# A Tutorial Introduction

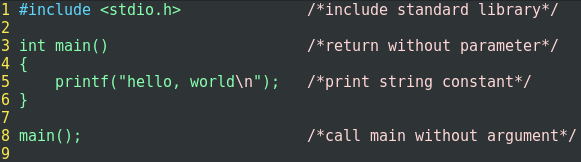
## Getting Start

The only way to learning new program is by writing program in it.

A c program whatever its size consists of ***functions*** and ***variables***. A function contains statements that specify the computing operations to be done, and variables store values used during the computation.

C program always start from the ***main*** function, main will call other functions to perform its job, some of the functions you wrote and some of them are from ***libraries***. One method of communicating data between functions is for the calling function to provide a list of values, called ***arguments***, to the function it calls.

### Print Help Function



### Main Factors of Function

1. Include Information
2. Function Parameters and Arguments
3. Function Return Type，any programming language has return type
4. Statements
5. Main function

## Variables and Arithmetic Expressions

The next program uses the formula: C = (5/9) \* (F-32) to print a Fahrenheit and Celsius table.

|  |
| --- |
| 0 -17  20 -6  40 4  60 15  80 26  100 37 |

In this program, it introduces several new concepts, including comments, declaration, variables, arithmetic expressions, loops and formatted output.

|  |
| --- |
|  |

A ***comment***, which in this case briefly explain what the program does and what the variables does. Any code between /\* and \*/ are ignored by compiler.

In C all variables must be declared before they are used, usually at the beginning of the function before any executable statements. A ***declaration*** announces the properties of the variables, it consists of type name and a list of variables.

C provides several other basic data types besides int and float, including:

* char character – a single byte
* short short integer
* long long integer
* double double-precision floating point
* float

The reason for multiplying by 5 and then divided by 9 is that in C, integer division truncates: any fractional part is discarded.

This example also shows a bit more about how ***printf*** works.

|  |  |
| --- | --- |
| %d | print as decimal integer |
| %6d | print as decimal integer with width 6 |
| %f | print as floating point number |
| %6f | print as floating point number with width 6 |
| %.2f | print as floating point number with 2 bits after point |
| %6.2f | print as floating point number with width 6 and 2 bits after point. |
| other | %s, %o, %x and % |

### Exercise

1. Write program to deal with Fahrenheit to Celsius conversion, and print conversion table
2. Write program to deal with Fahrenheit to Celsius conversion, and print conversion table

The formula is: *Celsius = 5/9 \* (Fahrenheit – 32)*

## Symbolic Constant

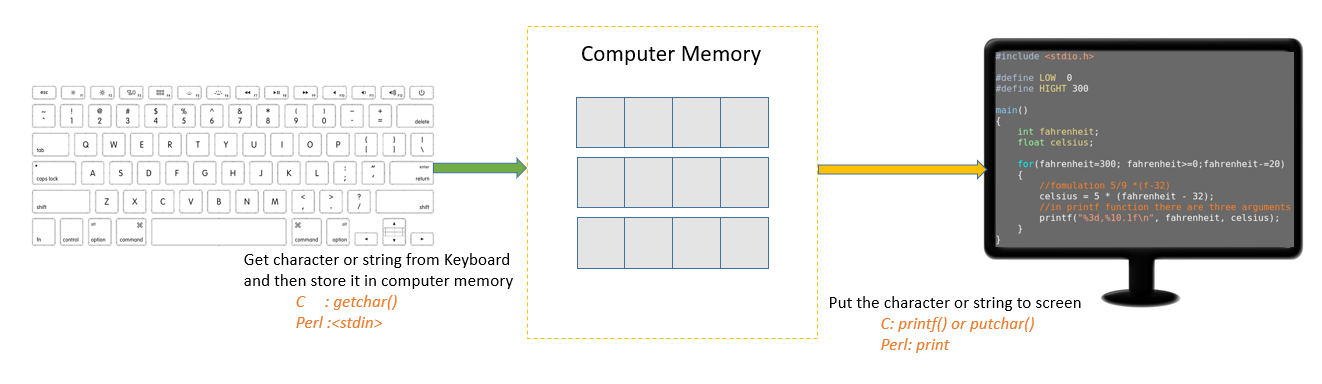
It’s bad to burry “magic number” like 300 and 20 in a program, they convey little information to someone who might have to read the program, and they are hard to change in a systematic way. One way to deal with the magic number is to give them meaningful name. A ***#define*** line define a ***symbolic name*** or ***symbolic constant***, to be a particular string of characters.

***#define name replacement text***

Thereafter any occurrence of ***name*** will be replaced by ***replacement text***, the replacement text can be any sequence of characters, not limited to numbers.

Symbolic constant names are conventionally written in Uppercase, so they can be easily distinguished from lower case variables.

## Character Input and Output



Text input and output regardless where it originates or where it goes to, is dealt with stream of characters. A ***text stream*** is a sequence of characters divided into lines. Each line consists of zero or more characters followed by a ***new line*** character. It is the responsibility of the library to make each input or output stream conform to this model. The C programmer using the library need not to worry about how lines are represented outside the program.

The standard library provides several functions for reading or writing one character at a time, of which ***getchar*** and ***putchar*** are the simplest. Each time it is called, getchar reads the next input character from a text stream and return that as its value. That is, after

c = getchar()

the variable c contains the next character of input. The function putchar print a character each time it is called:

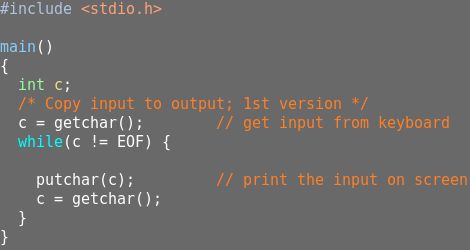
putchar(c)

now the variable c contains the integer variable of c as a character, usually on the screen.

### File Copying

Given getchar and putchar, you can write a surprising amount of useful code without knowing anything more about input and output. Here is an example:

|  |
| --- |
| *read a character from keyboard*  *while (charater is not end-of-file character)*  *output the character just read*  *read another charater* |

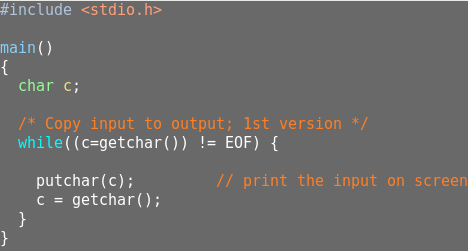


The type char is specifically meant for storing such character data, but any integer type can be used, we use int for a subtle but important reason. The problem is distinguishing the end of the input from valid data. The solution is that getchar returns a distinctive value when there is no more input, a value that cannot be confused with any real character. This value called EOF, for “end of file”, we must declare c to be a type big enough to hold any value that getchar returns. This is why we cannot use char.

