Strutture di controllo e tipi di dato

Alessandro Pellegrini
a.pellegrini@ing.uniroma2.it

Strutture di controllo

- Il C è estremamente semplice e fornisce un numero ridotto di strutture di controllo
  - Questo perché in tutte le ISA il cambio di flusso di controllo è realizzato con un numero molto esiguo di istruzioni macchina

- I costrutti fondamentali permettono di variare il flusso di controllo:
  - Effettuando un controllo di condizione
  - Effettuando un'iterazione
  - Effettuando un salto a funzione (a sottoprogramma)

Controllo di condizione

```
if(TEST) {
    CODE;
} else if(TEST) {
    CODE;
} else {
    CODE;
}
```

```
if(TEST)
  SINGOLO STATEMENT;
else
  SINGOLO STATEMENT;
```

```
switch (OPERAND) {
   case CONSTANT:
      CODE;
      break;
   default:
      CODE;
   }
}
```

- Sono basati su *tabelle di salto*

Non dimenticate il break!

```
switch (age) {
  case 1:  printf("You're one.");           break;
  case 2:  printf("You're two.");           break;
  case 3:  printf("You're three."); __attribute__((fallthrough));
  case 4:  printf("You're three or four."); break;
  default: printf("You're not 1,2,3 or 4!");
}
```

Cicli: while

```
while(TEST) {
    CODE;
}
```

```
while(TEST) {
    if(OTHER_TEST) {
        break;
    }
    CODE;
}
```

```
while(TEST) {
    if(OTHER_TEST) {
        continue;
    }
    CODE;
}
```

Cicli: do while e for

```
do {
    CODE;
} while(TEST);
```

```
for(INIT; TEST; POST) {
    CODE;
}
```

Anche in questi cicli è possibile utilizzare `break` e `continue`.

Differenza tra while e do/while

Salti e chiamate a funzioni

```
if(ERROR_TEST) {
    goto fail;
}

fail:
CODE;
```

- Le funzioni hanno un pattern standard:

```
TYPE NAME(ARG1, ARG2, ...)
{
   CODE;
   return VALUE;
}
```

- Per esempio:

```
int name(int arg1)
{
  CODE;
  return 0;
}
```

Attenzione al goto ed alla sua destinazione

```
int idempotente(int x){
   int ret;
   goto beyond;


   ret = x;
 beyond:
   return ret;
}
```

```
if(x == 0) {
     int y;
     y = 1;
  above:
     return y;
} else {
     goto above;
}
```

Variabili in C

- Ogni variabile, in C, **ha un tipo**
  - Differente dal concetto di “duck typing”, ad esempio in Python: *Se parla e si comporta come una papera, allora è una papera*
  - Perché?

- Le variabili in C possono essere raggruppate in tre tipologie:
  - Tipi primitivi (interi, virgola mobile, …)
  - Tipi aggregati (strutture, unioni)
  - Puntatori
- I tipi primitivi ed i puntatori sono gli unici tipi di variabili che hanno un corrispettivo in istruzioni assembly
- I tipi aggregati vengono automaticamente convertiti dal compilatore in accesso a tipi primitivi

Ambito delle variabili

- Ogni variabile dichiarata nel programma ha un certo *ambito* (scope), che determina la *visibilità* della variabile a determinate porzioni del programma:
  - variabili **globali**: occupano memoria all'interno delle sezioni `.data` e `.bss`
  - variabili **locali** (o *automatiche*): occupano memoria all'interno dello stack
  - varibili **statiche**: come le variabili globali, ma possono essere accedute solo all'interno della funzione/modulo C

- In C89, le variabili **dovevano** essere dichiarate all'inizio della funzione
  - È comunque una buona prassi da conservare anche nelle versioni successive
  - Dà un'indicazione della *complessità* della funzione che state scrivendo

La finestra di stack (o record di attivazione)

- Le variabili automatiche di una funzione occupano memoria all'interno dello stack
- È possibile distinguere il "contesto" di esecuzione di una funzione poiché all'ingresso viene automaticamente creato una "finestra" di stack (o record di attivazione)
- Durante la creazione, viene riservato spazio per le variabili automatiche
- Il contesto contiene anche l'indirizzo di ritorno all'istruzione successiva al salto a sottoprogramma
- Al termine della funzione, il record di attivazione viene *invalidato logicamente*
  - Il suo contenuto permane in memoria!

La finestra di stack (o record di attivazione)

```
void function() {
    int x = 128;
    return;
}
```

```
function:
    addiu   $sp,$sp,-24
    sw      $fp,20($sp)
    move    $fp,$sp
    li      $2,128
    sw      $2,8($fp)
    move    $sp,$fp
    lw      $fp,20($sp)
    addiu   $sp,$sp,24
    j       $ra
```

Ambito di blocco: un esempio

file: scope.c

```c
#include <stdio.h>

int main(void)
{
        int number = 5;
        {
                int number = 20;
                printf("inner number: %d\n", number);
        }
        printf("outer number: %d\n", number);

        return 0;
}
```

Posso dichiarare variabili nei case?

- Per prevenire il trasferimento di controllo all'interno dell'ambito di una variabile, non è possibile dichiarare variabili nei case, a meno di non dichiarare esplicitamente l'ambito con un blocco

```
switch(condition) {
    case 1:
            {
             int x = 0;
            } // scope of 'x' ends here
    default:
             fprintf("%d", x);
}
```

errore

Passaggio di parametri: convenzioni di chiamata

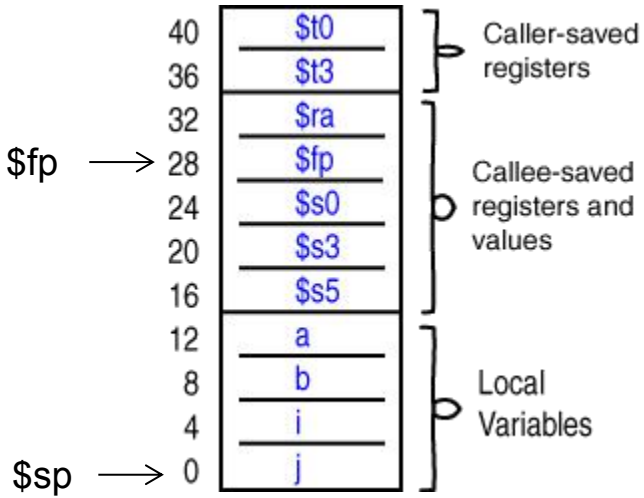
- Affinché una subroutine chiamante possa correttamente dialogare con la subroutine chiamata, occorre mettersi d'accordo su come passare i parametri ed il valore di ritorno
- Le *calling conventions* definiscono, *per ogni architettura e sistema*, come è opportuno passare i parametri
- Le convenzioni principali permettono di passare i parametri tramite:
  - lo stack
  - i registri
  - un misto delle due tecniche
- Generalmente il valore di ritorno viene passato in un registro perché la finestra di stack viene distrutta al termine della subroutine
  - Se la subroutine chiamante vuole conservare il valore nel registro, deve memorizzarlo nello stack prima di eseguire la call

Calling convention MIPS

- I primi quattro parametri sono passati nei registri `$a0–$a3`
- I parametri aggiuntivi vengono inseriti sullo stack in ordine inverso
- Sono previsti due registri da usare per restituire valori di ritorno: $v0 e $v1
- I registri sono divisi in *callee save* e *caller save*
  - callee save: `$s0–$s7`, `$sp`, `$fp`
  - caller save: `$v0–$v1`, `$a0–$a3`, `$t0–$t9`

Anatomia dello stack

```
void f(int first, int second, int third, int fourth, int fifth) {
        int a, b, i, j;
}
```



Tipi primitivi

- In C, ci sono 4 tipi *primitivi*:
  - **`char`**, **`int`**, **`float`**, **`double`**
- Questi tipi possono essere *alterati* (in termini di rappresentazione e dimensione) utilizzando dei *modificatori*:
  - **`signed`**, **`unsigned`**, **`short`**, **`long`**

- *Criticità* di questi tipi: **non è garantita una dimensione esatta in byte** se si cambia architettura o se si cambia compilatore
  - lo standard definisce solo delle relazioni tra i differenti tipi

- Esiste poi il "tipo" speciale **`void`** (tipo di dato "vuoto"):
  - Può essere utilizzato come valore di ritorno o come argomento
  - Non si può dichiarare una variabile di tipo **`void`**.

