Operatori

matematici

+	-	*	/	%

relazionali

<	>	<=	>=	==	!=

logici

&&	||	!

Precedenza degli operatori

Operatore	Precedenza
chiamata a funzione	
! + - & (operatori unari)	
* / %	
+ -	
< <= > = >	
== !=	
&&	
||	
=	

Costrutti decisionali

if

if (condizione**)**
  istruzione$_V$;
[**else**
  istruzione$_T$;]
	if (condizione**) {**
	  istruzione$_{V1}$;
	  ...
	  istruzione$_{Vn}$;
	} [**else {**
	  istruzione$_{T1}$;
	  ...
	  istruzione$_{Tn}$;
	}]
		if (condizione$_1$**)**
		  istruzione$_{V1}$;
		else if (condizione$_2$**)**
		  istruzione$_{V2}$;
		...
		else if (condizione$_n$**)**
		  istruzione$_{Vn}$;
		else
		  istruzione$_e$;

switch

switch (espressione**) {**
  case valore$_1$**:** istruzioni$_1$;
    [**break;**]
  case valore$_2$**:** istruzioni$_2$;
    [**break;**]
  case valore$_3$**:** istruzioni$_3$;
    [**break;**]
  ...
  case valore$_n$**:** istruzioni$_n$;
    [**break;**]
  [**default:** istruzioni$_d$;]
}

Syntax

The final stage of preprocessing is to convert all remaining preprocessing tokens in the translation unit to C tokens. The translator then parses these C tokens into one or more declarations. In particular:

- Declarations that define **objects** specify the storage for data that a program manipulates.
- Declarations that are function definitions specify the actions that a program performs.

You use expressions in declarations to specify values to the translator or to specify the computations that the program performs when it executes. This document shows the forms of all C tokens. It also summarizes the syntax of declarations, function definitions, and expressions. You use these syntactic forms, with preprocessing directives and macros, to write a C program.

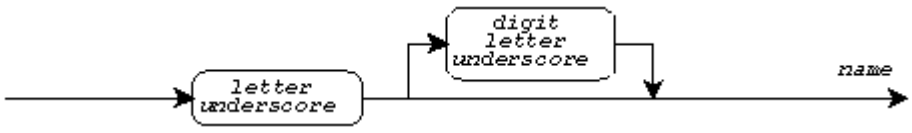
C Tokens

Each C token derives from a preprocessing token. Additional restrictions apply, however, so not all preprocessing tokens form valid C tokens. You must ensure that only valid C tokens remain in the translation unit after preprocessing.

Every preprocessing name forms a valid C token. Some of the names that you write are **keyword** C tokens (names that have special meaning to the translator). The following Table lists all defined keywords:

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

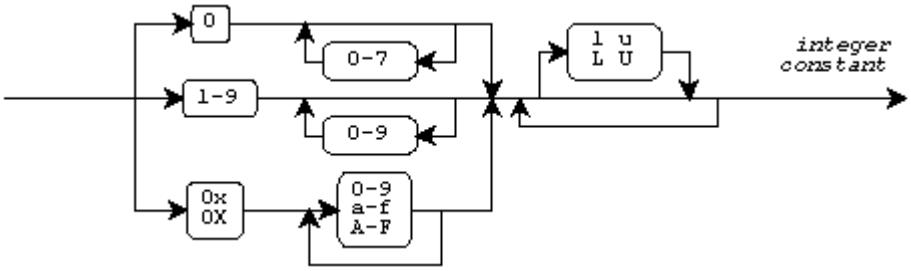
A **name** C token is a preprocessing name that is not a keyword:



You must ensure that distinct names with external linkage differ within the first six characters, even if the translator does not distinguish between lowercase and uppercase letters when comparing names.

