COMPUTER ENGINEERING LABORATORY

Luigi Rizzo

luigi.rizzo@unipd.it

October 2024-January 2025

UNIVERSITÀ DEGLI STUDI DI PADOVA

Command-line Arguments, struct, arithmetic expressions evaluation, functions declaration and definition, bitwise operators, input and output hints

UNIVERSITÀ
DEGLI STUDI
DI PADOVA

Command-line Arguments

In environments that support C/C++, there is a way to pass command-line arguments or parameters to a program when it begins executing.
Command-line arguments are a way to pass information to a C program from the command line when you run it. These arguments are commonly used for configuration, input data, or other parameters that affect how the program behaves. In C/C++, command-line arguments are received as parameters to the main function.
When main is called, it is called with two arguments.
The first (conventionally called argc, for argument count) is the number of command-line arguments the program was invoked with, including the program name itself; the second (argv, for argument vector) is a pointer to an array of character strings that contain the arguments, one per string.

Command-line Arguments

By convention, argv[0] is the name by which the program was invoked, so argc is at least 1.
If argc is 1, there are no command-line arguments after the program name.
Therefore command-line arguments are strings of characters that you provide when running a C program.
The main function has the following signature:

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

You can access and process command-line arguments by using argc to determine the number of arguments and argv to access the argument values.

Command-line Arguments

For example, to access the second command-line argument:

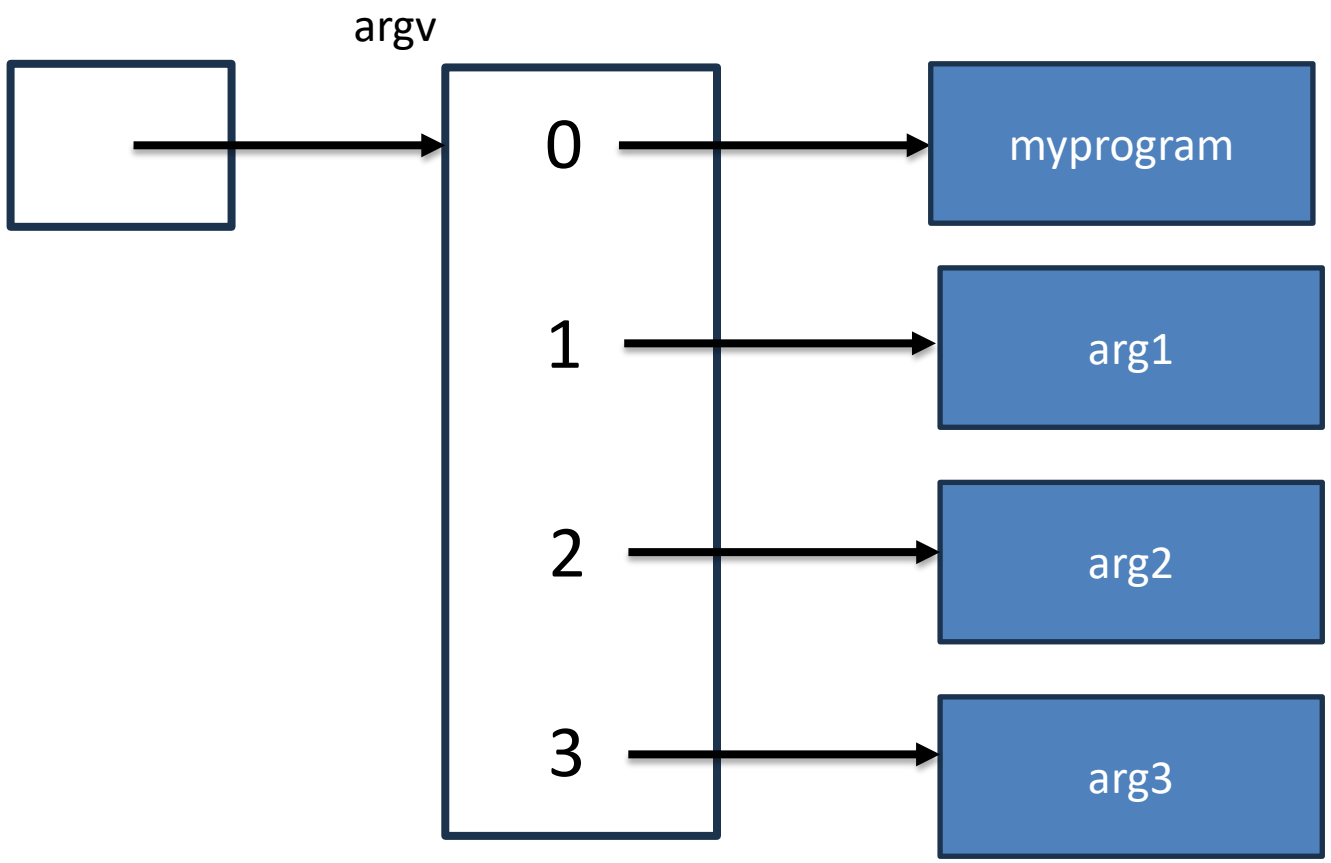
```
char *secondArg = argv[2];
```

To run a C/C++ program with command-line arguments, you typically execute it from the command line and provide the arguments after the program name:

```
$ ./myprogram arg1 arg2 arg3
```

In this example, myprogram is the name of the C program, and “arg1”, “arg2”, and “arg3” are the command-line arguments.

Command-line Arguments




```
for (int i = 1; i < argc; i++)
    printf("%s\t", argv[i]);
```

Command-line Arguments

An example to demonstrate how command-line arguments can be used:

```
#include <iostream>
int main(int argc, char *argv[]) {
    // Print the number of arguments
    std::cout << "Number of arguments: " << argc << std::endl;

    // Loop through and print each argument
    for (int i = 0; i < argc; i++) {
        std::cout << "Argument " << i << ": " << argv[i] << std::endl;
    }

    return 0;
}
```

Command-line Arguments

Compile and run the program from the command line, passing arguments:
/my_program arg1 arg2 arg3

Output:
Number of arguments: 4
Argument 0: ./my_program
Argument 1: arg1
Argument 2: arg2
Argument 3: arg3

What about if you need to convert command-line arguments to other types (e.g., integers)?

Converting command-line Arguments

You can use functions like std::atoi or atoi() for string-to-integer conversions.

```
#include <iostream>
#include <cstdlib> // for atoi

int main(int argc, char *argv[]) {
    if (argc > 1) {
        int num = std::atoi(argv[1]);  // Convert the first argument to an integer
        std::cout << "The first argument as an integer: " << num << std::endl;
    } else {
        std::cout << "No arguments provided!" << std::endl;
    }
    return 0;
}
```

Converting command-line Arguments

*int atoi(const char *str);*

Parameter: The string you want to convert.

Return value: The integer value of the string. If the string cannot be converted, it returns 0.

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
    if (argc > 1) {
        int num = atoi(argv[1]);  // Convert the first argument to an integer
        printf("String: %s, Integer: %d\n", argv[1], num);
    }
    return 0;
}
```

Converting command-line Arguments

Note: If the string contains non-numeric characters, atoi() will stop converting at the first non-numeric character and return the result up to that point. This can lead to errors in complex input scenarios.

For more control and better error handling, you can use strtol() (string to long integer). It can handle conversion errors more gracefully and supports different bases (e.g., base 10, base 16).

*long int strtol(const char *str, char **endptr, int base);*

Parameters:

str: The string to be converted.

endptr: A pointer to a pointer, will point to the first non-converted character.

base: The numerical base (e.g., 10 for decimal, 16 for hexadecimal).

Converting strings to integer values

Output:
Converted number: 123
Non-numeric part: abc

Why use strtol() over atoi()?

- Error Handling: strtol() allows you to check if the entire string was converted or if there was an error.
- Base Conversion: You can convert numbers in different bases (e.g., hexadecimal).

For simple conversions, atoi() works fine, but strtol() is recommended when you need more control.

Converting strings to integer values


```
#include <stdio.h>
#include <stdlib.h>
int main() {
    char str[] = "123abc";
    char *endptr;
    long int num;

    num = strtol(str, &endptr, 10);
    printf("Converted number: %ld\n", num);
    printf("Non-numeric part: %s\n", endptr);  // Points to "abc“

    return 0;
}
```

Command-line Arguments

Command-line arguments are commonly used for various purposes, such as specifying input files, configuration options, debugging flags, and any information that you want to pass to the program when it's executed.
They are especially useful when writing scripts or programs that need to be parameterized without modifying the source code.

When working with command-line arguments, it's essential to perform error checking to ensure that the expected number of arguments is provided and to handle invalid or missing arguments gracefully.

Struct

- Structures are **derived data types**, built using elements (members) of other types
  - A structure is defined with the keyword: **struct**
  - Example:

```
struct   student    {
   char   surname[20];
   char   name[20];
   unsigned   int   id;
   unsigned   int   birthYear;
   char   gender;
};
```

Variables declared within the parentheses are the members of the struct

Struct

- The type of the members of a struct can be any:
  - int, float, char, …
  - arrays, other structs, unions, …
  - Example:

```
struct    point        {
int x;
int y;
};
struct    rectangle    {
   struct point highLeft;
   struct point lowRight;
};
```

Struct

- Structure definitions do not reserve any memory space but create a new data type
- Structure type variables are declared like all other "standard" type variables (i.e. int, float, char ...)

```
struct    student std1, std2;
```

```
struct    student {
…
} std1, std2;
```

Struct

- Two operators are used to access individual members:
  - The structure member operator: .
  - The structure pointer operator: ->
  - Let's see two examples:

```
printf("%s\t%s\n", std1.name, std1.surname);
```

(Assuming that the pointer stdPtr was initialized with the std1 address)

```
printf("%s\t%s\n", stdPtr->name, stdPtr->surname);
```

Struct

- It is a good idea to combine the definition of structures with typedef for greater readability.
- Basically, typedef is used to assign symbolic names (aliases) to data types
- Usually struct and typedef are used together, but a classic example of using typedef is also the following:

```
typedef unsigned char Byte;
```

- There is some similarity between typedef and #define (both allow you to define aliases/symbolic names), however:
  - typedef is limited to defining aliases for types
  - #define is “more general” (e.g. we can define aliases for values) and is processed by the pre-processor, while typedef statements are processed by the compiler

Struct

- Types can use a different number of bytes depending on the architecture/compiler
  - For example int can use 2 or 4 bytes
- A very useful (and used) operator in C is sizeof
  - It is a unary operator that returns the number of bytes necessary to allocate a type or variable

```
sizeof(int);        //    4
```

```
char     name[20];

sizeof(name);       //    20
```

Expression Evaluation Rules

- The arithmetic of C is not always the same as how we carry out the calculations (we must take this into account).

$$\frac{3}{2} * 2$$

- How does C calculate the expression?

Hypothesis 1

$$\frac{3}{2} * 2 = 3$$

?

NO because the expressions are not executed in the expected order:

(3/2)*2

Expression Evaluation Rules

- How does C calculate the expression?

Hypothesis 2

$$\left(\frac{3}{2}\right) * 2 = 1.5 * 2 = 3$$

?

NO because 3/2 is an operation between integers, not between reals (the numbers do not have the decimal part, 3.0). The expression below is correct:

3.0/2.0*2.0=3

So which is the result of 3/2*2?

Hypothesis 3

$$\left(\frac{3}{2}\right) * 2 = 1 * 2 = 2$$

conversions between types

- In C the operands of an expression must have the same type: what happens if we try to add an integer and a real?

$$3.0 + 2 = ?$$

- C transforms the integer into a real (it is possible to force the opposite transformation, we will see it in the next slide):

$$3.0 + 2.0 = 5.0$$

conversions between types

Explicit type conversions can be forced (``coerced'') in any expression, with a unary operator called a cast.
In the construction
(*type name*) *expression*
the expression is converted to the named type by the conversion rules above.

The precise meaning of a cast is as if the expression were assigned to a variable of the specified type, which is then used in place of the whole construction.

conversions between types

For example, the library routine sqrt expects a double argument, and will produce nonsense if inadvertently handled something else.

So if `n` is an integer, we can use

```
sqrt((double) n)
```

to convert the value of `n` to `double` before passing it to `sqrt`

functions

In C, a function provides a convenient way to encapsulate some computation, which can then be used without worrying about its implementation. They encapsulate a set of instructions that can be executed repeatedly or as needed. With properly designed functions, it is possible to ignore how a job is done; knowing what is done is sufficient. C makes the use of functions easy, convenient and efficient.

Let's write a function of our own.

C has no exponentiation operator, we will illustrate the mechanics of function definition by writing a function power(m,n) to raise an integer m to a positive integer power n.

Here is the function power and a main program to exercise it.

functions


```
#include <stdio.h>
int power(int base, int n);
/* test power function */
main()
{
    int i;
    for (i = 0; i < 10; ++i)
    printf("%d %d %d\n", i, power(2,i), power(-3,i));
    return 0;
}
```

functions