Un esempio con variabili di tipo differente

file: types.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
        int distance = 100;
        float power = 2.345f;
        double super_power = 56789.4532;
        char initial = 'A';
        char first_name[] = "Alessandro";
        char last_name[] = "Pellegrini";

        printf("You are %d miles away.\n", distance);
        printf("You have %f levels of power.\n", power);
        printf("You have %f awesome super powers.\n", super_power);
```

Un esempio con variabili di tipo differente

```
printf("I have an initial %c.\n", initial);
printf("I have a first name %s.\n", first_name);
printf("I have a last name %s.\n", last_name);
printf("My whole name is %s %s.\n", first_name, last_name);

int bugs = 100;
double bug_rate = 1.2;
printf("You have %d bugs at the imaginary rate of %f.\n", bugs, bug_rate);
long int universe_of_defects = 1L * 1024L * 1024L * 1024L;
printf("The entire universe has %ld bugs.\n", universe_of_defects);
double expected_bugs = bugs * bug_rate;
printf("You are expected to have %.02f bugs.\n", expected_bugs);
```

Un esempio con variabili di tipo differente

```
    // this makes no sense, just a demo of something weird
    char nul_byte = '\0';
    int care_percentage = bugs * nul_byte;
    printf("Which means you should care %d%%.\n", care_percentage);

    return 0;
}
```

Tipi primitivi “modificati”

Tipo	Spiegazione	Dimensione minima (bit)	Specificatore di formato
`char`	Minima unità indirizzabile della macchina che contiene l’insieme basilare di caratteri. È di tipo intero. Può essere segnato o non segnato. Contiene CHAR_BIT bit.	8	%c
`signed char`	Come un char, ma segnato. Contiene almeno l’intervallo [-127, +127].	8	%c (%hhi per il numero)
`unsigned char`	Come un char, ma non segnato. Contiene almeno l’intervallo [0, 255].	8	%c (%hhu per il numero)

`short`
`short int`
`signed short`
`signed short int`
	Tipo intero segnato breve. Contiene almeno l’intervallo [-32,767, +32,767].	16	%hi o %hd

`unsigned short` `unsigned short int`	Tipo intero non segnato breve. Contiene almeno l’intervallo [0, 65,535].	16	%hu
`int` signed signed int	Tipo intero segnato di base. Contiene almeno l’intervallo [-32,767, +32,767]	16	%i o %d

perché convenzionalmente lo usiamo come se fosse a 32 bit?

Tipo	Spiegazione	Dimensione minima (bit)	Specificatore di formato
`unsigned` `unsigned int`	Tipo intero non segnato di base. Contiene almeno l'intervallo [0, 65,535].	16	%u

`long`
`long int`
`signed long`
`signed long int`
	Tipo intero segnato lungo. Contiene almeno l'intervallo [-2,147,483,647, +2,147,483,647].	32	%li o %ld

`unsigned long` `unsigned long int`	Tipo intero non segnato lungo. Contiene almeno l'intervallo [0, 4,294,967,295]	32	%lu

`long long`
`long long int`
`signed long long`
`signed long long int`
	Tipo intero segnato lungo lungo. Contiene almeno l'intervallo [-9,223,372,036,854,775,807, +9,223,372,036,854,775,807] (dal C99)	64	%lli o %lld

`unsigned long long` `unsigned long long int`	Tipo intero non segnato lungo lungo. Contiene almeno l'intervallo [0, +18,446,744,073,709,551,615] (dal C99)	64	%llu
`float`	Tipo reale a singola precisione. Le sue proprietà non sono specificate dallo standard.		%f
`double`	Tipo reale a doppia precisione. Le sue proprietà non sono specificate dallo standard.		%lf o%f
`long double`	Tipo reale a precisione estesa. Le sue proprietà non sono specificate dallo standard.		%Lf %LF

Cast: conversione tra tipi

- È possibile convertire un tipo in un altro effettuando un'operazione di *cast*:

  (tipo di destinazione)variabile;

- Ad esempio:

```
int  integer = 10;
double  real = (double)integer;
```

- Se si utilizzano tipi differenti in un'assegnazione, il compilatore effettua un *autocast*
- Attenzione a non abusare degli autocast, siate espliciti il più possibile: i bug sono dietro l'angolo

Pericolo dell'autocast

file: autocast.c

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
        double result = 1 / 2 * 2;
        printf("Result is %lf\n", result);
        return 0;
}
```

Il risultato è 0!

Pericolo dell'autocast

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
        double result = 1. / 2 * 2;
        printf("Result is %lf\n", result);
        return 0;
}
```

Il risultato è 1!

Pericolo dell'autocast

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
        double result = (double)1 / 2 * 2;
        printf("Result is %lf\n", result);
        return 0;
}
```

Il risultato è 1!

Pericolo dell'autocast

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
        double result = 1 / 2 * (double)2;
        printf("Result is %lf\n", result);
        return 0;
}
```

Il risultato è 0!

Classi di archiviazione (*storage class specifiers*)

- **`auto`**: allocazione automatica (fondamentalmente inutile)
- **`register`**: allocazione automatica. Suggerisce anche al compilatore di inserire l'oggetto in un registro della CPU (deprecato)
- **`static`**: linking interno: non è consentito accedere all'oggetto dall'esterno del modulo C
- **`extern`**: linking esterno: si indica al compilatore che l'oggetto è stato/sarà definito in un altro modulo C, affidando al linker il compito di rilocarlo (ridondante)
- **`_Thread_local`** (dal C11): nella programmazione concorrente, si dice al compilatore di utilizzare un'istanza di variabile diversa per ciascun thread (in `threads.h` viene definito un alias **`thread_local`**)

Tipi primitivi a dimensione fissa

- Introdotti con il C99, basati su definizioni presenti in `stdint.h`

Tipo	Definizione
`int8_t`	Intero segnato a 8 bit
`uint8_t`	Intero non segnato a 8 bit
`int16_t`	Intero segnato a 16 bit
`uint16_t`	Intero non segnato a 16 bit
`int32_t`	Intero segnato a 32 bit
`uint32_t`	Intero non segnato a 32 bit
`int64_t`	Intero segnato a 64 bit
`uint64_t`	Intero non segnato a 64 bit

- Il pattern è (u)int<n>_t, che viene rispecchiato anche in alcune costanti:
  - INT<n>_MAX: Numero positivo massimo dell'intero segnato di <n> bit, ad esempio INT16_MAX
  - INT<n>_MIN: Minimo numero negativo dell'interno segnato di <n> bit
  - UINT<n>_MAX: Massimo numero positivo dell'intero non segnato di <n> bit

Altre definizioni presenti in `stdint.h`

Tipo	Definizione
`(u)int_least<n>_t`	Utilizza *almeno* (N) bit
`(U)INT_LEAST<n>_MAX`	Valore massimo del tipo corrispondente a <n> bit
`INT_LEAST<n>_MIN`	Valore minimo del tipo corrispondente a <n> bit
`(u)int_fast<n>_t`	Simile a `_least`, ma richiede un'implementazione hardware veloce
`(U)INT_FAST<n>_MAX`	Valore massimo del tipo corrispondente a <n> bit
`INT_FAST<n>_MIN`	Valore minimo del tipo corrispondente a <n> bit
`(u)intptr_t`	Un intero grande abbastanza da contenere un puntatore
`(U)INTPTR_MAX`	Valore massimo di un `(u)intptr_t`
`INTPTR_MIN`	Valore minimo di un `intptr_t`
`(u)intmax_t`	La rappresentazione intera più grande disponibile sull'hardware corrente
`(U)INTMAX_MAX`	Valore massimo di un `(u)intmax_t`
`INTMAX_MIN`	Valore minimo di un `intmax_t`
`ptrdiff_t`	Un intero segnato risultante dalla sottrazione di due puntatori
`PTRDIFF_MAX`	Valore massimo di `ptrdiff_t`
`PTRDIFF_MIN`	Valore minimo di `ptrdiff_t`

Un esempio con strutture di controllo

file: if.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
        int i = 0;

        if (argc == 1) {
                printf("You only have one argument.\n");
        } else if (argc > 1 && argc < 4) {
                printf("Here's your arguments:\n");
                for (i = 0; i < argc; i++) {
                        printf("%s ", argv[i]);
                }
                printf("\n");
        } else {
                printf("You have too many arguments.\n");
        }

        return 0;
}
```

I booleani

- In C non esiste il tipo “booleano”
  - Non esiste alcuna istruzione in nessuna ISA che gestisca i booleani
  - Per convenzione: 0 è false, un valore diverso da 0 è true

```
char booleano = 1;
if(booleano)
    printf(“true\n”);
```

- Nel C11 è stato introdotto l’header `stdbool.h`
  - Utilizza un nuovo tipo primitivo: **`_Bool`**
  - L’header definisce le keyword **`bool`**, **`true`**, **`false`**
  - Quando uno scalare viene convertito in **`_Bool`**, il risultato è zero se lo scalare è zero, 1 altrimenti.