Integer and Floating-Point Constants

Every preprocessing number in the translation unit must be either an **integer constant** or a **floating-point constant** C token. An integer constant is a preprocessing number that represents a value of an integer type:



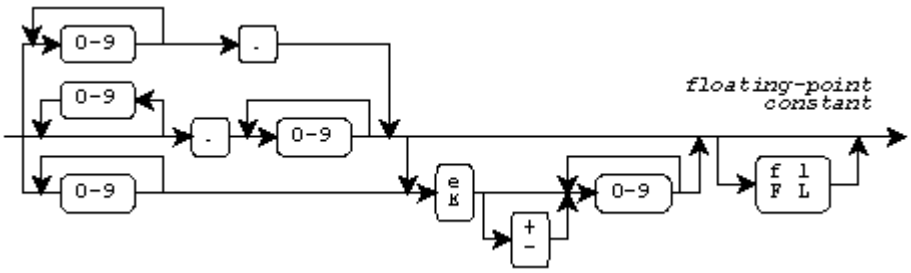
The value of an integer constant depends on its form:

- A leading `0x` or `0X` indicates a hexadecimal (base 16) integer.
- A leading `0` indicates an octal (base 8) integer.
- A leading nonzero digit indicates a decimal (base 10) integer.

You write any combination of:

- at most one `l` or `L` suffix to indicate a *long* type
- at most one `u` or `U` suffix to indicate an *unsigned* type

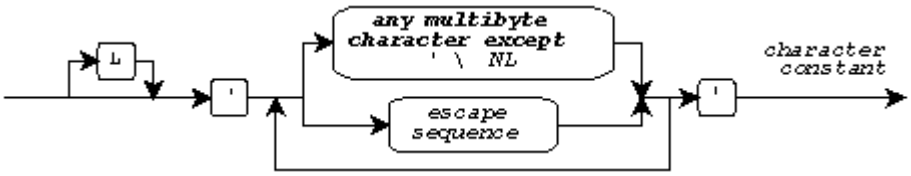
A floating-point constant is a preprocessing number that represents a value of a floating-point type. You write either a decimal point, an exponent, or both to distinguish a floating-point constant from an integer constant:



You write at most one `f` or `F` suffix to indicate type *float,* or at most one `l` or `L` suffix to indicate type *long double.*

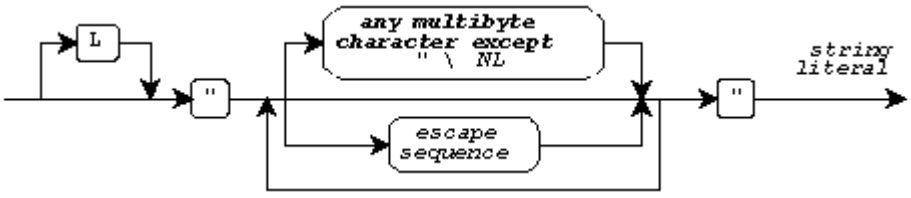
Character Constants and String Literals

A **character constant** C token has the same form as a preprocessing character constant:



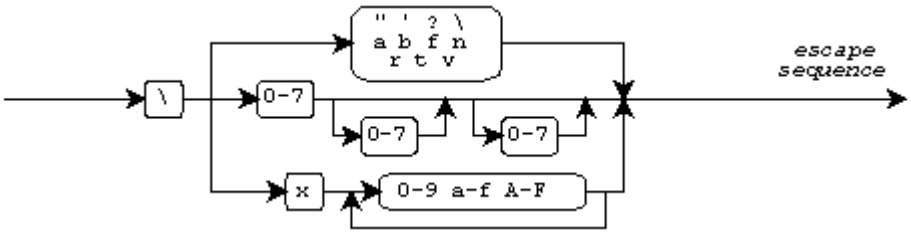
Its value depends on the character(s) you specify and any prefix you write.

A **string literal** C token has the same form as a preprocessing string literal:



Its value depends on the character(s) you specify and any prefix you write.