```
/* power: raise base to n-th power; n >= 0 */
int power(int base, int n)
{
    int result =  1;
    for (int i = 1; i <= n; ++i)
        result *= base;
    return result;
}
```

- We initialize result to 1 (since any number to the power of 0 is 1).
- We then loop n times, multiplying result by base each time.

What about handling negative exponents?
We shall modify the function to return a floating-point value.
Here's an example that handles both positive and negative exponents:

```
#include <stdio.h>

double power(double base, int exponent) {
    double result = 1.0;
    int positive_exp = (exponent < 0) ? -exponent : exponent;  // Get absolute value of exponent

    for (int i = 0; i < positive_exp; i++) {
        result *= base;
    }

    return (exponent < 0) ? 1.0 / result : result;  // Handle negative exponents
}
```

functions

A function definition has this form:

return-type function-name(parameter declarations, if any)
{
declarations
statements
}

A function in C is defined using the *return_type* (the type of value the function returns), a *function_name* (an identifier for the function), and a list of parameters (input values, if any).

The first line of power itself,

int power(int base, int n)

declares the parameter types and names, and the type of the result that the function returns.

functions

Function definitions can appear in any order, and in one source file or several, although no function can be split between files.

The function body contains a sequence of statements that define the function's behavior. It can include variable declarations, calculations, and control structures like loops and conditionals.

To use a function, you call it by its name and pass any required arguments. The function executes its code and may return a value.
For example, the function power is called twice by main, in the line

printf("%d %d %d\n", i, power(2,i), power(-3,i));

Each call passes two arguments to power, which each time returns an integer to be formatted and printed.

functions

Parameters are variables defined in the function's declaration. Arguments are values passed to the function when it's called. The values of arguments are assigned to the parameters inside the function.

The declaration

int power(int base, int n);

just before *main* says that power is a function that expects two *int* arguments and returns an *int*. This declaration, is called a function prototype.

Parameter names are optional in a function prototype, so for the prototype we could have written

int power(int, int);

Well-chosen names are good documentation, however, so better use them.

functions

The return statement is used to send a value back to the calling code. The return type in the function declaration specifies the type of value that the function can return.
The integer value that power computes is returned to main by the return statement. Any expression may follow return:

return expression;

A function needs not return a value. There can be a return statement at the end of main too. Since main is a function like any other, it may return a value to its caller, which is in effect the environment in which the program is executed. Typically, a return value of zero implies normal termination; non-zero values signal unusual or erroneous termination conditions.

functions

- C provides a wide range of built-in library functions (e.g., printf, scanf, strlen) that simplify common tasks and can be used in your programs.
- Functions can call themselves, which is known as recursion. Recursive functions are often used to solve problems that can be broken down into smaller, similar subproblems.
- If a function doesn't need to return a value, you can specify *void* as the return type.

Arguments - Call by Value/Reference

In C, all function arguments are passed ``by value.'' This means that the called function is given the values of its arguments in temporary variables rather than the originals. This doesn't affect the original value outside the function.

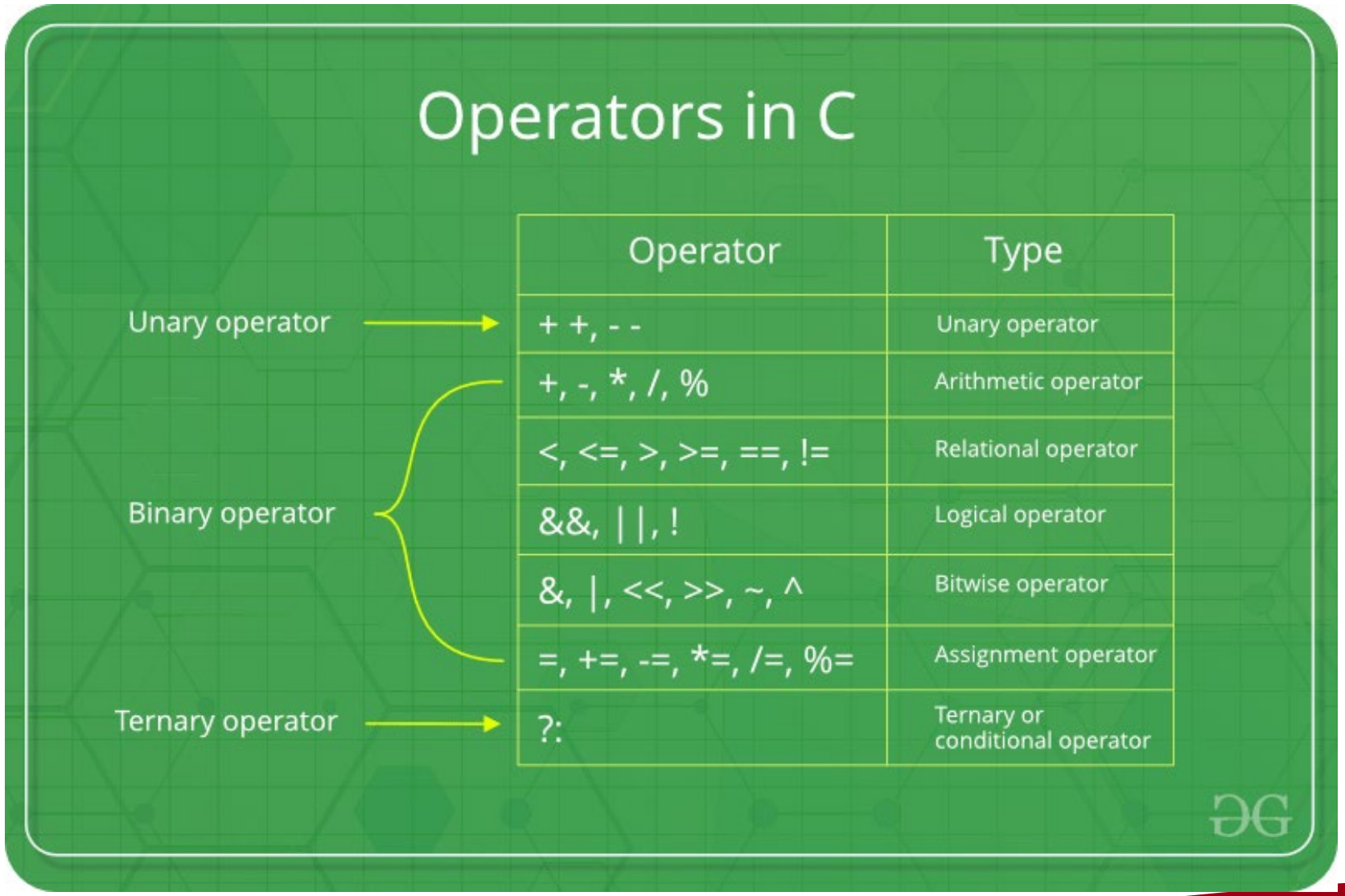
To modify the original values of variables within a function, you can pass pointers to those variables. This allows you to achieve "pass by reference" behavior. The caller must provide the address of the variable to be set (technically a pointer to the variable), and the called function must declare the parameter to be a pointer and access the variable through it.

Arguments - Call by Value/Reference

In the case of arrays, when the name of an array is used as an argument, the value passed to the function is the location or address of the beginning of the array - there is no copying of array elements.

Functions can be used to encapsulate error-handling code, which helps keep the main program flow cleaner and more focused on the primary task.