Tutti gli operatori

Operatori Aritmetici	
+	somma
-	sottrazione
*	moltiplicazione
/	divisione
%	modulo
++	incremento
--	decremento

Operatori Logici	
&&	and logico
||	or logico
!	negazione logica
? :	confronto ternario

Operatori di Confronto	
==	uguaglianza
!=	non uguaglianza
>	maggiore di
<	minore di
>=	maggiore o uguale
<=	minore o uguale

Operatori Bit a Bit	
&	and binario
|	or binario
^	xor binario
~	complemento a 1
<<	shift a sinistra
>>	shift a destra

Assegnazione	
=	assegnazione *uguale*
+=	assegnazione *più-uguale*
-=	assegnazione *meno-uguale*
*=	assegnazione *per-uguale*
/=	assegnazione *diviso-uguale*
%=	assegnazione *modulo-uguale*
<<=	assegnazione *shift-sinistro-uguale*
>>=	assegnazione *shift-destro-uguale*
&=	assegnazione *and-uguale*
^=	assegnazione *xor-uguale*
|=	assegnazione *or-uguale*

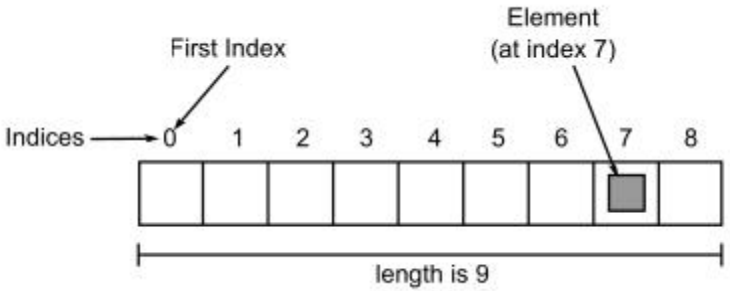
Un esempio

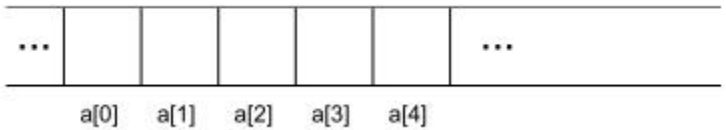
file: bool-while.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
        int i = 26;
        do {
                printf("%d, ", --i);
        } while(i);
        printf("\b\b \n");

        i = 0;
        while (i < 26) {
                printf("%d, ", i);
                i++;
        }
        printf("\b\b \n");

        return 0;
}
```

Array

- Un array (impropriamente tradotto con *vettore*) è una *collezione di elementi*, ciascuno identificato da un *indice*
- In memoria, gli elementi del vettore sono conservati in maniera contigua
- L'indirizzo dell'elemento cui si cerca di accedere viene calcolato con una *trasformazione lineare* dell'indice
  - ciascun elemento ha una dimensione differente in funione del suo *tipo*





Array

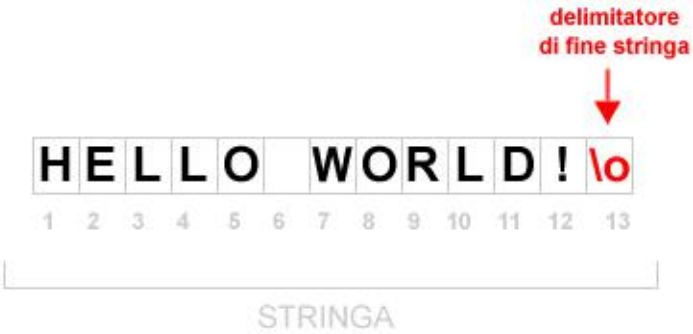
- In C, se chiamiamo una variabile di tipo vettore a, a questo nome simbolico sarà associato l'*indirizzo in memoria* in cui incomincia il vettore
- L'*operatore di spiazzamento* [ ] consente di calcolare la trasformazione lineare da un indice all'indirizzo:

```
a[i] = a + size * i
```

Stringhe

- Le stringhe sono conservate in memoria come un vettore di caratteri
- È necessario utilizzare un carattere speciale per identificare il punto in cui la stringa termina in memoria
- Per questo scopo si utilizza un valore speciale nella codifica ASCII: il *terminatore di stringa* (pari al byte zero, indicato col il carattere `'\0'`)

- Data questa rappresentazione in memoria, in C le stringhe sono dei vettori di char, terminati dal terminatore di stringa.



Array e stringhe

file: strings.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
        int nums[4] = { 0 };
        char name[4] = { 'a', 'p' };

        // first, print them out raw
        printf("nums: %d %d %d %d\n", nums[0], nums[1], nums[2], nums[3]);
        printf("name each: %c %c %c %c\n", name[0], name[1], name[2], name[3]);
        printf("name: %s\n", name);

        // set up the numbers
        nums[0] = 1;
        nums[1] = 2;
        nums[2] = 3;
        nums[3] = 4;
```

Array e stringhe

```
    // set up the name
    name[0] = 'A';
    name[1] = 'l';
    name[2] = 'e';
    name[3] = '\0';

    // then print them out initialized
    printf("nums: %d %d %d %d\n", nums[0], nums[1], nums[2], nums[3]);
    printf("name each: %c %c %c %c\n", name[0], name[1], name[2], name[3]);
    printf("name: %s\n", name);
}
```

Array e tabelle di salto

file: jump-table.c

```c
#include <stdio.h>


int main(int argc, char *argv[])
{
        int i = 0;

        char *states[] = { // A custom array of strings
                "California", "Oregon",
                "Washington", "Texas"
        };

        int num_states = 4;

        for (i = 1; i < argc; i++) {
                printf("arg %d: %s\n", i, argv[i]);
        }

        for (i = 0; i < num_states; i++) {
                printf("state %d: %s\n", i, states[i]);
        }

        return 0;
}
```

Usiamo le funzioni

file: functions.c

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

 // forward declarations
 int can_print_it(char ch);
 void print_letters(char arg[]);

 void print_arguments(int argc, char *argv[])
 {
         int i = 0;
         for (i = 0; i < argc; i++) {
                 print_letters(argv[i]);
         }
 }
```

Usiamo le funzioni

```c
void print_letters(char arg[])
{
        int i = 0;
        for (i = 0; arg[i] != '\0'; i++) {
                char ch = arg[i];
                if (can_print_it(ch)) {
                        printf("'%c' == %d ", ch, ch);
                }
        }
        printf("\n");
}

int can_print_it(char ch)
{
        return isalpha(ch) || isblank(ch);
}

int main(int argc, char *argv[])
{
        print_arguments(argc, argv);
        return 0;
}
```

sizeof

- **`sizeof`**`()` è un operatore unario che calcola *a tempo di compilazione* la dimensione di un tipo o di una variabile
  - Il valore restituito è in termini di char
  - **`sizeof`**`(`**`char`**`)` vale sempre 1

- Il parametro di **`sizeof`**`()` può essere:
  - un tipo
  - un tipo composto (struct e union, ne parleremo più avanti)
  - il nome di una variabile

enum

- Tipo “enumerazione”
- Permette di utilizzare dei nomi simbolici per degli insiemi
- Funziona come degli interi (effettivamente i nomi simbolici sono convertiti a interi)
- Se utilizzati in uno switch-case, qualche volta il compilatore genera dei warning se non si controllano tutti gli elementi della enum

enum: un esempio

```c
#include <stdio.h>

enum suit {
    CLUB,
    DIAMONDS,
    HEARTS = 20,
    SPADES = 3
} card;

int main(void)
{
    card = CLUB;
    printf("Size of enum variable = %d bytes", sizeof(card));
    return 0;
}
```

Array a lunghezza variabile

- A volte, non è possibile sapere la dimensione di un vettore a tempo di compilazione
- Dal C99 sono disponibili gli array a lunghezza variabile
  - Si tratta di vettori la cui dimensione è nota solo a tempo d'esecuzione
- Tipicamente, la dimensione è espressa in termini di una variabile (anche un parametro di una funzione)
- Questi vettori vengono istanziati dai compilatori tipicamente sullo stack

Array a lunghezza variabile

file: VLA.c

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

void generate_and_print(int n)
{
        int i;
        double vals[n];

        for (i = 0; i < n; i++) {
                vals[i] = (double)rand() / RAND_MAX * 50;
        }

        for (i = 0; i < n; i++)
                printf("%.03f, ", vals[i]);
        printf("\b\b \n");
}
```

```c
int main(void)
{
        srand(time(NULL));
        generate_and_print(3);
        generate_and_print(1);
        generate_and_print(12);
}
```

