each row is an array of 13 ints.

In this particular case, it is a pointer to objects that are arrays of 13 ints. Thus if the array `daytab` is to be passed to a function `f`, the declaration of `f` would be:

```
f(int daytab[2][13]) { ... }
```

Thus if the array `daytab` is to be passed to a function `f`, the declaration of `f` would be:

```
f(int daytab[2][13]) { ... }
```

It could also be

```
f(int daytab[][13]) { ... }
```

since the number of rows is irrelevant, or it could be

```
f(int (*daytab)[13]) { ... }
```

which says that the parameter is a pointer to an array of 13 integers. The parentheses are necessary since brackets `[]` have higher precedence than `*`.

Without parentheses, the declaration

```
int *daytab[13]
```

is an array of 13 pointers to integers.

More generally, only the first dimension (subscript) of an array is free; all the others have to be specified.

How initialization of Pointer Arrays can be managed?

```
/* month_name: name of n-th month */  
static char *month_name[] = {  
    "Illegal month", "January", "February", "March", "April", "May", "June",  
    "July", "August", "September", "October", "November", "December"  
};
```

# Multi-dimensional Arrays



The declaration of `month_name` is an array of character pointers. The initializer is a list of character strings; each is assigned to the corresponding position in the array.

The characters of the  $i$ -th string are placed somewhere, and a pointer to them is stored in `month_name[i]`.

Since the size of the array name is not specified, the compiler counts the initializers and fills in the correct number.

What about the difference between a two-dimensional array and an array of pointers?

# Multi-dimensional Arrays



Given the definitions

```
int a[10][20];
```

```
int *b[10];
```

then `a[3][4]` and `b[3][4]` are both syntactically legal references to a single `int`.

But `a` is a true two-dimensional array: 200 `int`-sized locations have been set aside, and the conventional rectangular subscript calculation  $20 * \text{row} + \text{col}$  is used to find the element `a[row][col]`. For `b`, however, the definition only allocates 10 pointers and does not initialize them; initialization must be done explicitly, either statically or with code.

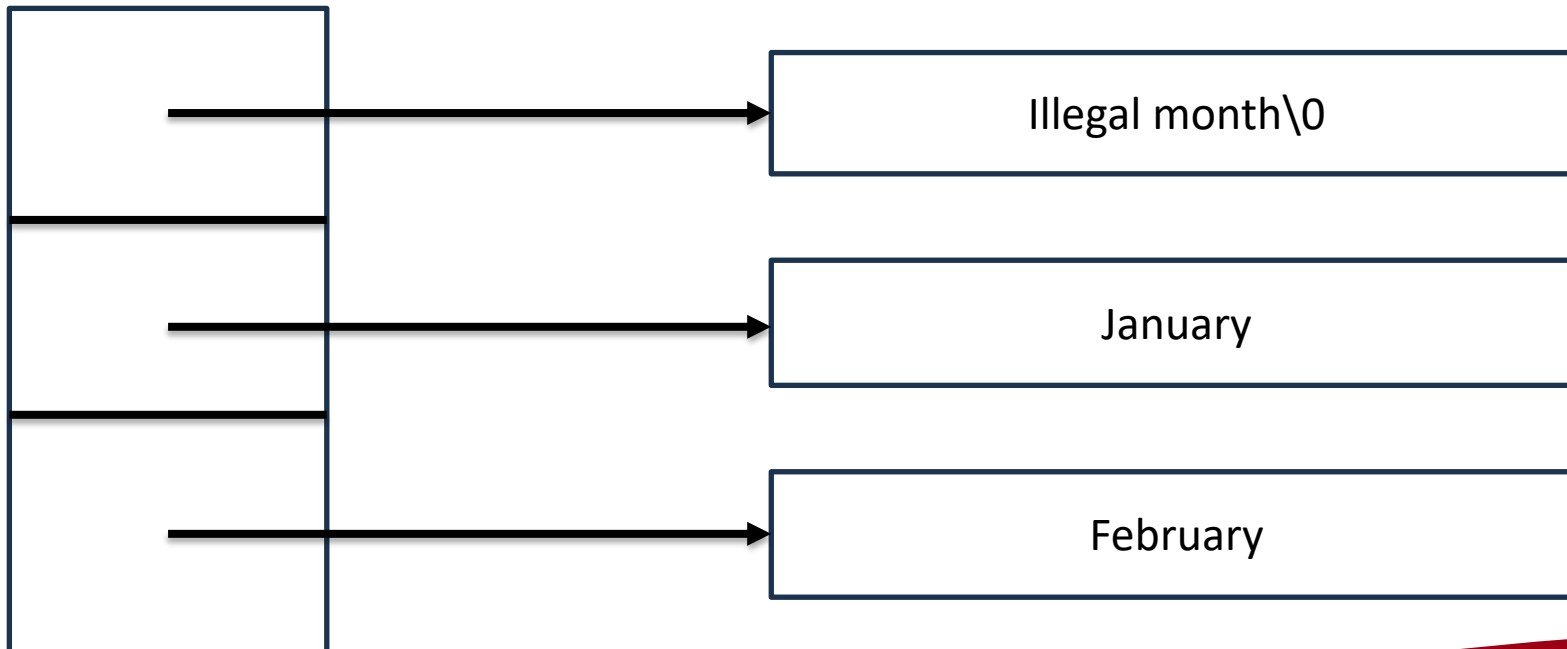
The important advantage of the pointer array is that the rows of the array may be of different lengths.