An **escape sequence** has the same form as within a preprocessing character constant or string literal:



An **operator** or **punctuator** token has the same form as a preprocessing operator or punctuator, except that the tokens `#` and `##` (and `%:` and `%:%:`, with Amendment 1) have meaning only during preprocessing. Moreover, the remaining Amendment 1 additions map to other C tokens:

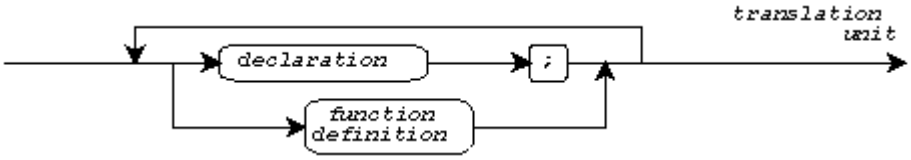
- `<:` becomes `[`
- `:>` becomes `]`
- `<%` becomes `{`
- `%>` becomes `}`

The following table shows all remaining operators and punctuators:

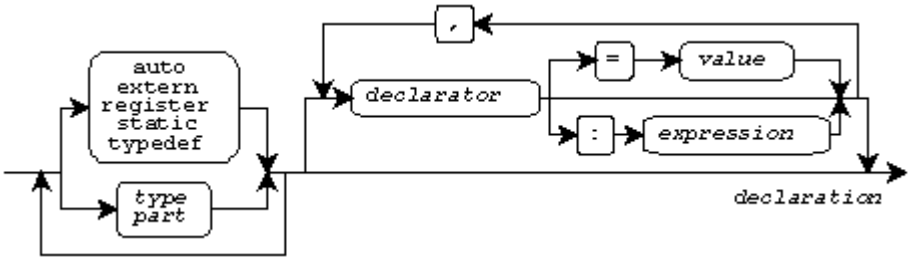
```
...     &&    -=    >=    ~     +     ;     ]
<<=     &=    ->    >>    %     ,     <     ^
>>=     *=    /=    ^=    &     -     =     {
!=      ++    <<    |=    (     .     >     |
%=      +=    <=    ||    )     /     ?     }
        --    ==    !     *     :     [
```

Declaration Syntax

The translator parses all C tokens that constitute a translation unit:



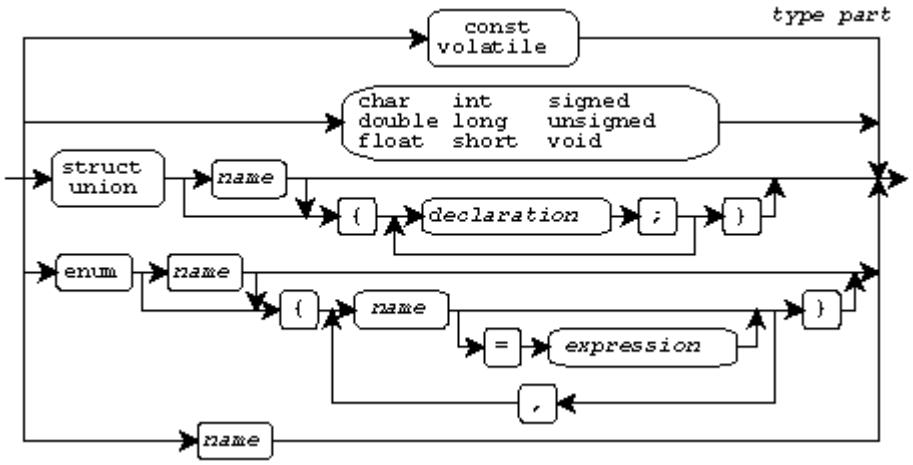
as one or more **declarations**, some of which are **function definitions**. A declaration (other than a function definition) takes a variety of forms:



Declarations can contain other declarations. You cannot write a function definition inside another declaration, however. There are many contexts for declarations. Some forms of declarations are permitted only in certain contexts.

Storage Class and Type Parts

You begin a declaration with an optional **storage class** keyword, intermixed with zero or more **type parts**:



The storage class keyword is from the set:

```
auto      extern    register  static    typedef
```

You write a type part as any one of the following.