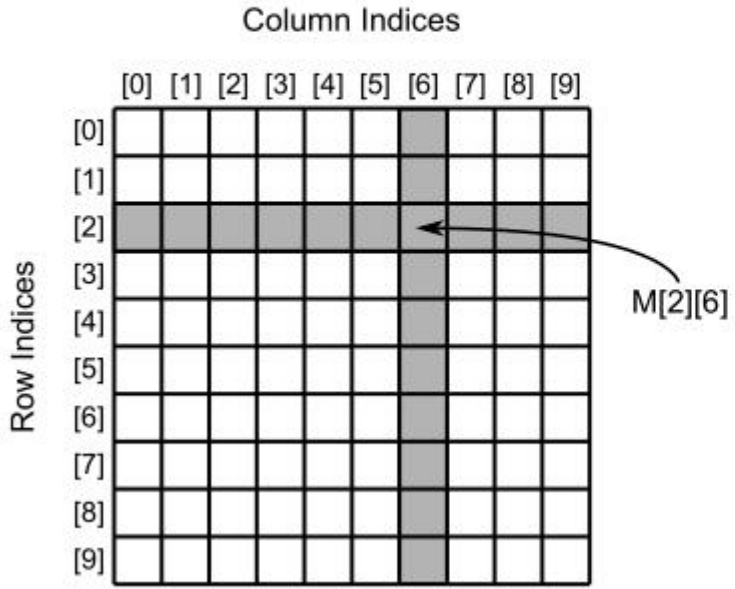
Matrici

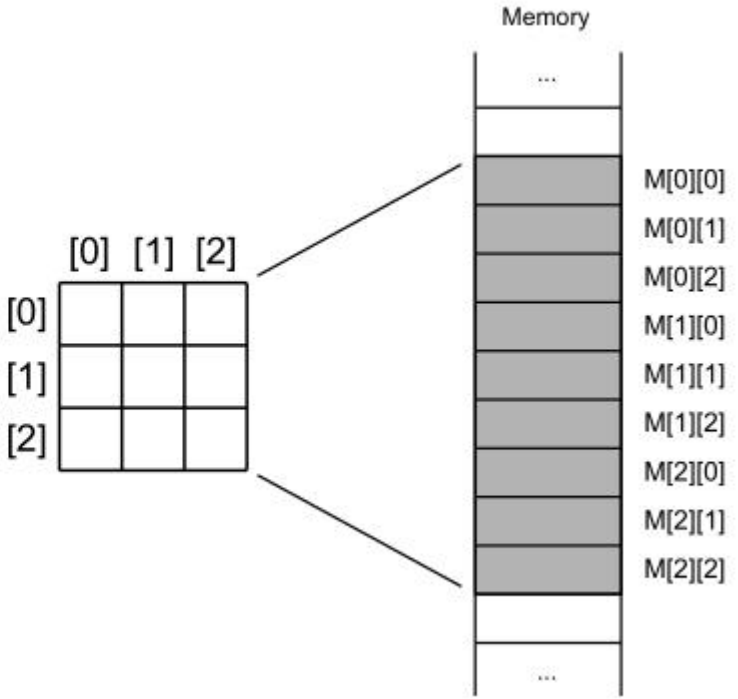
- Le matrici sono strutture dati a più dimensioni
- Tuttavia, il modello di memoria è piatto!
- Pertanto, le matrici vengono “linearizzate” in memoria
- Il comportamento dell’operatore `[ ]` è differente nel caso delle matrici:
  - Quando si accede utilizzando un doppio spiazzamento `[i][j]`, questo spiazzamento viene linearizzato nella forma:

```
address = i * N * size + j * size
```

- Ciò è possibile solo se le dimensioni delle matrici sono note a tempo di compilazione!
- Per estensione, lo stesso funzionamento è valido per le matrici *n*-dimensionali

Matrici


Column Indices
[0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
Row Indices
[0]
[1]
[2]
[3]
[4]
[5]
[6]
[7]
[8]
[9]
M[2][6]


Memory
...
[0] [1] [2]
[0]
[1]
[2]
M[0][0]
M[0][1]
M[0][2]
M[1][0]
M[1][1]
M[1][2]
M[2][0]
M[2][1]
M[2][2]
...

Come passare le matrici come parametri

- Poiché le matrici sono rappresentate in forma lineare in memoria, quando vengono passate come argomenti di una funzione è necessario specificarne in qualche modo la dimensione

- Nei compilatori più “arcaici” può essere necessario conoscere la dimensione a tempo di compilazione
- Questo problema viene risolto mediante l’utilizzo dei vettori a lunghezza variabile

- La funzione, in realtà, riceve l’*indirizzo* della matrice in memoria
  - sono possibili *side effect* nella funzione chiamata

Matrici: primo esempio

file: matrix.c

```c
#include <stdio.h>

int A[2][3] = {{1, 3, 0}, {-1, 5, 9}};
int B[][3] = {{2, 7, -4}, {3, -2, 7}};
int C[2][3] = {0, 0, 0, 0, 0, 0};

void print_matrix(int M[2][3])
{
        int i, j;

        for(i = 0; i < 2; i++) {
                for(j = 0; j < 3; j++) {
                        printf("%d\t", M[i][j]);
                }
                puts("");
        }
}
```

Matrici: primo esempio

```
int main(int argc, char *argv[])
{
        int i, j;

        puts("Matrix A:");
        print_matrix(A);

        puts("\nMatrix B:");
        print_matrix(B);

        for(i = 0; i < 2; i++) {
                for(j = 0; j < 3; j++) {
                        C[i][j] = A[i][j] + B[i][j];
                }
        }

        puts("\nMatrix C:");
        print_matrix(C);
}
```

Matrici: secondo esempio

file: matrix2.c

```c
#include <stdio.h>

int A[2][3] = {{1, 3, 0}, {-1, 5, 9}};
int B[][3] = {{2, 7, -4}, {3, -2, 7}};


void print_matrix1(int row, int col, int M[row][col])
{
        int i, j;

        for(i = 0; i < row; i++) {
                for(j = 0; j < col; j++) {
                        printf("%d\t", M[i][j]);
                }
                puts("");
        }
}
```

Matrici: secondo esempio

```
void print_matrix2(int col, int M[][col])
{
        int i, j;

        for(i = 0; i < 2; i++) {
                for(j = 0; j < col; j++) {
                        printf("%d\t", M[i][j]);
                }
                puts("");
        }
}
```

Matrici: secondo esempio

```
int main(int argc, char *argv[])
{
        puts("Matrix A:");
        print_matrix1(2, 3, A);

        puts("\nMatrix B:");
        print_matrix2(3, B);

        return 0;
}
```

Puntatori

- Ciascun tipo T ha il corrispondente tipo *puntatore a T*.
- Un puntatore è un tipo di dato che contiene l'*indirizzo* dell'area di memoria che contiene una variabile di quel tipo.
- Un puntatore differisce dalla variabile di quel tipo poiché si utilizza il *dichiaratore di tipo asterisco* (*) tra il tipo e il nome della variabile:

```
int *intptr;
```

- Attenzione! L'asterisco modifica la variabile, non il tipo!

```
int *var1, var2;
```

- Esiste un generico puntatore a memoria: `void *`

Puntatori: come usarli

- `type *ptr`: Un puntatore di tipo `type` chiamato `ptr`
- `*ptr`: Il valore della variabile puntata da `ptr`, qualunque sia il tipo
- `*(ptr + i)`: Il valore di (qualsiasi cosa si trovi all'indirizzo `ptr + i`)
- `&var`: L'indirizzo della variabile `var`
- `type *ptr = &var`: Un puntatore di tipo `type` chiamato `ptr` inizializzato all'indirizzo di `var`
- `ptr++`: Incrementa l'indirizzo cui punta `ptr`

Aritmetica dei puntatori

- Come detto in precedenza:
  - `ptr++`: Incrementa l'indirizzo cui punta `ptr`
- I puntatori tuttavia hanno un tipo, pertanto cosa vuol dire "incrementare l'indirizzo?"
- L'aritmetica dei puntatori è tale per cui l'*indirizzo* contenuto nella variabile puntatore viene incrementato di un numero di byte pari alla *taglia* del tipo primitivo puntato
- dato un `int *p = (void *)1000`, assumendo `sizeof(int) == 4`:
  - `p+1 == 1004`
  - `p+2 == 1008`
  - `p+n == 1000+n*4`
- Operazioni valide sono somma, sottrazione e sottrazione tra puntatori

Aritmetica dei puntatori

- Quanto vale p dopo questo incremento?

```
void *p = (void *)1000;
p++;
```

- È un'operazione **illegale**!
- Standard C11 § 6.5.6, paragrafo 2:
  - *Per l'addizione, entrambi gli operandi devono avere un tipo aritmetico, oppure un operando deve essere un puntatore a un tipo di oggetto e l'altro deve avere un tipo intero.*
- Standard C11 § 6.2.5, paragrafo 19:
  - *Il tipo void comprende un insieme vuoto di valori; è un tipo di oggetto incompleto che non può essere completato.*
- Attenzione: *alcuni* compilatori (gcc, icc, clang) consentono di effettuare operazioni aritmetiche su puntatori `void *`, assumendo una dimensione del tipo pari a 1
- NON è corretto né portabile effettuare queste operazioni.

Esempi di uso dei puntatori

file: pointers.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
        // The data we will be printing
        int ages[] = { 23, 43, 12, 89, 2 };
        char *names[] = {
                "Alan", "Frank",
                "Mary", "John", "Lisa"
        };
        int count = 5;
        int i = 0;

        // first way: indexing
        for (i = 0; i < count; i++) {
                printf("%s has %d years alive.\n", names[i], ages[i]);
        }
        puts("---");
```

Esempi di uso dei puntatori

```
// second way: using pointers
int *cur_age = ages;
char **cur_name = names;
for (i = 0; i < count; i++) {
        printf("%s is %d years old.\n", *(cur_name + i), *(cur_age + i));
}
puts("---");

// third way: "offsetting" pointers
for (i = 0; i < count; i++) {
        printf("%s is %d years old again.\n", cur_name[i], cur_age[i]);
}
puts("---");
```

Esempi di uso dei puntatori

```
        // fourth way: let's make it unnecessarily complex
        for (cur_name = names, cur_age = ages;
            (cur_age - ages) < count; cur_name++, cur_age++) {
                printf("%s lived %d years so far.\n", *cur_name, *cur_age);
        }
        return 0;
}
```

Un altro esempio

file: shift.c

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

#define MAX_BITS (int)(sizeof(unsigned int) * CHAR_BIT)
char string[MAX_BITS + 1];
char *print_bits(unsigned int value);

int main(int argc, char *argv[]) {
        unsigned int a = 256, sa;
        int b = -256, sb;

        sa = a >> 2;
        sb = b >> 2;

        printf("a\t\t= %03d (%s)\n", a, print_bits(a));
        printf("after shift\t= %03d (%s)\n\n", sa, print_bits(sa));
        printf("b\t\t= %04d (%s)\n", b, print_bits(b));
        printf("after shift\t= %04d (%s)\n", sb, print_bits(sb));

        return 0;
}
```