Bitwise operators



Bitwise operators

Bitwise operators in C are used to perform operations at the bit level, allowing you to manipulate individual bits of data. These operators are essential for tasks like working with flags, setting or clearing specific bits, and optimizing memory usage. There are six bitwise operators in C: AND (&), OR (|), XOR (^), NOT (~), left shift (<<), and right shift (>>).

Bitwise AND (**&**):

The bitwise AND operator (&) performs a bitwise AND operation on each pair of corresponding bits of two integers. It returns a 1 if both bits are 1, otherwise, it returns 0.
1010 & 1101 ➔ 1000

Bitwise operators

Bitwise OR (**|**):

The bitwise OR operator (|) performs a bitwise OR operation on each pair of corresponding bits of two integers. It returns a 1 if at least one of the bits is 1.
1010 | 1101 ➔ 1111

Bitwise XOR (**^**):

The bitwise XOR operator (^) performs a bitwise XOR (exclusive OR) operation on each pair of corresponding bits of two integers. It returns a 1 if the bits are different, and 0 if they are the same.
1010 ^ 1101 ➔ 0111

Bitwise operators

Bitwise NOT (**~**):

The bitwise NOT operator (~) inverts the bits of an integer. It changes 0s to 1s and 1s to 0s.
~1010 ➔ 0101

Left Shift (**<<**):

The left shift operator (<<) shifts the bits of an integer to the left by a specified number of positions filling vacated bits with zero. Bits that are shifted out of the left side are discarded. It effectively multiplies the number by 2 raised to the power of the shift count.
1010 << 2 ➔ 101000

Bitwise operators

Right Shift (**>>**):

The right shift operator (>>) shifts the bits of an integer to the right by a specified number of positions. It effectively divides the number by 2 raised to the power of the shift count.

- For unsigned integers, the leftmost bits are filled with zeros.
- For signed integers, the leftmost bits are filled with the sign bit (0 for positive values, 1 for negative values) to maintain the sign of the number.

1010 >> 2 ➔ 10

Bitwise operators

Practical Use Cases:

Bitwise operators are commonly used for low-level programming, such as working with hardware registers, data compression, encryption, and image processing.
They are also used to optimize memory usage by packing multiple Boolean flags or options into a single integer, which saves space and speeds up access times.
Important Note:
Be cautious when working with bitwise operators, as incorrect usage can lead to unexpected results or undefined behavior. It's essential to understand how these operators work at the bit level and use them with care.

Input and Output

Input and output are not part of the C language itself.
We shall use the standard library, a set of functions that provide input and output, string handling, storage management, mathematical routines, and a variety of other services for C programs.
The library implements a simple model of text input and output. A text stream consists of a sequence of lines; each line ends with a newline character.
The simplest input mechanism is to read one character at a time from the standard input, normally the keyboard, with **getchar**:

int getchar(void)

Input and Output

getchar returns the next input character each time it is called, or EOF when it encounters end of file. The symbolic constant EOF is defined in <stdio.h>.

In many operating systems, a file may be substituted for the keyboard by using the < convention for input redirection: if a program program uses getchar, then the command line *program <infile* causes program to read characters from infile instead of standard input.

The input may come also from another program via a pipe mechanism: on many systems, the command line *otherprogram | program* runs the two programs otherprogram and program and pipes the standard output of otherprogram into the standard input for program.

Input and Output

The function

int putchar(int)

is used for output: putchar(c) puts the character c on the standard output, which is by default the screen. putchar returns the character written, or EOF is an error occurs. Again, output can usually be directed to a file with *>filename*: if program uses **putchar**,

program >outfile

will write the standard output to outfile instead. If pipes are supported,

program | anotherprogram

puts the standard output of program into the standard input of anotherprogram.

Input and Output

Each source file that refers to an input/output library function must contain the line

#include <stdio.h>

before the first reference. When the name is bracketed by < and > a search is made for the header in a standard set of places (for example, on UNIX systems, typically in the directory /usr/include).

An example, considering the program lower, that converts its input to lower case.

Input and Output