This program would be written more concisely by experienced programmer. In C, any assignment, such as

c = getchar()

is an expression and has a value, which is the value of the left hand side after the assignment. This means that an assignment can appear as part of a larger expression.



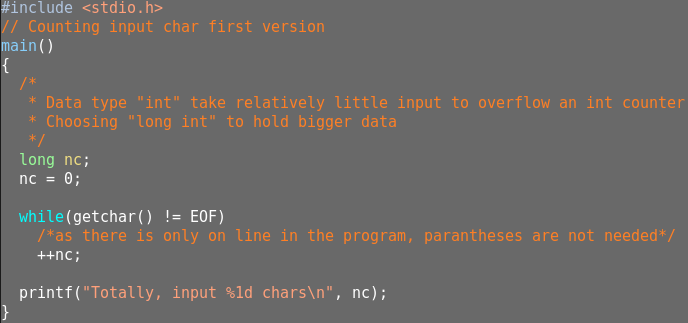
The not equal to sign “!=” has a higher precedence than equal to sign “=”, this is why we must put a parentheses around c=getchar().

### Character Counting

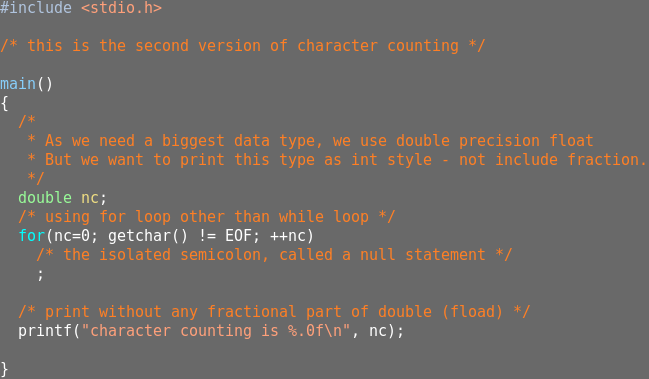
In this chapter, you should remember the following:

1. The value of EOF is -1
2. Double is the abbreviation of double precision float number, so the double and float support %f
3. Int is 16 bit, char is 8 bit, and float is 16 bit, double is 32 bit;
4. The null statement is expressed as “;”

The first version we are using long to hold bigger data, and using auto increment to hold the character counting.

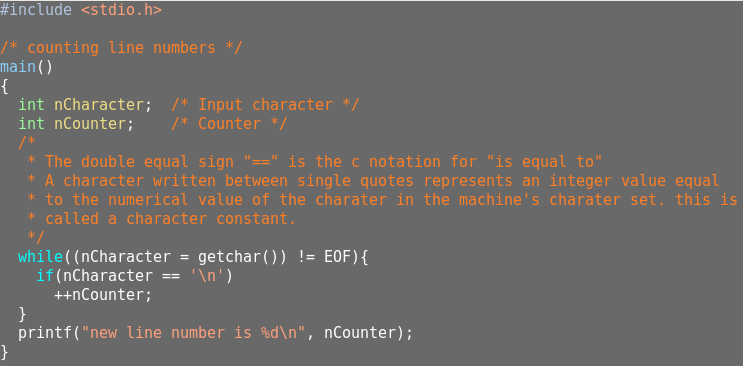


Maybe we need more big data type to hold even more bigger data, then we choose “***double***” or “***double precision float***”. In that case, we do not want to print any fractional part.



### Line Counting

In the following program, we counting the line numbers, as we mentioned above, the standard library ensures that an input text stream appears as a sequence of lines, each terminated by a newline. Hence, counting lines is just counting newlines.

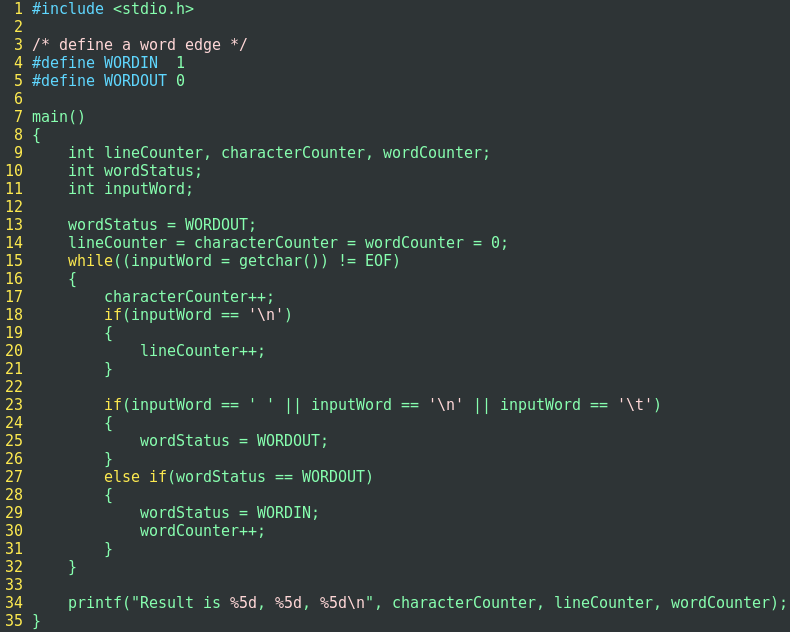


Character constant, for example, ‘A’ is a character constant, in ASCII character set its value is 65. The escape sequence used in string constants are also legal in character constant. So ‘\n’ stands for the newline character, which is 10 in ASCII set.