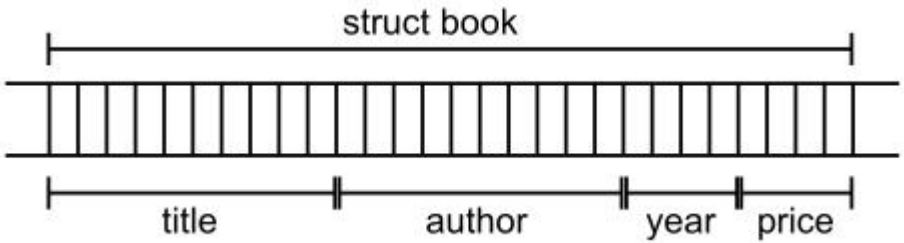
Un altro esempio

```
char *print_bits(unsigned int value)
{
        int i;
        char *p;

        p = string + MAX_BITS;
        p[0] = 0;

        for(i = 0; i < MAX_BITS; i++) {
            *--p = value & 1 ? '1' : '0';
            value >>= 1;
        }
        return string;
}
```

Strutture

- Le strutture sono un modo per *aggregare* in maniera logica tipi primitivi, altre strutture, o vettori.

```
struct book {
    char title[10];
    char author[10];
    int publication_year;
    float price;
};
```



Esempio di uso delle strutture

file: structs.c

```c
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>

struct person {
        char *name;
        int age;
        int height;
        int weight;
};

struct person *person_create(struct person *who,
                             char *name, int age, int height,
                             int weight)
{
        assert(who != NULL);

        who->name = strdup(name);
        who->age = age;
        who->height = height;
        who->weight = weight;

        return who;
}
```

Esempio di uso delle strutture

file: structs.c

```c
void person_print(struct person *who)
{
        printf("Name: %s\n", who->name);
        printf("\tAge: %d\n", who->age);
        printf("\tHeight: %d\n", who->height);
        printf("\tWeight: %d\n", who->weight);
}

int main(int argc, char *argv[])
{
        struct person joe;
        (void)person_create(&joe, "Joe Alex", 32, 64, 140);

        printf("Joe is at memory location %p:\n", &joe);
        person_print(&joe);

        joe.age += 20;
        person_print(&joe);

        return 0;
}
```

Altri modi per inizializzare una struct

```c
struct student
{
    char  name[100];
    int   roll;
    float marks;
};
```

```c
struct student stu1;

// Must match member order
struct student stu1 = { "Mario", 12, 79.5f };

// Designated struct initialization
struct student stu1 = {
    .roll  = 12,
    .name  = "Mario",
    .marks = 79.5f
};
```

`struct`/`union` di tipo incompleto

- Una struct o una union possono essere di *tipo incompleto*
- In questo caso, i loro membri non sono ancora stati definiti
- Non è possibile istanziare una variabile di tipo incompleto
- Tuttavia, è possibile definirne un puntatore

```
struct thing *pt;
```

- In questo modo è possibile risolvere riferimenti circolari tra header differenti e per realizzare strutture dati ricorsive
- Sono utili anche per *nascondere i tipi*: il tipo incompleto viene dichiarato in un header, ma il corpo solo in un file sorgente

Un esempio

list.h:

```
struct list;

bool is_empty(struct list *t);
void *get_head(struct list *t);
```

list.c

```
#include “list.h”
struct list {
    int data;
    struct list *next;
};

<implementazione delle funzioni>
```

Passaggio di parametri

- In C, il passaggio di parametri è sempre fatto *per valore*
  - La funzione riceve *una copia* del parametro
    - in caso di vettori/matrici, il passaggio per valore riguarda il puntatore associato!
  - Il motivo è evitare *side effect* nascosti nell'invocazione di funzioni (anche di libreria)

- Nel caso di struct, questo vuol dire che viene effettuata una copia di tutta la struttura
  - Se non si hanno problemi di side effect, il costo computazionale della funzione *aumenta drasticamente*

- È possibile utilizzare i puntatori per effettuare un *passaggio per riferimento* (o per indirizzo)

Passaggio di parametri

```
struct huge {
    char member[4096];
};

extern f1(struct huge s);
extern f2(struct huge *s);

struct huge glbl;

void f(void) {
    f1(glbl);
}
```

```
addiu   $sp, $sp, -4096
move    $a0, $sp
lui     $a1, %hi(glbl)
addiu   $a1, $a1, %lo(glbl)
li      $a2, 4096
jal     memcpy
move    $a0, $t0
jal     f1
addiu   $sp, $sp, 4096
```

Passaggio di parametri

```
struct huge {
    char member[4096];
};

extern f1(struct huge s);
extern f2(struct huge *s);

struct huge glbl;

void f(void) {
    f2(&glbl);
}
```

```
lui     $a0, %hi(glbl)
addiu   $a0, $a0, %lo(glbl)
jal     f2
```

Passare un array per valore

file: arr-by-val.c

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

#define SIZE 5

struct wrapper {
        int arr[SIZE];
};

void print(char *msg, int M[SIZE]) {
        int i;
        printf("%s\n", msg);
        for (i = 0; i < SIZE; ++i)
                printf("%d ", M[i]);
        printf("\n");
}

void modify(struct wrapper p) {
        int i;
        print("\nIn modify(), before changing", p.arr);

        for (i = 0; i < SIZE; ++i)
                p.arr[i] = 100;

        print("\nIn modify(), after changing", p.arr);
}
```

Passare un array per valore

file: arr-by-val.c

```
int main()
{
        int i;
        struct wrapper s;

        for (i = 0; i < SIZE; i++)
                s.arr[i] = 10;
        modify(s);

        print("\nIn main(), after calling modify()", s.arr);
        return 0;
}
```

sizeof, array e puntatori

file: sizeof.c

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
        int nums[] = { 10, 12, 13, 14, 20 };
        char name[] = {'A', 'l', 'e', 's', 's', 'a', 'n', 'd', 'r', 'o', '\0'};
        char *name_ptr = name;

        printf("The size of an int: %ld\n", sizeof(int));
        printf("The size of nums (int[]): %ld\n", sizeof(nums));
        printf("The number of ints in nums: %ld\n\n", sizeof(nums) / sizeof(int));

        printf("The size of a char: %ld\n", sizeof(char));
        printf("The size of name (char[]): %ld\n", sizeof(name));
        printf("The number of chars: %ld\n\n", sizeof(name) / sizeof(char));

        printf("The size of name_ptr: %ld\n", sizeof(name_ptr));
        printf("Bugged number of chars: %ld\n", sizeof(name_ptr) / sizeof(char));

        return 0;
}
```

Puntatori ed array sono la stessa cosa?

- Vi è una forte relazione tra vettori e puntatori
- Il nome della variabile usata per dichiarare un vettore può facilmente decadere ad un puntatore al primo elemento

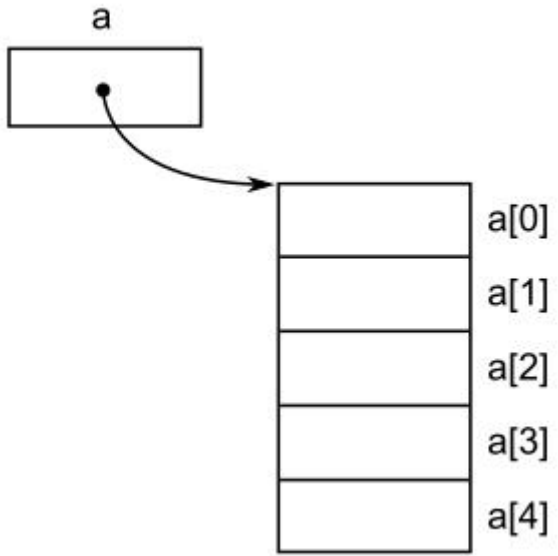
- La seguente assegnazione è perfettamente legale:

```
int a[4] = {3, 2, 1, 0};
int *p = a;
```

- Nell'esempio, `a[1]` e `p[1]` generano entrambi l'indirizzo dove il valore 2 è memorizzato

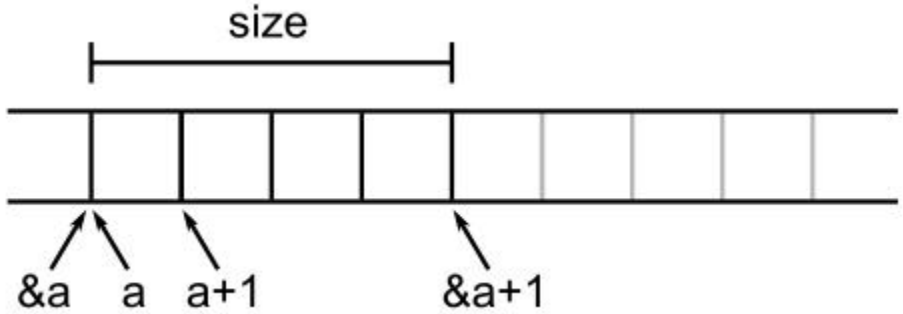
Puntatori ed array sono la stessa cosa?