# Multi-dimensional Arrays



That is, each element of `b` need not point to a twenty-element vector; some may point to two elements, some to fifty, and some to none at all.

The most frequent use of arrays of pointers is to store character strings of different lengths





Pointers to functions in C allow you to store the address of a function in a variable, providing a way to call a function indirectly through the pointer. This feature is powerful and often used in scenarios where functions need to be passed as arguments to other functions or stored in data structures.

Declaration:

```
return_type (*pointer_name)(parameter_types);
```

- `return_type`: The return type of the function.
- `pointer_name`: The name of the pointer to the function.
- `parameter_types`: The types of parameters the function takes.

Example:

```
int (*addPointer)(int, int); /* Pointer to a function that takes two integers  
and returns an integer */
```

Initialization:

```
pointer_name = &function_name;
```

- `&function_name`: The address-of operator is used to get the address of the function.

Example:

```
int add(int a, int b) {  
    return a + b;  
}
```

```
int (*addPointer)(int, int) = &add; /* Initializing the pointer with the address  
of the 'add' function */
```

Function call through pointer

```
return_type result = pointer_name(arg1, arg2, ...);
```

- *arg1, arg2, ...*: Arguments passed to the function through the pointer.

Example:

```
int result = (*addPointer)(3, 4); /* Calling the 'add' function through the  
pointer */
```

# Pointers to functions



```
#include <stdio.h>

// Function to add two integers
int add(int a, int b) {
    return a + b;
}

// Function to subtract two integers
int subtract(int a, int b) {
    return a - b;
}
```

**In this example, two functions (add and subtract) are defined. Pointer to functions (operationPointer) is declared, initialized with the addresses of these functions, and used to call the functions indirectly.**

```
int main() {
    // Declare pointers to functions
    int (*operationPointer)(int, int);

    // Initialize pointers with addresses of functions
    operationPointer = &add;
    printf("Result of add: %d\n", (*operationPointer)(5, 3));

    operationPointer = &subtract;
    printf("Result of subtract: %d\n",
    (*operationPointer)(5, 3));

    return 0;
}
```

Dynamic memory allocation in C is the process of allocating memory during the execution of a program. It allows you to allocate memory at runtime, and it's managed using pointers.

Dynamic memory allocation is achieved using three key functions: **malloc()**, **calloc()**, and **realloc()**.

Memory allocated dynamically needs to be explicitly deallocated using the **free()** function to prevent memory leaks.

# malloc() (Memory Allocation)



The malloc function in C (memory allocation) is used to dynamically allocate a specified number of bytes of memory during program execution. It stands for "memory allocation." This function is part of the <stdlib.h> library.

Syntax:

```
void *malloc(size_t size);
```

- `size_t`: Unsigned integer type used for sizes.
- `size`: Number of bytes to allocate.

Usage:

```
int *arr = (int *)malloc(5 * sizeof(int));
```

Allocates space for an array of 5 integers.

The result is cast to `int *` because malloc returns a generic `void *` pointer.

# malloc() (Memory Allocation)



## Return Value:

If the memory allocation is successful, malloc returns a pointer to the beginning of the allocated memory.