### Word Counting

The last example is that input a line stream, then count the number of lines, words and characters. Here is the pseudo code:

|  |
| --- |
| 1. input a character 2. if the character is not equal to EOF, the counter the character 3. if the input is a word, it need two conditions    1. start with a character not equal to ‘ ‘, ‘\t’ or ‘\n’    2. stop with a character equal to ‘’, ‘\t’ or ‘\n’ |



### Exercise

1. Write a program to count the blanks, tabs and new lines
2. Write a program to copy its input to output, replace each string of one or more blanks with one blank.
3. Write a program to copy its input to output, replace each tab by \t, back space by \b and back slash by \\, this makes the tabs and backspace visible in an unambiguous way

## Arrays

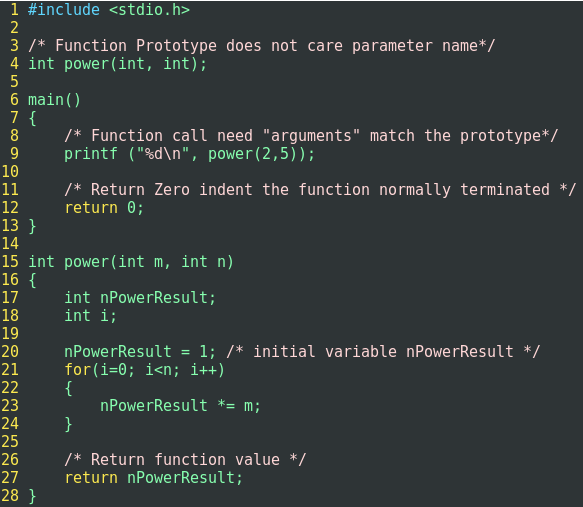
## Functions

A function provides a convenient way to encapsulate some computation which can then be used without worrying about its implementation. with proper designed function, it is possible to ignore how a job is done, just knowing what is done is sufficient.

Function definitions can appear in any order, and in one source file or several, although no function can be split between files. if the source program appears in several files you may have to say more to compile and load it than if it all appears in one, but that is an operating system matter, not a language attribute. The key element of a function as follows:

* Function Declaration or function prototype, the declaration, which is called a function prototype, has to agree with the definition and uses of it. It is an error if the definition of a function or any uses of it do not agree with its prototype. Parameter name in a prototype is usually not needed or need not agree.
* Function Definition
* Function Call
* Return type
* Return Value
* Parameters when define a function
* Argument when call a function

Usually there is a return statement at the end of main function, since main function like any other, it may return a value to its caller which is in effect the environment in which the program was executed. Typically, a return value of zero implies normal termination; non-zero values signal unusual or erroneous termination conditions. For the interest of simplicity, we sometimes may omit return statements from the main function up to this point, but we will include them hereafter, as a reminder that programs should return status to their environment.



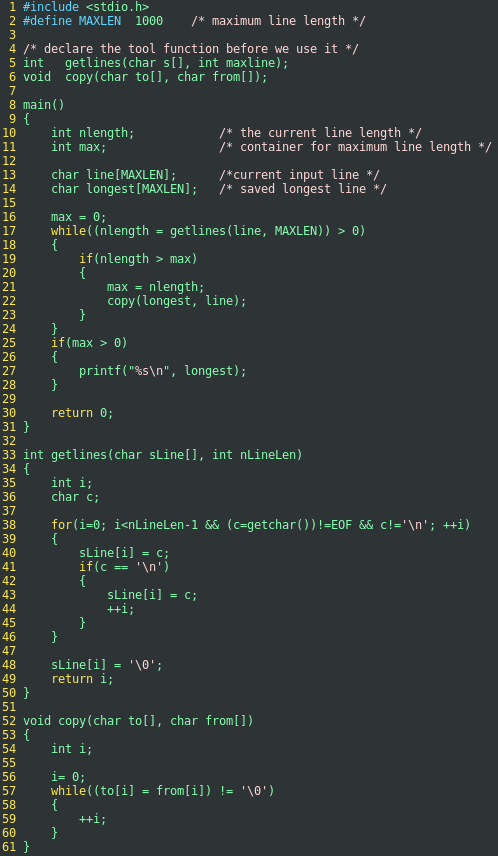
## Argument – Call by Value

In C, all function arguments are passed “by value”, that means that the called function is given the values of its arguments in temporary variables rather than the originals. This lead to some different properties than are seen with “call by reference” languages.

## Character Array

The most common type of array is the array of character. Example

***Input a line stream, calculate the longest line and print the line as well as the line length***:



* the function getlines and copy are declared at the beginning of the function;
* main and getlines are communicated with each other through a pair of arguments and a return value.
* the purpose of supplying the size of an array in a declaration is to set aside storage.
* the length of the array of s is not necessary in getlines since its size is set in main.
* getlines uses return to send a value back to the caller.
* Copy is void which state explicitly that no value is returned.
* getlines put a character ‘\0’ (***the null character, whose value is zero***)

## External Variable and Scope

Each local variable in a function comes into existence only when the function is called, and disappears when the function is existed. This is why such variable is called ***automatic*** variable. Because the automatic variable come and go with function invocation, they do not retain their value from one call to the next, and must be explicitly set upon each entry. if they are not set, they will contain garbage.

As an alternative to automatic variables, it is possible to define variables that are ***external*** to all functions. Because external variables are globally accessible, they can be used instead of argument lists to communicate data between functions.

An external variable must be ***defined***, exactly once, outside of any function; this sets aside storage for it, the variable must also be ***declared*** in each function that want to access it. The declaration may be an explicit ***“extern”*** statement or may be implicit from context.

If the definition of an extern variable occurs in the source file before its use in a particular function, then there is no need for an extern declaration in the function.

If the program is in several source files and a variable is defined in ***file1*** and used in ***file2*** and ***file3***, then extern declarations are needed in ***file2*** and ***file3*** to connect the occurrences of the variable. The usual practice is to collect extern declaration of variables and functions in a separate file, historically called ***header***, that is included by ***#include*** at the front of each source file. the suffix ***.h*** is conventions of variables for each header names. the functions of the standard library, for example, are declare in the headers like ***<stdio.h>***.

“***Definition***” refers to the place where the variable is created or assign storage; “***Declaration***” refers to places where the nature of the variable is stated but no storage is allocated.

## Practices

# Types, Operators and Expressions

***Variables and Constant*** are the basic data objects manipulated in a program. ***Declarations*** list the variables to be used and state what type they have and perhaps what their initial values are. ***Operators*** specify what is to be done to them, ***Expressions*** combine variables and constants to produce new values.

## Variable Name

## Data Type and Sizes

There are only a few basic data type in c:

|  |  |
| --- | --- |
| char | a single byte, capable of holding one character in the local character set |
| int | an integer, typically reflecting the natural size of integers on the host machine. |
| float | single-precision floating point |
| double | double-precision floating point |

In addition, there are a number of qualifiers that can be applied to these basic types. ***short*** and ***long*** to integers.

|  |
| --- |
| short int sh;  long int counter; |

the word int can be omitted in such declarations and typically is.

the intent of short and long should provide different lengths of integers where practical; int normally be the natural size for a particular machine. short is often 16 bit, long 32 bits and int either 16 bits or 32 bits. *Each compiler is free to choose appropriate sizes for its own hardware, subject only to the restriction that shorts and ints are at least 16 bits and long are at least 32 bits*.

the qualifier ***signed*** and ***unsigned*** may be applied to char or any integer.