- Un puntatore può quindi essere dereferenziato utilizzando l'operatore spiazzamento [ ]
- Tuttavia, l'operatore [ ] viene in questo caso valutato in maniera diversa dai vettori:
  - "*Prendi il puntatore, dereferenzialo, spiazzati da quel punto come se avessi un vettore*"
- Si tratta di una funzionalità offerta dal linguaggio per semplificare l'implementazione di operazioni su vettori e matrici



Puntatori ed array sono la stessa cosa?

- Allo stesso tempo, &a e a hanno due significati differenti
- Il tipo di a è `int *`, mentre il tipo di &a è `int (*)[size]`
  - a sarà tradotto in un puntatore al primo elemento del vettore, mentre &a verrà valutato come indirizzo di tutto il vettore
- L'aritmetica dei puntatori può dare risultati differenti:



Puntatori ed array sono la stessa cosa?

cartella: array-pointers

file1.c:

```c
#include <stdio.h>

extern int *array;

int main(void)
{
   printf("Third element of the
     array is %d\n", array[2]);
   return 0;
}
```

file2.c:

```c
int array[] = {0, 1, 2, 3, 4};
```

- Questi due file compilano in un programma correttamente
- Non generano nemmeno un warning
- Il programma è corretto?

Attenzione ai puntatori e allo scope delle variabili!

file: doomed-pointer.c

```c
#include <stdio.h>

int *buggy_return(void)
{
        int a_variable = 10;
        return &a_variable;
}

void using_stack(void)
{
        char a_string[] = "Hello World!";
        printf("%s\n", a_string);
}
```

```c
int main(int argc, char *argv[])
{
        int *dummy = buggy_return();
        printf("%d\n", *dummy);
        using_stack();
        printf("%d\n", *dummy);
        return 0;
}
```

Precedenza, associatività, ordine di valutazione

- Gli operatori hanno per definizione una precedenza
  - Questa può essere alterata mediante l'utilizzo delle parentesi
- Per definizione, hanno anche un'associatività, che determina in quale modo questi vengono computati qualora più operatori con la stessa precedenza vengono integrati in una sola espressione
- L'ordine di valutazione invece *non è definito* e dipende dal compilatore:

```
x = f() + g() // Quale funzione viene invocata per prima?
```

- Il codice seguente, quindi, è ambiguo:

```
printf("%d %d\n", ++n, power(2, n));
```

Precedenza e associatività degli operatori

Operatori	Associatività
`() [] -> .`	destra a sinistra
`! - ++ -- + - * & (type) sizeof`	destra a sinistra
`* / %`	sinistra a destra
`+ -`	sinistra a destra
`<< >>`	sinistra a destra
`< <= > >=`	sinistra a destra
`== !=`	sinistra a destra
`&`	sinistra a destra
`^`	sinistra a destra
`|`	sinistra a destra
`&&`	sinistra a destra
`||`	sinistra a destra
`?:`	destra a sinistra
`= += -= *= /= %= &= ^= |= <<= >>=`	destra a sinistra
`,`	sinistra a destra

Esempi di applicazione

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

p = b;          /* *p equivale a b[0] */
*p++ = 5;       /* b[0] = 5; *p equivale a b[1] */

p = b;
*++p = 5;       /* b[1] = 5; *p equivale a b[1] */

p = b;
*p = 5;         /* b[0] = 5 */
b[1] = ++*p;    /* b[1] = 6; b[0] = 6; *p equivale a b[0] */
```

Esempi di applicazione

Espressione	Equivale a	Significato
`*p++`	`*(p++)`	p viene incrementato dopo del suo utilizzo per accedere alla memoria.
`*++p`	`*(++p)`	p viene incrementato prima del suo utilizzo per accedere alla memoria.
`++*p`	`++(*p)`	p è utilizzato per l'accesso alla memoria, ma il contenuto della locazione viene incrementato prima del suo utilizzo.
`(*p)++`	`(*p)++`	p è utilizzato per l'accesso alla memoria; il contenuto della locazione viene incrementato dopo del suo utilizzo.

typedef

- Con la keyword typedef è possibile assegnare nomi alternativi a dei tipi di dato esistenti
- Utilizzi principali:
  - indicare meglio cosa rappresenta una variabile (`speed_t`, anziché `float`)
  - semplificazione di dichiarazioni (rendere meno prolissa la dichiarazione di una `struct`)
- Un uso estensivo può offuscare la leggibilità del codice
- Possono creare problemi nelle dipendenze circolari degli header
- Può rendere più semplice la portabilità del codice (non dovendo cambiare il tipo in tutto il sorgente)

typedef

```
struct var {
        int data1;
        int data2;
        char data3;
};

struct var a;

typedef struct var newtype;

newtype a;
```

Tipo unione

- Un'unione è un valore che può assumere una qualsiasi tra diverse rappresentazioni o formati all'interno della stessa posizione in memoria
- Si tratta quindi di un blocco di memoria che viene utilizzata per conservare, una per volta, variabili di tipo differente
- Può essere utile anche per effettuare “cast particolari”
- L’accesso ai membri delle unioni segue le stesse regole delle struct
  - operatore punto per le variabili di tipo unione
  - operatore freccia per i puntatori alle variabili di tipo unione

Tipo unione

file: union.c

```c
#include <stdio.h>

typedef enum type
{
        INT,
        FLOAT,
        DOUBLE
} type_t;

typedef union operand
{
        int i;
        float f;
        double d;
} operand_t;
```

Tipo unione

```
void sum(type_t t, operand_t op1, operand_t op2)
{
        switch(t) {
                case INT:
                        printf("%d + %d = %d\n", op1.i, op2.i, op1.i + op2.i);
                        break;
                case FLOAT:
                        printf("%.02f + %.02f = %.02f\n", op1.f, op2.f, op1.f + op2.f);
                        break;
                case DOUBLE:
                        printf("%.02lf + %.02lf = %.02lf\n", op1.d, op2.d, op1.d + op2.d);
                        break;
                default:
                        printf("Unexpected operand types.\n");
        }
}
```

Tipo unione

```c
int main(void)
{
        // Designated union initializers
        operand_t
            op11 = { .i = 3 },
            op12 = { .i = 5 },
            op21 = { .f = 2.18f },
            op22 = { .f = 8.79f },
            op31 = { .d = 1.53 },
            op32 = { .d = 4.71 };

        sum(INT, op11, op12);
        sum(FLOAT, op21, op22);
        sum(DOUBLE, op31, op32);

        printf("\nsizeof(int) = %zu\n", sizeof(int));
        printf("sizeof(float) = %zu\n", sizeof(float));
        printf("sizeof(double) = %zu\n", sizeof(double));
        printf("sizeof(operand_t) = %zu\n", sizeof(operand_t));

        return 0;
}
```

Union cast

file: union-cast.c

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

union binary_float_t {
        float real;
        uint32_t integer; // Assumes float is 32 bits wide
};

int main(void)
{
        union binary_float_t f;
        f.real = 3.141592F;
        printf("Hex representation of %f is %#04x\n", f.real, f.integer);
        return 0;
}
```

Maschere di bit

- Per risparmiare spazio, o per rispettare la struttura di alcuni registri di controllo hardware, può essere necessario memorizzare più valori all'interno di una sola parola
- I tipi del C però rispettano il modello di memoria flat
  - non si può indirizzare nulla che sia più piccolo di un byte
- Le maschere di bit permettono di rispondere a questi due differenti requisiti
- In questo modo, è possibile inserire in una singola parola più valori, associandoli ad esempio a singoli *flag*
- La composizione può essere svolta utilizzando operazioni bit a bit (*bit fliddling*), oppure utilizzando i campi di bit (*bit fields*) nelle `struct`

Maschere di bit

- Esempio di approccio classico: un frammento di un compilatore che manipola una tabella di simboli

```
// Equivalent to: enum {KEYWORD = 01, EXTERN = 02, STATIC = 04};
#define KEYWORD  01
#define EXTERN   02
#define STATIC   04

flags |= EXTERN | STATIC; // Set two bits
flags &= ~(EXTERN | STATIC); // Clear two bits
if((flags & (EXTERN | STATIC) == 0) ... // True if both are cleared
```

Maschere di bit

- Lo stesso esempio utilizzando i campi di bit

```
struct {
        unsigned int is_keyword: 1;
        unsigned int is_extern: 1;
        unsigned int is_static: 1;
} flags;

flags.is_extern = flags.is_static = 1; // Set two bits
flags.is_extern = flags.is_static = 0;  // Clear two bits
if(flags.is_extern == 0 && flags.is_static == 0) ...
```

Strutture e unioni anonime

- Dal C11, è possibile dichiarare ed utilizzare strutture e tipi unione anonimi
- Dal momento che non sono forniti dei nomi, si può accedere direttamente ai loro membri
- È necessario definirne uno *scope*, quindi queste vengono tipicamente utilizzate all'interno di altre strutture o unioni

Strutture e unioni anonime

file: anonymous-union.c

```c
#include <stdio.h>

struct scope
{
    // Anonymous union
    union
    {
        char alpha;
        int num;
    };
};

int main()
{
    struct scope x;
    x.num = 65;

    printf("x.alpha = %c, x.num = %d\n", x.alpha, x.num);

    return 0;
}
```