If the allocation fails (usually due to insufficient memory), it returns NULL.

## Check for Allocation Failure:

```
if (arr == NULL) {  
    printf("Memory allocation failed\n");  
    return 1;  
}
```

# calloc() (Contiguous Allocation)



The `calloc` function in C (contiguous allocation) is used to dynamically allocate a specified number of blocks of memory, each of a specified size, during program execution. It is part of the `<stdlib.h>` library, just like `malloc`. The key difference between `malloc` and `calloc` is that `calloc` initializes the allocated memory to zero.

Syntax:

```
void *calloc(size_t num_elements, size_t element_size);
```

- `num_elements`: Number of elements to allocate.
- `element_size`: Size of each element in bytes.



# calloc() (Contiguous Allocation)



## Return Value:

If the memory allocation is successful, `calloc` returns a pointer to the beginning of the allocated memory.

If the allocation fails (usually due to insufficient memory), it returns `NULL`.

## Usage:

```
int *arr = (int *)calloc(5, sizeof(int));
```

Allocates space for an array of 5 integers and initializes them to zero.

The result is cast to `int *` because `calloc` returns a generic `void *` pointer.

# calloc() (Contiguous Allocation)



Check for Allocation Failure:

```
if (arr == NULL) {  
    printf("Memory allocation failed\n");  
    return 1;  
}
```

# realloc() (Reallocation)



The `realloc` function in C (reallocation) is used to dynamically change the size of a previously allocated block of memory during program execution. It allows you to resize the memory allocated by `malloc` or `calloc`. The function is part of the `<stdlib.h>` library.

Syntax:

```
void *realloc(void *ptr, size_t new_size);
```

- `ptr`: Pointer to the previously allocated memory.
- `new_size`: New size in bytes.

# realloc() (Reallocation)



## Return Value:

If the reallocation is successful, `realloc` returns a pointer to the beginning of the reallocated memory.

If the reallocation fails (usually due to insufficient memory), it returns `NULL`. The original memory block remains unchanged in this case.

## Usage:

```
int *arr = (int *)malloc(5 * sizeof(int));  
// ... use arr ...  
arr = (int *)realloc(arr, 10 * sizeof(int));
```

Changes the size of the previously allocated array to accommodate 10 integers.

# realloc() (Reallocation)



Check for Reallocation Failure:

```
if (arr == NULL) {  
    printf("Memory reallocation failed\n");  
    free(arr); // Free the previously allocated memory  
    return 1;  
}
```

It's important to check if the reallocation was successful, as it may fail due to insufficient memory.

# free() (Deallocation)



The free function in C is used to deallocate memory that was previously allocated dynamically using functions like malloc, calloc, or realloc. It is part of the <stdlib.h> library.

Syntax:

```
void free(void *ptr);
```

- ptr: Pointer to the dynamically allocated memory.

Usage:

```
int *arr = (int *)malloc(5 * sizeof(int));
```

```
// ... use arr ...
```

```
free(arr);
```

Deallocates the dynamically allocated memory pointed to by arr.

# free() (Deallocation)



Return Value:

free doesn't return any value.

After calling free, the memory is released, and the pointer should not be used further. Always ensure that the pointer being passed to free is the same as the one returned by the memory allocation functions, and it has not been deallocated earlier.

# Example of memory management



```
#include <stdio.h>
#include <stdlib.h>

int main() {
    // Dynamic memory allocation using malloc
    int *arr1 = (int *)malloc(5 * sizeof(int));
    if (arr1 == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }

    // Dynamic memory allocation using calloc
    int *arr2 = (int *)calloc(5, sizeof(int));
    if (arr2 == NULL) {
        printf("Memory allocation failed\n");
        free(arr1); // Free the previously allocated memory
        return 1;
    }

    // Reallocate memory using realloc
    arr1 = (int *)realloc(arr1, 10 * sizeof(int));
    if (arr1 == NULL) {
        printf("Memory reallocation failed\n");
        free(arr2); // Free the previously allocated memory
        return 1;
    }

    // Deallocate memory using free
    free(arr1);
    free(arr2);

    return 0;
}
```



# free() (Deallocation)



In this example:

malloc and calloc are used to allocate memory for integer arrays.

realloc is used to resize the array allocated by malloc.

free is used to deallocate the dynamically allocated memory.

Check for the success of memory allocation to handle potential failures.