The standard header ***<limits.h> and <float.h>*** contain symbolic constants for all of these sizes along with other properties of the machine and compiler.

## Constants

An integer constant like 1234 is an int. A long constant is written with a terminal l or L, as in 123456789l(L), an integer too big to fit into an int will also be taken as a long. Unsigned constants are written with a terminal u or U, and the suffix ul or UL indicates unsigned long.

Floating-point constants contain a decimal point (123.4) or an exponent(1e-2) or both, their type is double, unless suffixed. the suffixes f or F indicate a float constant, l or L indicate a long double.

The value of integer can be specified in octal, or hexadecimal instead of decimal. A leading 0(Zero) on an integer constant means octal; a leading 0x or 0X means hexadecimal. For example, decimal 31 can be written 037 in octal, or 0x1f in hex. Octal or hexadecimal can also be followed by L to make them long and U to make them unsigned: 0XFUL is an unsigned long constant with value 15 decimal.

A ***character constant*** is an ***integer*** written as on character with ***single quotes***, such as ‘x’, the value of a character constant is the numeric value of the character in the machine’s character set. For example, in the ASCII character set the character constant ‘0’ has value 48, which is unrelated to the numeric value 0.

Certain characters can be represented in character and string constants by escape sequences like \n(newline); these sequence look like two characters, but represent only one.

The complete set of escape sequence is:

|  |  |  |  |
| --- | --- | --- | --- |
| \a | *alert (bell) character* | \\ | *backslash* |
| \b | *backspace* | \? | *question mark* |
| \f | *formfeed* | \’ | *single quote* |
| \n | *newline* | \’’ | *double quote* |
| \r | *carriage return* | \000 | *octal number* |
| \t | *horizontal tab* | \xhh | *hexadecimal number* |
| \v | *vertical tab* |  |  |

The character constant ‘\0’ represents the character with value zero, the null character. ‘\0’ is often written instead of 0 to emphasize the character nature of some expression, but the numeric value is just 0;

A ***constant expression*** is an expression that involves only constants. Such expression may be evaluated during compilation rather that run-time, and accordingly may be used in any place that a constant can occur, as in:

|  |
| --- |
| *#define MAXLINE 1000*  *char line[MAXLINE+1]* |

A ***string constant*** or ***string literal***, is a sequence of zero or more characters surrounded by double quotes as in

|  |
| --- |
| “I am a string” or  “” */\* the empty string\*/* |

The quotes are not part of the string, but serve only to delimit it. The same escape sequences used in character constant apply in strings; \” represent the double-quote character. String constants can be concatenated at compile time:

|  |
| --- |
| *“hello,” “world” is equal to*  *“hello, world”* |

Technically, a *string constant* is an array of characters, the internal representation of a string has a null character ‘\0’ at the end, so the physical storage required is one more than the number of characters written between the quotes. This representation means that there is no limit to how long a string can be, but programs must scan a string completely to determine its length. The standard library function ***strlen(s)*** return the length of its character. Function strlen and other string functions are declared in the standard header *<string.h>*.

Be careful to distinguish between a character constant and a string that contains only a single character: ‘x’ is not the same as “x”, ‘x’ is a character constant will produce a numeric number, but “x” is an array of characters that contains one character and a null(‘\0’);

There is one other kind of constant, the ***enumeration constant***. An enumeration is a list of constant integer values, as in

|  |
| --- |
| *enum Boolean { NO, YES}* |

The first name in an enum has value 0, the next 1, and so on, unless explicit values are specified. If not all values are specified, unspecified values continue the progression from the last specified value, as in the second example:

|  |
| --- |
| *enum escapes { BELL = ‘\a’, BACKSPACE = ‘\b’, TAB = ‘\t’, NEWLINE = ‘\n’}* |
| *enum months { JAN = 1, FEB, MAR, ARP, MAY, JUN, JUL } /\* FEB is 2, and so on \*/* |

Enumeration provide a convenient way to associate constant values with names, and alternative to #define with advantage that the values can be generated for you automatically.

## Declarations

*All variables must be declared before use*, although certain declarations can be made implicitly by context. A declaration specifies a type, and contains a list of one or more variables of that type. as in:

|  |
| --- |
| *int lower, upper, step;*  *char c, line[1000];* |

A variable may also be initialized in its declaration, if the name is followed by an equals sign and an expression, the expression serves as an *initializer*, as in:

|  |
| --- |
| *char esc = ‘\\’;*  *int I = 0;*  *float eps = 1.0e-5* |

The External and static variables are initialized to zero by default, automatic variables for which there is no explicit initializer have undefined values.

The qualifier ***const*** can be applied to the declaration of any variable to specify that its value will not be changed. For an array, the const qualifier says that the elements will not be altered:

|  |
| --- |
| *const double e = 2.71828182845905;*  *const char msg[] = “warning:”;* |

The const declaration can also be used with array arguments to indicate that the function does not change that array:

|  |
| --- |
| *int strlen(const char []);* |

## Arithmetic Operators

The binary arithmetic operators are ***+, -, \*, /*** and the modules operator ***%***. The % operator cannot be applied to float or double, the direction of truncation for / and the sign of the result for % are machine-dependent for negative operands.

the binary + and – operators have the same precedence, which is lower that the precedence of \*, / and %.

## Relational and Logical Operators

The Relational operators are

|  |
| --- |
| < > <= and >= |

They all have the same precedence, just below them in precedence are the equality operator:

|  |
| --- |
| == and != |

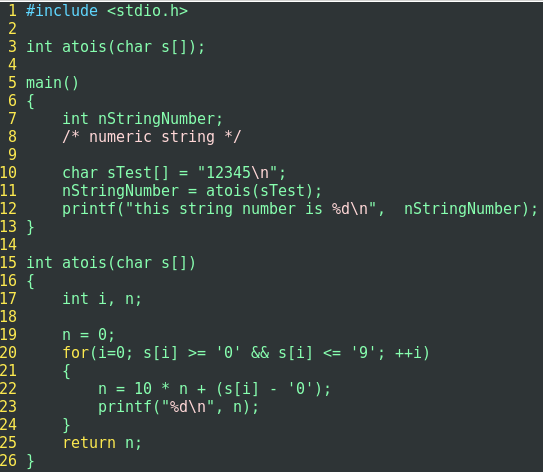
Relational operators &&(and) and ||(or) have lower precedence than arithmetic operators.