Strutture e unioni anonime

file: anonymous-struct.c

```c
#include <stdio.h>

struct scope
{
    // Anonymous structure
    struct
    {
        char alpha;
        int num;
    };
};

int main()
{
    struct scope x;
    x.num = 65;
    x.alpha = 'A';

    printf("x.alpha = %c, x.num = %d\n", x.alpha, x.num);

    return 0;
}
```

Qualificatori di tipo

- I qualificatori di tipo permettono di fornire al compilatore altre informazioni legate al modo in cui si intende utilizzare una certa variabile (tipo *qualificato*): esprimono delle *proprietà* dei tipi.
- **`const`** (C89): indica che i dati sono in sola lettura.
- **`volatile`** (C89): indica che un valore può cambiare tra due accessi diversi, anche se apparentemente nulla l'ha cambiato (previene ottimizzazioni del compilatore).
- **`restrict`** (C99): si utilizza solo con dichiarazioni di puntatori. Indica al compilatore che, per tutta la vita del puntatore, solo questo avrà accesso all'oggetto puntato.
- **`_Atomic`** (C11): garantisce accesso atomico ad un dato
  - si possono utilizzarei tipi **`atomic_X`** se si include l'header `stdatomic.h`

_Atomic: esempio

file: atomic.c

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

 _Atomic int acnt;
 int cnt;

 int f(void *thr_data)
 {
         for (int n = 0; n < 10000; n++) {
                 cnt++;
                 acnt++;
         }
         return 0;
 }
```

_Atomic: esempio

file: atomic.c

```
int main(void)
{
        thrd_t thr[10];

        for (int n = 0; n < 10; ++n)
                thrd_create(&thr[n], f, NULL);
        for (int n = 0; n < 10; ++n)
                thrd_join(thr[n], NULL);

        printf("The atomic counter is %u\n", acnt);
        printf("The non-atomic counter is %u\n", cnt);
}
```

Puntatori a funzione

- In C è possibile utilizzare dei *puntatori a funzione*
- Si tratta di *variabili* a cui possono essere assegnati gli indirizzi di memoria di funzioni
- Tramite questi puntatori è possibile invocare le funzioni puntate
- Sono uno degli strumenti fondamentali per la realizzazione di sistemi basati su oggetti che permettono di realizzare *overloading delle funzioni*

Puntatori a funzione

file: function-pointers.c

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

 #define ELEMENTS 6
 int values[] = { 40, 10, 100, 90, 20, 25 };

 typedef int (*compare_t)(const void *, const void *);

 /**
  * retval meaning
  * <0     The element pointed to by p1 goes before the element pointed to by p2
  * 0      The element pointed to by p1 is equivalent to the element pointed to by p2
  * >0     The element pointed to by p1 goes after the element pointed to by p2
  */
 int compare(const void *a, const void *b)
 {
         return (*(int *)a - *(int *)b);
 }
```

Puntatori a funzione

```c
void shuffle(int *array, size_t n)
{
        if (n > 1) {
                size_t i;
                for (i = 0; i < n - 1; i++) {
                        size_t j = i + rand() / (RAND_MAX / (n - i) + 1);
                        int t = array[j];
                        array[j] = array[i];
                        array[i] = t;
                }
        }
}

void print_array(char *header, int *array, size_t n)
{
        int i;

        printf("%s: ", header);
        for (i = 0; i < n; i++)
                printf("%d ", values[i]);
        puts("");
}
```

Puntatori a funzione

```
int main(int arcg, char *argv[])
{
        compare_t compare_f1 = compare;
        int (*compare_f2)(const void *, const void *) = compare;

        srand(time(0));

        print_array("Original", values, ELEMENTS);
        qsort(values, ELEMENTS, sizeof(int), compare);
        print_array("Sorted with f.name", values, ELEMENTS);
        shuffle(values, ELEMENTS);
        print_array("Shuffled", values, ELEMENTS);

        qsort(values, ELEMENTS, sizeof(int), compare_f1);
        print_array("Sorted with f.ptr 1", values, ELEMENTS);
        shuffle(values, ELEMENTS);
        print_array("Shuffled", values, ELEMENTS);

        qsort(values, ELEMENTS, sizeof(int), compare_f2);
        print_array("Sorted with f.ptr 2", values, ELEMENTS);

        return 0;
}
```

Qualificatori di tipo e puntatori

- I qualificatori di tipo possono essere utilizzati per differenziare i puntatori e l'area di memoria puntata

- Nel caso di puntatori, sono in gioco più dati:
  - Il puntatore stesso
  - Il dato puntato

- La posizione dei qualificatori impatta su cosa vogliamo qualificare
  - Ciò può rendere complesso l'interpretazione e la scrittura di tipi di dato
  - Al contempo, rende estremamente versatile il linguaggio

Alcuni esempi

- Consideriamo le seguenti dichiarazioni di tipi:

- `int *ptr`: un puntatore a intero
- `int const *ptr`: un puntatore a intero costante
- `const int *ptr`: un puntatore a intero costante
- `int * const ptr`: un puntatore costante a intero
- `int const * const ptr`: un puntatore costante a intero costante
- `const int * const ptr`: un puntatore costante a intero costante
- `int **ptr`: un puntatore a un puntatore a intero
- `int ** const ptr`: un puntatore costante a un puntatore a intero
- `int * const * ptr`: un puntatore a un puntatore costante a intero
- `int const **ptr`: un puntatore a un puntatore a un intero costante
- `int * const * const ptr`: un puntatore costante a un puntatore costante a intero

La faccenda è complicata in generale!

- `char **argv`: un puntatore a puntatore a carattere
- `const char * const * const ptr`: un puntatore costante a un puntatore costante a un carattere costante
- `volatile char * const ptr`: un puntatore costante a un carattere volatile
- `int *ptr[13]`: un vettore di 13 puntatori a intero
- `int (*ptr)[13]`: un puntatore a un vettore di 13 interi
- `void *ptr()`: una funzione che restituisce un puntatore generico
- `void (*ptr)()`: un puntatore a una funzione senza valore di ritorno
- `char (*(*x())[])()`: una funzione che restituisce un puntatore a un vettore di puntatori a funzione che restituiscono un carattere
- `char (*(*x[3])())[5]`: un vettore di tre puntatori a funzione che restituiscono un puntatore ad un vettore di cinque caratteri

La faccenda è complicata in generale!

- `char **argv`: un puntatore a puntatore a carattere
- `const char * const * const ptr`: un puntatore costante a un puntatore costante a un carattere costante
- `volatile char * const ptr`: un puntatore costante a un carattere volatile
- `int *ptr[13]`: un vettore di 13 puntatori a intero
- `int (*ptr)[13]`: un puntatore a un vettore di 13 interi
- `void *ptr()`: una funzione che restituisce un puntatore generico
- `void (*ptr)()`: un puntatore a una funzione senza valore di ritorno
- `char (*(*x())[])()`: una funzione che restituisce un puntatore a un vettore di puntatori a funzione che restituiscono un carattere
- `char (*(*x[3])())[5]`: un vettore di tre puntatori a funzione che restituiscono un puntatore ad un vettore di cinque caratteri

Seriamente?!

La regola della spirale

- La regola della spirale è una “regola empirica” che permette di decifrare qualsiasi tipo C
- La “lettura” di un tipo parte dal nome della variabile e segue una spirale in senso orario
- Ci si ferma quando si sono presi in considerazione tutti gli elementi
- Ciò che si trova fra parentesi ha la precedenza!

- Funziona molto bene con la lingua inglese (per via degli aggettivi prima del sostantivo)

La regola della spirale: esempi

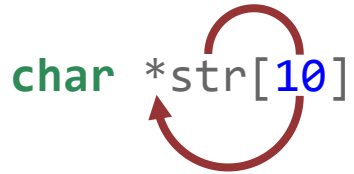
```
char *str[10]
```

str è un...

La regola della spirale: esempi

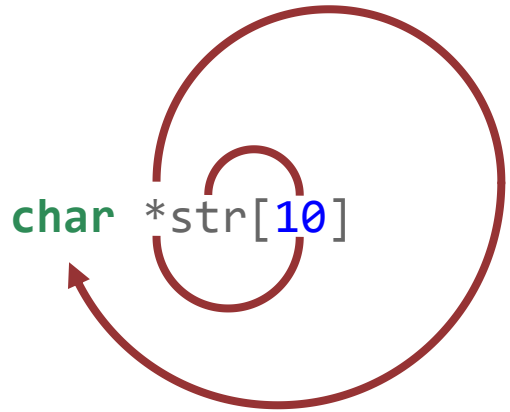
str è un vettore di 10...

La regola della spirale: esempi



str è un vettore di 10 puntatori a...

La regola della spirale: esempi



str è un vettore di 10 puntatori a carattere

La regola della spirale: esempi

```
char *(*fp)(int, float *)
```

fp è un...

La regola della spirale: esempi

fp è un puntatore a...

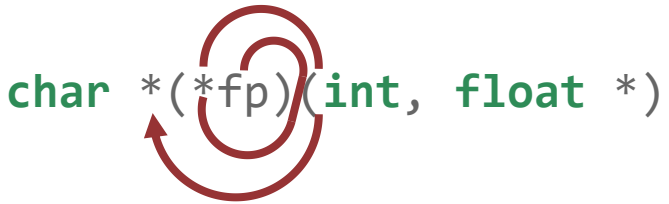
La regola della spirale: esempi

fp è un puntatore a...