- a type qualifier keyword, from the set:

```
const     volatile
```

- a type specifier keyword, from the set:

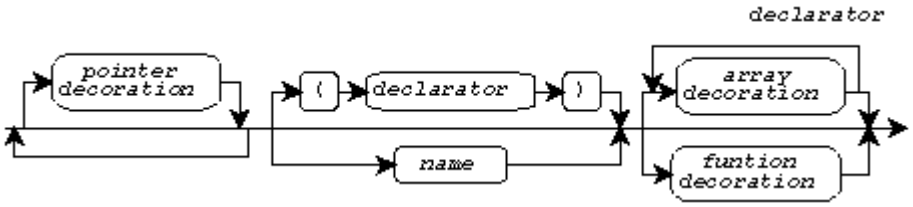
```
char      double    float     int       long
short     signed    unsigned  void
```

- a structure, union, or enumeration specification
- a type definition name

You can write only certain combinations of type parts.

Declarators

You can follow the storage class and type part of a declaration with a list of **declarators** separated by commas. Each declarator can specify a name for the entity that you are declaring as well as additional type information:



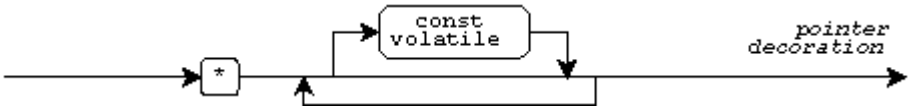
You write a declarator as, in order:

1. zero or more **pointer decorations**

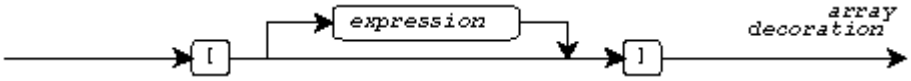
2. an optional name or a declarator in parentheses

3. zero or more **array decorations** or at most one **function decorations**

A pointer decoration consists of an asterisk (`*`) followed by an optional list of type qualifier keywords:



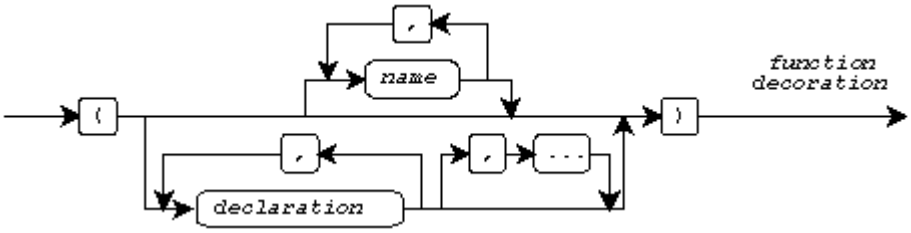
An array decoration consists of an optional expression enclosed in brackets (`[]`):



A function decoration is a sequence of one of the following:

- zero or more parameter names
- one or more parameter declarations

In either sequence, the parameters are separated by commas and enclosed in parentheses:



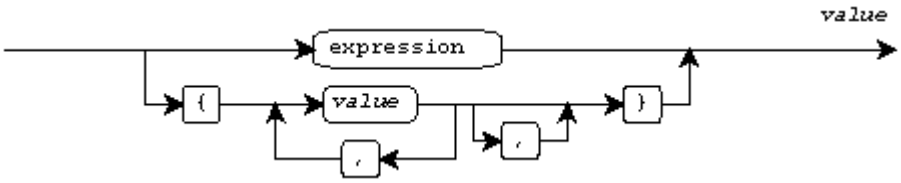
Some of these forms are permitted in certain contexts and not in others.

Object Initializers and Bitfield Specifications

You can follow each declarator with one of the following:

- an optional object initializer, consisting of an equal sign (=) followed by a value
- an optional bitfield size, consisting of a colon (`:`) followed by an expression

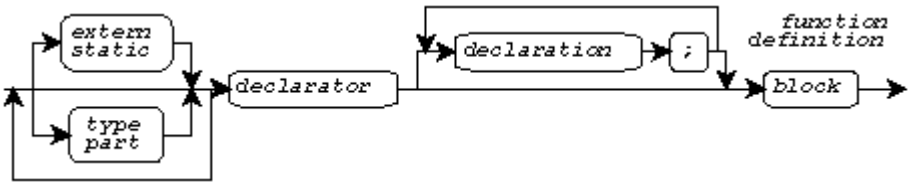
You write an object initializer **value** as either an expression or a list of such values separated by commas and enclosed in braces `{}`:



You can write a trailing comma after the last value in a comma separated list of object initializers.