## Type Conversions

When an operator has operands of different types, they are converted to a common type according to a small number of rules. In general, the only automatic conversions are those that convert a “narrower” operand into a “wider” one without losing information, such as converting an integer to floating point in an expression like f + I;

*A char is just a small integer, so char may be freely used in arithmetic expressions*. This permits considerable flexibility in certain kinds of character transformation.

A typical function called atoi (ASCII to integer), which convert a string of digits into its numeric equivalent.



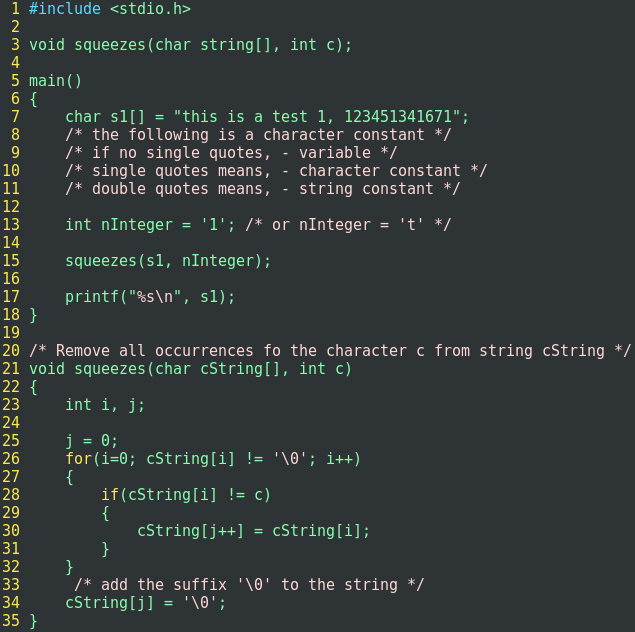
The standard header ***<ctype.h>***, defines a family of functions that provide tests and conversions that are independent of character set.

Implicit arithmetic conversions work much as expected, in general, if an operator like + or \* that takes two operands, has operands of different types, the “***lower***” type is ***promoted*** to the “***higher***” type before the operation proceeds, the result is of the higher type. If there are no unsigned operands, however the following information set of rules will suffice:

* If either operand is long double, convert the other to long double.
* otherwise, if either operand is double, convert the other to double.
* otherwise, if either operand is float, convert the other to float.
* otherwise, convert char and short to int.
* then, if either operant is long, convert the other to long.

## Increment and Decrement Operator

There are increment and decrement operator, ++ and --, they can be used as prefix or suffix. ++n expression will add its value by 1 ***before*** its value is used, and n++ expression will add its value by 1 ***after*** its value has been used.



## Bitwise Operator

C provides six operators for bit manipulation, these may only be applied to integral operands, that is, char, short, int and long, whether signed or unsigned.

|  |  |
| --- | --- |
| & | bitwise and |
| | | bitwise inclusive or |
| ^ | bitwise exclusive or |
| << | left shift |
| >> | right shift |
| ~ | one's complement |

The bitwise AND operator & is often used to mask off some set of bits, for example,

|  |
| --- |
| *n = n & 0177 sets to zero all but the low-order 7 bits of n.*  *0177 is octal format, binary is 1111111* |

The bitwise OR operator | is used to turn bits on:

|  |
| --- |
| *x = x | set\_on; sets to one in x the bits that are set to one in set\_on* |

The bitwise exclusive OR operator ^ sets a one in each bit position where its operands have different bits, and zero where they are the same.

One must distinguish the bitwise operator & and | from the logical operators && and ||, which imply left to right evaluation of a truth value.

The shift operator << and >> perform left and right shift of their left operand by the number of bit positions given by the right operand, which must be positive, thus, x << 2 shift

the value of x left by two positions, filling vacated bits with zero, this is equivalent to multiplication by 4. right shifting signed quantity, will fill with sign bits on some machines and with 0-bits on others

the unary operator ~ yields the one's complement of an integer, that is, it converts each 1-bit into a 0-bit and vice versa, for example:

x =x & ~077

sets the last six bits of x to zero.

*Example, we assume bit 0 position is at most right, write a program, which return n bits from position m* *for unsigned number x*.

## Assign Operators and Expression

Most binary operators have a corresponding assignment operator op=, where op is one of

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| + | - | \* | / | % | << | >> | & | ^ | | |

If *expr1* and *expr2* are expressions, then:

expr1 ***op=*** expr2 is equivalent to expr1 = (expr1) op (expr2)

## Conditional Expressions

The Conditional Expression, written with the ternary operator “?:”. In the expression:

|  |
| --- |
| *expression 1 ? expression 2: expression 3* |

the expression, expression 1 is evaluated first, if it is non-zero(true), then expression 2 is evaluated, and that is the value of the conditional expression. Otherwise expression 3 is evaluated and that is the value of the conditional expression.

|  |
| --- |
| z = (a>b) ? a : b; /\* z=max(a,b) \*/ |

## Precedence and Order of Evaluation

|  |  |  |
| --- | --- | --- |
| **Category** | **Operator** | **Associativity** |
| Postfix | () [] -> . | Left to right |
| Unary | + - ! ~ \* & *++* *--* sizeof | Right to left |
| Multiplicative | \* / % |  |
| Additive | + - |  |
| Shift | << >> |  |
| Relational | < > <= >= |  |
| Equality | == != |  |
| Bitwise AND | & |  |
| Bitwise XOR | ^ |  |
| Bitwise OR | | |  |
| Logical AND | && |  |
| Logical OR | || |  |
| Conditional | ?: |  |
| Assignment | = += -= /= \*= %= <<= >>= &= ^= |= |  |
| Comma | , |  |

# Control Flow

## Statements and Blocks

Expressions such as x = 0 or i++ or printf(…) becomes a statement when it is followed by a semicolon, as in:

|  |
| --- |
| *x = 0;*  *i++;*  *printf();* |

In C, semicolon is a statement terminator, rather than a separator as it is in other language.

Braces {and} are used to group declarations and statements together into a compound statement, or block, so that they are syntactically equivalent to a single statement.

## If-Else

Since an if simply test the numeric value of an expression, certain coding shortcuts are possible. The most obvious is writing:

|  |
| --- |
| *if(expression)* |

instead of

|  |
| --- |
| *if (expression != 0)* |

## Else-If

*if (expression 1)*

*statements;*

*else if (expression 2)*

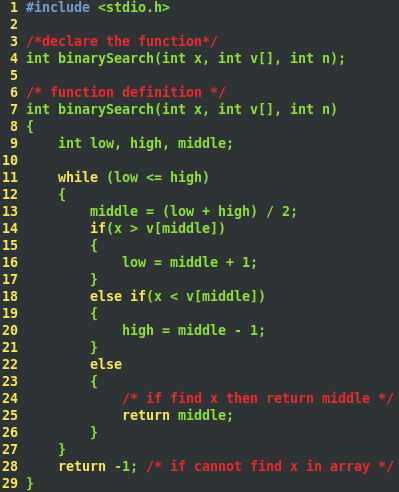
*statements;*

*else if (expression 3)*

*statements*

*else*

To illustrate the three-way decision, here is a binary search function that decide if a particular value x occurs in stored array v, the element v must be in an increasing order.