```
#include <stdio.h>
#include <ctype.h>
main() /* lower: convert input to lower case*/
{
    int c;
    while ((c = getchar()) != EOF)
        putchar(tolower(c));
    return 0;
}
```

The function tolower is defined in <ctype.h>; it converts an upper case letter to lower case and returns other characters untouched.

Input and Output: printf()

``functions'' like getchar and putchar in <stdio.h> and tolower in <ctype.h> are often macros, thus avoiding the overhead of a function call per character.
The output function **printf** translates internal values to characters. The declaration is

*int printf(char *format, arg1, arg2, ...);*

printf converts, formats, and prints its arguments on the standard output under control of the format. It returns the number of characters printed.

Input and Output: scanf()

The function **scanf** is the input analog of printf, providing many of the same conversion facilities in the opposite direction. The declaration is

*int scanf(char *format, ...);*

scanf is a standard library function used to read formatted input from the standard input (typically the keyboard). It is part of the <stdio.h> library. scanf reads characters from the standard input, interprets them according to the specification in format, and stores the results through the remaining arguments.

Input and Output: scanf()

- Parameters:
  - **format**: A string that specifies the format of the input to be read (e.g., %d for integers, %f for floats, etc.)
  - **....**: The addresses of variables where the input values will be stored (pointers to the variables).

- Return value:
  - The number of successfully read and assigned input items (arguments). If input fails, scanf returns EOF.

Input and Output: scanf()

- Common Format Specifiers:
  - %d Reads an integer
  - %f Reads a float
  - %e Reads a double
  - %c Reads a single character
  - %s Reads a string (stops at whitespace)
  - %x Reads a hexadecimal integer