Function Definition Syntax

A function definition declares a function and specifies the actions it performs when it executes:



You write a function definition as, in order:

1. an optional storage class and type parts

2. a declarator

3. zero or more parameter declarations each terminated by a semicolon

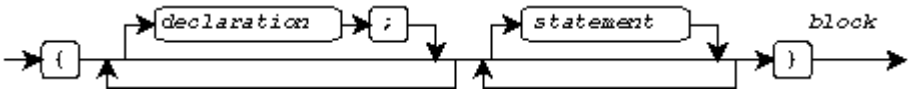
4. a **block**

The declarator contains a function decoration that describes the parameters to the function. You can write parameter declarations before the block only if the function decoration contains a list of parameter names.

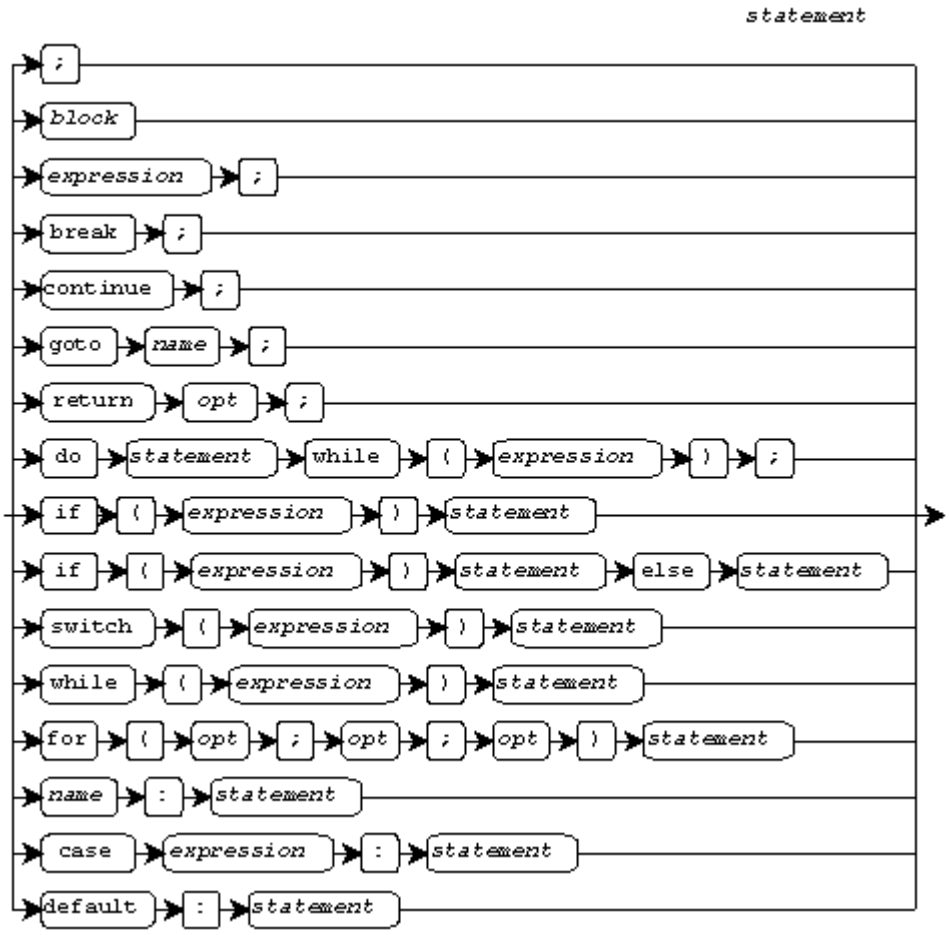
A block consists of braces surrounding, in order:

1. zero or more declarations each terminated by a semicolon

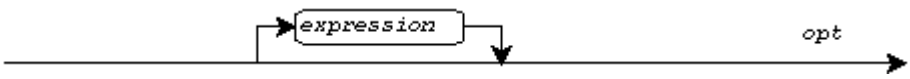
2. zero or more statements



A block contains a sequence of statements that specifies the actions performed by the block when it executes:



Here, `opt` represents an optional expression:

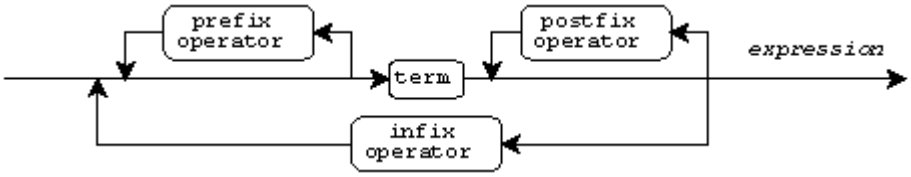


Statements specify the flow of control through a function when it executes. A statement that contains expressions also computes values and alters the values stored in objects when the statement executes.

Expression Syntax

You use expressions to specify values to the translator or to specify the computations that a program performs when it executes.

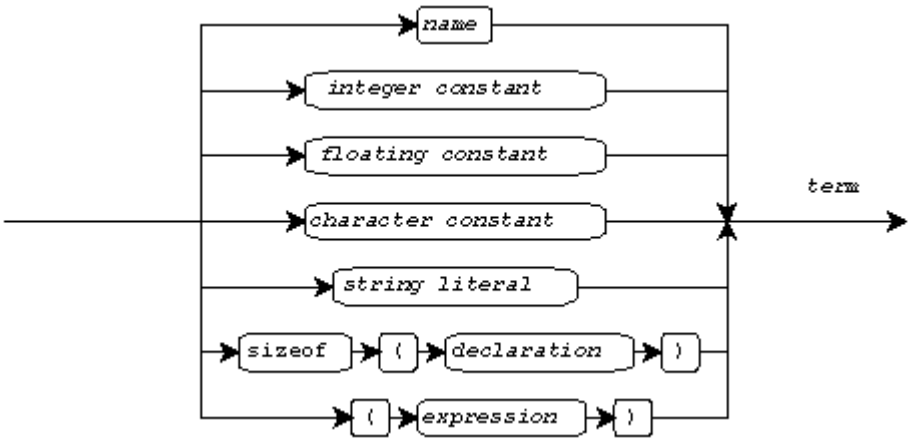
- You write an expression as one or more **terms** separated by **infix operators**:
- Each term is preceded by zero or more **prefix operators**. Each term is followed by zero or more **postfix operators**



Only certain combinations of operators and terms form valid expressions.

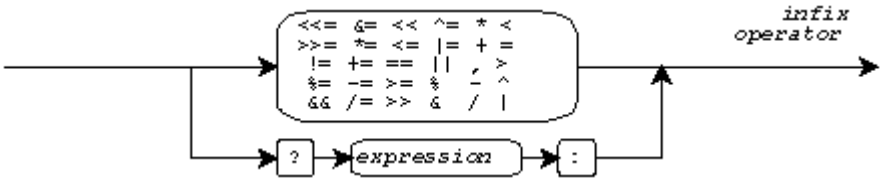
You write a term as one of the following:

- a name that is declared as a function, object, or enumeration constant
- an integer constant
- a floating-point constant
- a character constant
- a string literal
- the *sizeof* operator followed by a declaration enclosed in parentheses
- an expression enclosed in parentheses



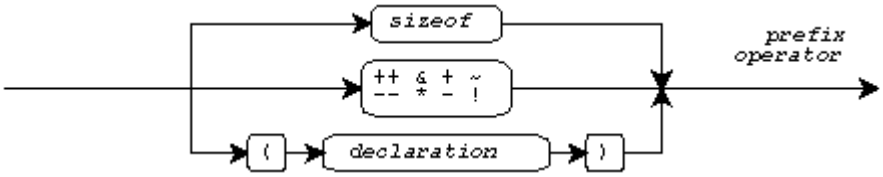
You write an infix operator as one of the following:

- one of the infix operator tokens
- the conditional operator pair `?` `:` enclosing another expression



You write a prefix operator as one of the following:

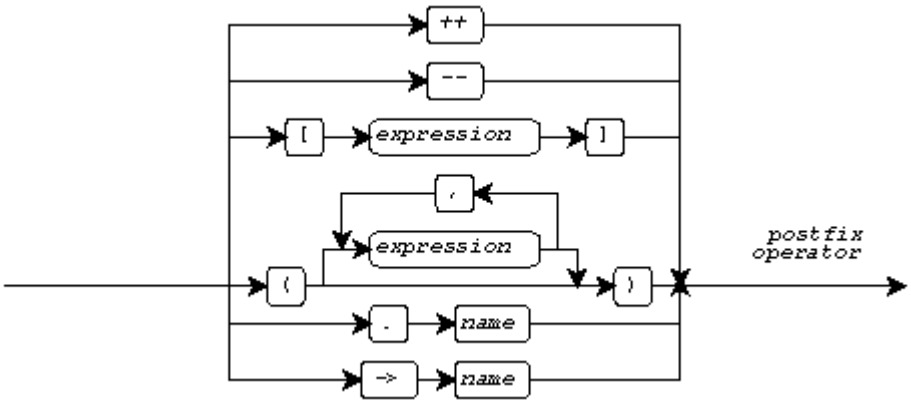
- the keyword `sizeof`
- one of the prefix operator tokens
- a *type cast* operator (consisting of a declaration enclosed in parentheses)



You can write only certain forms of declarations in a type cast.

You write a postfix operator as one of the following:

- the postfix operator token `++`
- the postfix operator token `--`
- an array subscript expression, enclosed in brackets `[]`
- a function call argument expression list, enclosed in parentheses `()`
- the member selection operator (a period), followed by the name of a structure or union member
- the member selection operator `->`, followed by the name of a structure or union member



You can write only certain forms of expressions in some contexts.

See also the **Table of Contents** and the **Index**.

Costrutti iterativi

while

while (condizione di ripetizione del ciclo**)** istruzione;	**while (**condizione di ripetizione del ciclo**){** istruzione$_1$; istruzione$_2$; ... istruzione$_n$; **}**

for

for (espressione di inizializzazione**;** condizione di ripetizione del ciclo**;** espressione di aggiornamento**)** istruzione;	**for (**espressione di inizializzazione**;** condizione di ripetizione del ciclo**;** espressione di aggiornamento**) {** istruzione$_1$; istruzione$_2$; ... istruzione$_n$; **}**

do-while

do istruzione; **while (**condizione di ripetizione del ciclo**);**	**do{** istruzione$_1$; istruzione$_2$; ... istruzione$_n$; **}while (**condizione di ripetizione del ciclo**);**

Direttive

include

#include nome_libreria

define

#define simbolo valore

Librerie ANSI

stdio.h

Prototipo	**Scopo**
FILE *fopen(char *filename, char *mode);	Apre un file **filename** in modalità **mode** e restituisce il puntatore al file. Restituisce NULL nel caso di errore.
void fclose(FILE *fp);	Chiude il file individuato da **fp**.

int getc(FILE *fp);	Legge un carattere dal file individuato da **fp** e passa al successivo.
int getchar(void);	Legge un carattere dallo standard input.
int putc(char ch, FILE *fp);	Scrive il carattere **ch** sul file individuato da **fp**.
int putchar(char ch);	Scrive il carattere **ch** sullo standard output.
char * gets(char buffer[]);	Legge una linea di testo da standard input e la memorizza in **buffer** .
int fgets(char buffer[], int max, FILE *fp);	Legge una linea di testo di al più **max** caratteri dal file individuato da **fp** e la memorizza in **buffer**. Il terminatore di riga viene memorizzato in buffer.
int fputs(char *buffer, FILE *fp);	Scrive la stringa **buffer** nel file individuato da **fp**.
void printf(char *buffer, ...);	Scrive la stringa **buffer** sullo standard ouput.
void fprintf(FILE *fp, char *buffer, ...);	Scrive la stringa **buffer** sul file individuato da **fp**.
int scanf(char *format, ...);	Acquisisce i dati dallo standard input secondo il formato **format** e li memorizza nelle variabili i cui indirizzi vengono indicati dopo la ,. Restituisce il numero di valori correttamente acquisiti.
int fscanf(FILE *fp, char *format, ...);	Acquisisce i dati dal file individuato da **fp** secondo il formato **format** e li memorizza nelle variabili i cui indirizzi vengono indicati dopo la ,. Restituisce il numero di valori correttamente acquisiti.