## Switch

The switch statement is a multi-way decision, that test whether an expression matches one of a number of constant integer values, and branch accordingly.

|  |
| --- |
| *switch (expression){*  *case constant-expression: statements;*  *case constant-expression: statements;*  *default: statements;*  *}* |

Input characters from keyboard continuously, write a program to count the number of each digits, white spaces and other characters.



## Loops – While and For

We have already encounter the while and for loops in previous chapter, In while loop

*while (expression)*

*statements;*

First of all, the expression is evaluated, if ***non-zero***, then statement is executed and re-evaluate the expression, the cycle continuous, until expression becomes ***zero***. at which point, executions resume after the statements.

The for loop:

|  |
| --- |
| *for (expression 1; expression 2; expression 3)*  *statement;* |

is equal to while loop:

|  |
| --- |
| *expression 1;*  *while (expression 2)*  *statement;*  *expression 3;* |

Grammatically, the three part of for loop are expressions, most commonly, expression 1 and expression 3 are assignment or function calls and expression 2 is a relational expression. Any of the three parts can be omitted, it is simply dropped from expansion. if expression 2 is not present, it is taken as permanently true. So ***for (; ;)*** is an infinite loop.

Whether to use for or while is largely a matter of personal preference. The for is preferable when there is a simple initialization and increment, since it keeps the loop control statements close together and visible at the top of the loop.

## Loops – Do-while

As we discussed in Chapter 1, the while and for loops test the termination condition at the top. By contrast, the loop do-while, test the termination at the bottom after making each pass through the loop body, that is the body is always executed at least once.

The syntax of the do is:

|  |
| --- |
| *do*  *statement;*  *while (expression);* |

## Break and Continue

It is sometimes convenient to be able to exit from a loop other than by testing the top or bottom, the ***break*** statement provides an early exit from for, while and do, just at from switch. A break causes the innermost enclosing loop or switch to be exited immediately.

## Goto and Labels

# Function and Program Structure

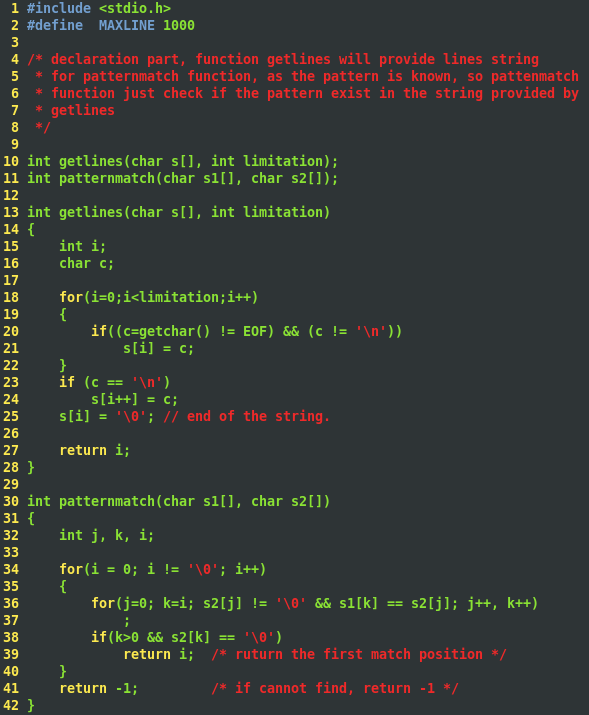
## Basics of Function

To begin, let us design a program to print each line of its input, that contains a particular “pattern” or string of characters, for example search for pattern of letters of “could” in each set of lines:

|  |
| --- |
| *we could do that by tomorrow*  *find a world match the pattern*  *Maybe you could do that by a c programming* |

print the lines include “could”, the job will fall neatly into three part:

|  |
| --- |
| *while (there is another line)*  *if (the line contains pattern)*  *print the line* |



Generally speaking, a function includes:

|  |
| --- |
| *function\_type function\_name (argument declarations)*  *{*  *declarations and statements*  *}* |

various part maybe absents, a minimal function as: dummy () { }, this function return nothing and need nothing.

A program is just a set of definitions of variables and functions, communication between functions are arguments and values returned by function, and through external variables. The functions can occur in any order in the source file, and source file can be split into several files, so long as no function is split.

The return expression is the mechanism for returning a value from the called function to its caller.

On Linux system suppose there are three functions stored in three files, called main.c, getlines.c and patternmatch.c, with gcc command compile the three file, it places the resulting object code in file main. o, getlines. o and patternmatch.c, then load them into one executable file a.out

## Function Return None-Integers

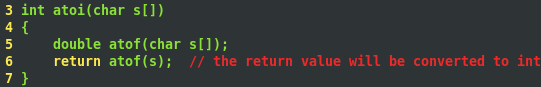
the Function default return type is integer, if the return will not be an integer, then you must declare the return type in function declare expression.

The calling function must know the called function’s return type, one way to ensure this is to declare the called function explicitly, in the calling routine. Example of declaration as follows:

|  |
| --- |
| *double sum, atof(char s[])* |

it says that the sum is a double type and function atof take one char s[] argument and return double data type. One thing need to mention is that the function atof must be declared and defined consistently, if the declaration and definition is inconsistent, then the compiler will report error.

If there is no prototype for a function, a function is implicitly declared by its first appearance in an expression. If there is no function name has been previously declared, and followed by a left parenthesis, then it is declared by context to be a function name. If a function takes no arguments, then all parameter check turns off by the compiler. If function takes arguments, then declare them, if takes no arguments, then use void.



## External Variables

External variables are defined outside any functions, and are thus potentially available to many functions. Functions themselves are always external, because c does not allow function to be defined inside any function.

Because external variables are globally accessible, they provide an alternative to function arguments and return values for communicating data between functions, any function may access an external variable by referring to it by name, if the name has been declared somehow.