stdlib.h

Prototipo	**Scopo**
int abs(int n);	Restituisce il valore assoluto di **n**.
void *malloc(int nBytes);	Alloca un blocco di memoria di dimensioni **nByte** e restituisce il puntatore al primo indirizzo del blocco. Se non c'è sufficiente memoria restituisce NULL.
void free(void *p);	Libera la memoria associata al puntatore **p**, allocata in precedenza mediante una malloc.

math.h

Prototipo	**Scopo**
double ceil(double x)	Arrotonda **x** all'intero superiore rappresentato in virgola mobile.
double cos(double x)	Restituisce il coseno dell'angolo **x** (espresso in radianti).
double exp(double x)	Restituisce e^{x}.
double fabs(double x)	Restituisce il valore assoluto di **x** (tipo double).
double floor(double x)	Arrotonda **x** all'intero inferiore.
double fmod(double x, double y)	Restituisce il resto della divisione **x** / **y** rappresentato in virgola mobile.
double log(double x)	Restituisce il logaritmo naturale di **x**.
double log10(double x)	Restituisce il logaritmo base10 di **x**.
double pow(double x, double y)	Restituisce x^{y}.
double sin(double x)	Restituisce il seno dell'angolo **x** (espresso in radianti).
double sqrt(double x)	Resituisce la radice quadrata positiva di **x**.
double tan(double x)	Restituisce la tangente dell'angolo **x** (espresso in radianti).

ctype.h

Prototipo	**Scopo**

int isalpha(int ch);	Restituisce 0 se **ch** non è un carattere alfabetico, altrimenti un numero intero diverso da 0. ('A' - 'Z' oppure 'a' - 'z')
int isupper(int ch);	Restituisce 0 se **ch** non è un carattere maiuscolo, altrimenti un numero intero diverso da 0. ('A' - 'Z')
int islower(int ch);	Restituisce 0 se **ch** non è un carattere minuscolo, altrimenti un numero intero diverso da 0. ('a' - 'z')
int isdigit(int ch);	Restituisce 0 se **ch** non è una cifra, altrimenti un numero intero diverso da 0. ('0' - '9')
int isalnum(int ch);	Restituisce 0 se **ch** non è un carattere alfanumerico, altrimenti un numero intero diverso da 0. ('A' - 'Z' oppure 'a' - 'z' oppure '0' - '9')
int isspace(int ch);	Restituisce 0 se **ch** non è un carattere di spaziatura, altrimenti un numero intero diverso da 0. (spazio, tabulatore, a capo, nuova linea, tabulatore verticale, nuova pagina)
int toupper(int ch);	Se **ch** è un carattere alfabetico restituisce il corrispondente carattere maiuscolo, in caso negativo restituisce il carattere non modificato.
int tolower(int ch);	Se **ch** è un carattere alfabetico restituisce il corrispondente carattere minuscolo, in caso negativo restituisce il carattere non modificato.

string.h

Prototipo	Scopo
int strlen(char *s);	Restituisce la lunghezza della stringa **s**, escluso il terminatore '\0'
int strcmp(char *s1, char *s2);	Restituisce 0 se la stringa **s1** è uguale a **s2**, un numero negativo se **s1** viene prima di **s2** nell'ordine lessicografico, un numero positivo in caso contrario.
char *strcpy(char dest[], char *sorg);	Copia la stringa **sorg** nell'array **dest**, e restituisce ls stringa copiata.

Versione pdf stampabile