If a large number of variables must be shared among functions, external variables are more convenient and efficient than long argument lists. External variables are also useful because of their greater scope and lifetime. Automatic variables are internal to a function, they come into existence when the function is entered and disappear when it is left. External variables, on the other hand, are permanent, so they retain values form one function invocation to the next. An example to calculate the following equation:

|  |
| --- |
| *(1-2) \* (4+5)* |

but in the real practice we only input:

|  |
| --- |
| *1 2 – 4 5 + \** |

Parentheses are not needed, the notation is unambiguous, as long as how many operands each operator required. The implementation is simple, each operand is pushed onto a stack, when an operator is arrived, the proper number of operands is popped, the operator is applied to them and the result is pushed back onto the stack. In the example above for example, 1 and 2 are pushed, when “– “comes, the result -1 is pushed again, next, push 4 and 5 onto the stack, when + comes, 4 and 5 have been popped and result 9 should be pushed onto the stack, when \* comes, pop -1 and 9 apply \* and push the result onto the stack.

From above analysis, we write pseudo code as follows:

|  |
| --- |
| *when (next operand or operator comes and they are not end of file mark)*  *if(number)*  *push it to stack*  *else if(operator)*  *pop number from stack*  *do operation*  *push result to stack*  *else if(new line)*  *pop result and print it*  *else*  *error happens* |

## Scope Rules

The scope of a name is part of the program within which the name can be used, for an automatic variable declared at the beginning of a function, the scope is the function in which the name is declared. Local variables of the same name in different functions are unrelated, the same is true of the parameters of the function, which are in effect local variables.

The scope of an external variable or a function lasts from the point at which it is declared to the end of the file being compiled. For example, if main, sp, val, push and pop are defined in one file, in the order of:

|  |
| --- |
| *main () { …}*  *int sop = 0;*  *double val[MAXVAL];*  *void push(double f)*  *double pop(void);* |

then the variables sp and val may be used in push and pop simply by naming them; no further declarations are needed, but these names are not visible in main, nor are push and pop themselves.

On the other hand, if an external variable is to be referred to before it is defined, or if it is defined in a different source file from the one where it is being used, then an ***extern*** declaration is mandatory.

It is important to distinguish between the declaration of an external variable and its definition. A declaration announces the properties of a variable, a definition also causes storage to be set aside. If lines:

|  |
| --- |
| *int tsp;*  *double val[MAXVAL]* |

outside any functions, they ***define*** the external variable tsp and val, cause storage to be set aside, and also serve as declaration for the rest of the source file. On the other hand, the lines:

|  |
| --- |
| *extern int tsp;*  *extern double val[MAXVAL]* |

***declare*** for the rest of the source file that tsp is an int and val is a double array, but they do not create the variable or reserve storage for them.

There must be only one definition of an external variable among all the files that make up the source program; other files may contain ***extern*** declarations to access it. Array size must be specified with the definition, but are optional with an extern declaration. Initialization of an external variable goes only with the definition.

To illustrate the mechanism of how external variable works, let’s write the following program:

|  |
| --- |
| In file 1:  *extern in tsp;*  *extern double val[];*  *void push(double f);*  *double pop(void);*  In file 2:  *int sp = 0;*  *double val[MAXVAL];* |

Because the extern declarations in file 1 lie ahead of and outside the function definitions, they apply to all functions; one set of declarations suffices for all of file 1.

## Head Files

## Static Variables

The variables sp and val in stack.c and buf and bufp in getch.c, are for private use of in their respective source file and are not meant to be accessed by anything else. But the current issue is if we add ***extern*** to them in other files we can access it. The ***static*** declaration, applied to an external variable or function, limits the scope of that object to the rest of the source file being compiled. External static thus provides a way to hide names like buf and bufp in the getch-ungetch combination, which must be external so they can be shared, yet which should not be visible to users of getch and ungetch.

Static storage is specified by prefixing the normal declaration with the word static. If the two routines and the two variables are compiled in one file, as in:

|  |
| --- |
| *static char buf[BUFSIZE]; /\* buffer for ungetch \*/*  *static int bufp = 0; /\* next free position in buf \*/* |

There is no other routine will be able to access buf and bufp, and those names will not conflict with the same names in other files of the same program. In the same way, the variables that push and pop use for stack manipulation can be hidden, by declaring sp and val to be static.

The external static declaration is most often used for variables but it can be applied to function and internal variables as well. Normally, function names are global, visible to any part of the entire program. if a function is declared static, however, its name is invisible outside of the file in which it is declared.

The static declaration can also be applied to internal variables, internal static variables are local to a particular function just as automatic variables are, but unlike automatics, they retain the value in existence rather than coming and going each time the function is activated. This means that internal static variables provide private, permanent storage within a single function.

## Register Variables

A register declaration advises the compiler that the variable in question will be heavily used the idea is that register variables are to be placed in machine registers, which may result in smaller and faster programs.

## Block Structure

Variables can be defined in a block-structured fashion within a function. declarations of variables may follow the left brace that introduces any compound statement, not just the one that begins a function. Variables declared in this way hide any identically named variables in outer blocks, and remain in existence until the matching right brace. For example:

|  |
| --- |
| *if (n > 0) {*  *int I; /\* declare a new I \*/*  *for (I = 0; I < n; I++)*  *….* |

The scope of variable I is the true branch of the if; this I is unrelated to any I outside the block. An automatic variable declared and initialized in a block is initialized ***each time*** the block is entered. A static variable is initialized only the first time the block is entered.

Automatic variables, including formal parameters, also hide external variables and functions of the same name. Given the declaration:

|  |
| --- |
| *int x;*  *int y;*  *function (double x)*  *{*  *double y;*  *}* |

then within the function, occurrences of x refer to the parameter, which is a double; outside of function, they refer to the external int. The same is true of the variable y.

## Initialization

In the absence of explicit initialization, external and static variables are guaranteed to be initialized to zero; automatic and register variables have undefined initial values(garbage).

For external and static variables, the initializer must be a ***constant*** expression; the initialization is done once, conceptually before the program begins execution. for automatic and register variables, it is done each time the function or block is entered. For automatic and register variables, it is done each time the function or block is entered.

For automatic and register variables, the initializer is not restricted to being a constant, it may be any expression involving previously defined values, even function calls. In effect, initialization of automatic variables is just shorthand for assignment statement. which form to prefer is largely a matter of taste. We have generally used explicit assignment, because initializers in declarations are harder to see and further away from the point of use.

An array may be initialized by following its declaration with a list of initializer enclosed in braces and separated by commas. for example, to initialize an array “days” with the number of days in each month:

|  |
| --- |
| *int days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }* |

When the size of an array is omitted, the compiler will compute the length by counting the initializers.

If there are fewer initializer for an array than the number specified, the missing elements will be zero for external, static, and automatic variables. It is an error to have too many initializers.

Character arrays are a special case of initialization; a string may be used instead of the braces and commas notation:

|  |
| --- |
| *char pattern[] = “could”*  is equal to pattern:  *char pattern[] = {‘c’, ‘o’, ‘u’, ‘l’, ‘d’}* |

## Recursion

## The C Preprocessor

C language provide certain language facilities by means of preprocessor, which is conceptually a separate first step in compilation. The two most frequently used feature are #define and #include.

### File Inclusion

File inclusion makes it easy to handle collections of #defines and declarations. Any source line of a form: “***#include*** *“filename”*” or *“#include <filename>”* is replaced by the content of filename. If it is quoted, search for the file typically begins where the source program was found; if it is enclosed in < and >, search follows an implement-defined rule to the file.

#include is the preferred way to tie the declarations together for a large program. It guarantees that all the source files will be supplied with the same definitions and variable declarations and thus eliminates a particularly nasty kind of bug.

### Macro Substitution

A definition has form:

***#define*** *name “replacement text”*

it calls for a macro substitution of the simplest kind – subsequent occurrences of the token name will be replaced by the replacement text.

Names maybe undefined with ***#undef***, usually to ensure that a routine is really a function, not a macro. For example:

*#undef getchar*

*int getchar(void){}*

Formal parameters are not replaced within quoted strings. If, however, a parameter name is preceded by a # in the replacement text, the combination will be expanded into a quoted string with the parameter replaced by the actual argument. This can be combined with string concatenation to make, for example, a debugging print macro:

*#define dprint(expr) printf(#expr “ = %g\n”, expr)*

when this is invoked, as in “*dprint(x/y)”*

the macro is expanded into

*printf(“x/y” “ = %g\n”, x/y)*

and the strings are concatenated, so the effect is

*printf(“x/y = %g\n”, x/y)*

The preprocessor operator ***##*** provides a way to concatenate actual arguments during macro expansion. If a parameter in the replacement text is adjacent to a ##, the parameter is replaced by the actual argument, the ## and surrounding white space are removed, and the result is re-scanned. For example, the macro paste concatenates its two arguments:

*#define paste (front, back) front ## back*

so paste(name,1) create the token name1.

### Conditional Inclusion

It is possible to control preprocessing itself with conditional statements that are evaluated during preprocessing. This provides a way to include code selectively, depending on the value of conditions evaluated during compilation.

The ***#if*** line evaluates a *constant integer* expression, if the expression is none zero, subsequent lines until an ***#endif*** or ***#elif*** or ***#else*** are included.

For example, to make sure that the contents of a file hdr.h are included only once, the contents of the file are surrounded with a conditional like this:

|  |
| --- |
| *#if !defined(HDR)*  *#define HDR*  */\* contents of hdr.h go here \*/*  *#endif* |

A similar style can be used to avoid including files multiple times. If this style is used consistently, then each header can itself include any other headers on which it depends, without the user of the header having to deal with the interdependence.

This is sequence tests the name SYSTEM to decide which version of a header to include:

|  |
| --- |
| *#if SYSTEM == SYSV*  *#define HDR “sysv.h”*  *#elif SYSTEM == BSD*  *#define HDR “bsd.h”*  *#elif SYSTEM == MSDOS*  *#define HDR “msdos.h”*  *#else*  *#define HDR “default.h”*  *#endif*  *#include HDR* |

the ***#ifdef*** and ***#ifndef*** lines are specialized forms that test whether a name is defined. The first example of #if above could have been written:

|  |
| --- |
| *#ifndef HDR*  *#define HDR*  */\* contents of hdr.h go here \*/*  *#endif* |

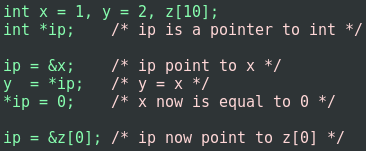
# Pointer and Arrays

## Pointer and Address

A pointer is a variable that contains the address of a variable. A typical machine has an array of consecutively numbered or addressed memory cells that may be manipulated individually or in contiguous groups.

the unary operator ***&*** gives the address of an object, so the statement p = &c assign the address of “c” to “p”, and “p” is said to “point to” variable “c”. 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 point to.



The unary operator \* and & bind more tightly than arithmetic operators, so the assignment:

*y = \*ip + 1*

takes whatever ip points at, adds 1 and assigns the result to y, while *\*ip += 1* increments what ip points to, as do *++\*ip* and *(\*ip)++.* the parentheses are necessary in this last example, without them, the expression would increment ip instead of what it point to, because unary operators like \*, and ++ associate right to left.

Finally, since pointer are variables, they can be used without dereferencing, for example: iq = ip, copies the content of ip into iq, thus make iq point to whatever ip pointed to.

## Pointers and Function Arguments

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

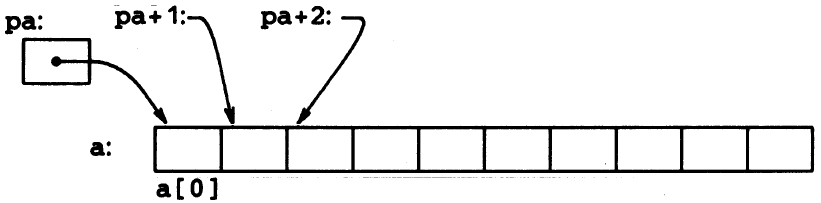
## Pointers and Arrays

In C, there is a strong relationship between pointers and arrays, strong enough that pointers and array should be discuss simultaneously. In fact, any operation can be done with array subscripting, can also be done with pointer. The pointer version will be faster, but somewhat harder to understand. For example:

*int a [10];*

Define an array of size 10, that is, a block of 10 consecutive objects named, a [0], a [1], … a [9]

If pa pointer to a [0] of an array, then pa + 1 will point to a [1], “pa + i” is the address of a [i], and \*(pa + i) is the contents of a[i].



The correspondence between indexing and pointer arithmetic is very close by definition, the value of a variable or expression of type array is the address of element of zero of the array.

Since the name of an array is the synonym of the location of the initial element, so expression

“pa = a” is equal to “pa = &a[0]”, “int \*pa”; Also a[i] is equal to \*(a+i), because of a = &a[0].

On the other hand, if “pa” is a pointer, then “pa[i]” is equal to “pa + i”.

There is a difference between a pointer and an array name need to be kept in mind, that is, a pointer is a variable, but array name is not a variable, so, construct, pa++ is legal, but a++ is not legal.