La regola della spirale: esempi

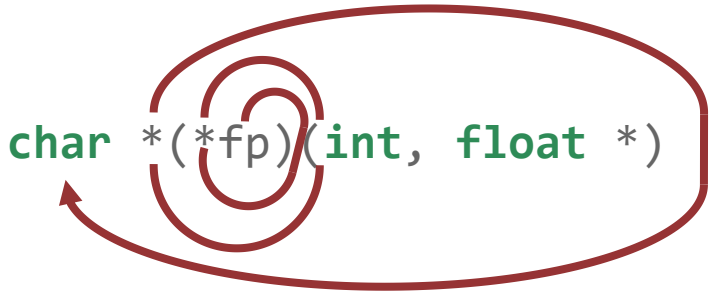
fp è un puntatore a una funzione che accetta un intero e un puntatore a float...

La regola della spirale: esempi



fp è un puntatore a una funzione che accetta un intero e un puntatore a float che restituisce un puntatore a...

La regola della spirale: esempi



fp è un puntatore a una funzione che accetta un intero
e un puntatore a float che restituisce un puntatore
a carattere

La regola della spirale: esempi

```
void (*signal(int, void (*fp)(int)))(int)
```

- `signal` è fra parentesi, quindi dobbiamo partire da lì!

`signal è...`

La regola della spirale: esempi

signal è una funzione che accetta un intero e un **??**...

La regola della spirale: esempi

signal è una funzione che accetta un intero e un **??**...

fp è un...

La regola della spirale: esempi

signal è una funzione che accetta un intero e un **??**...

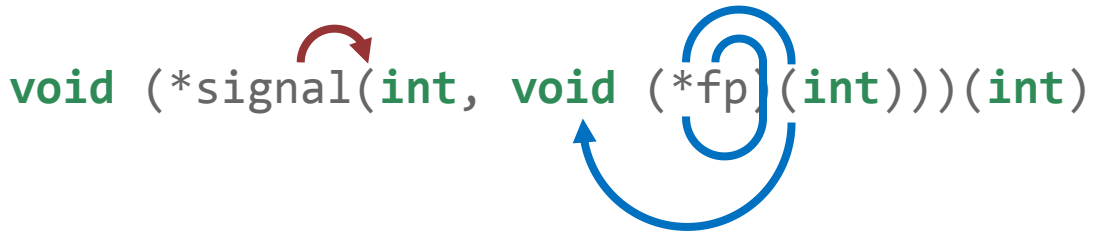
fp è un puntatore a...

La regola della spirale: esempi

signal è una funzione che accetta un intero e un **??**...

fp è un puntatore a una funzione che accetta un intero...

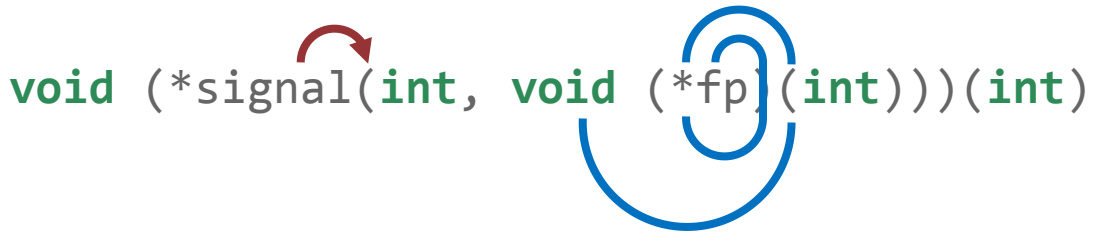
La regola della spirale: esempi



signal è una funzione che accetta un intero e un **??**...

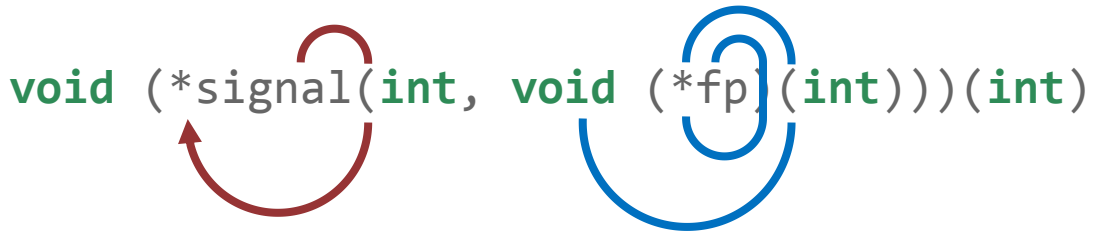
fp è un puntatore a una funzione che accetta un intero e non restituisce nulla...

La regola della spirale: esempi



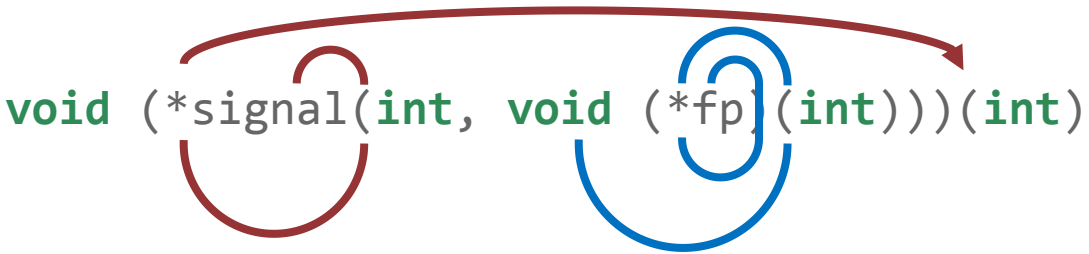
signal è una funzione che accetta un intero e un puntatore a una funzione che accetta un intero e non restituisce nulla...

La regola della spirale: esempi



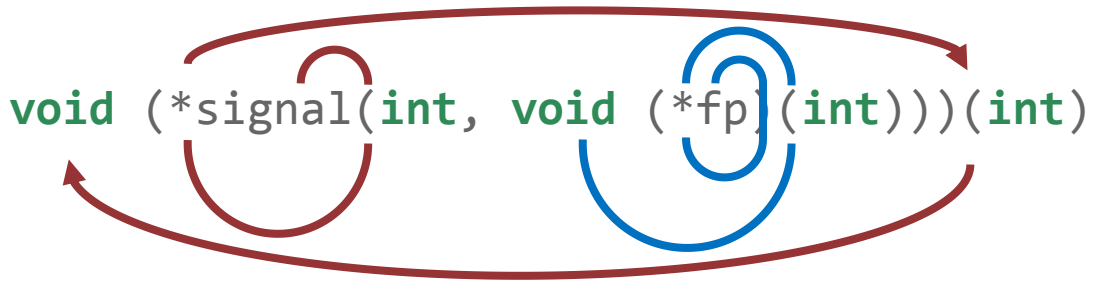
signal è una funzione che accetta un intero e un puntatore a una funzione che accetta un intero e non restituisce nulla che restituisce un puntatore a...

La regola della spirale: esempi



signal è una funzione che accetta un intero e un puntatore a una funzione che accetta un intero e non restituisce nulla che restituisce un puntatore a una funzione che accetta un intero...

La regola della spirale: esempi



signal è una funzione che accetta un intero e un puntatore a una funzione che accetta un intero e non restituisce nulla che restituisce un puntatore a una funzione che accetta un intero e non restituisce nulla

La regola della spirale: esempi

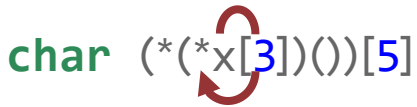
```
char (*(*x[3])())[5]
```

x è un...

La regola della spirale: esempi

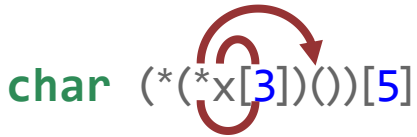
x è un vettore di tre...

La regola della spirale: esempi



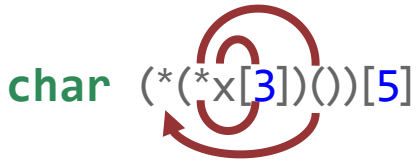
x è un vettore di tre puntatori a...

La regola della spirale: esempi



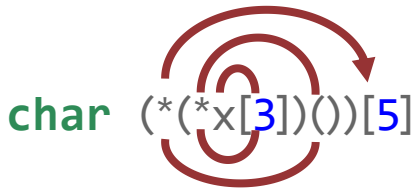
x è un vettore di tre puntatori a funzione che...

La regola della spirale: esempi



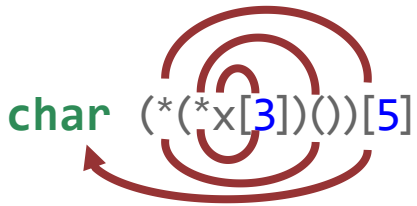
x è un vettore di tre puntatori a funzione che restituiscono un puntatore...

La regola della spirale: esempi



x è un vettore di tre puntatori a funzione che restituiscono un puntatore ad un vettore di cinque...

La regola della spirale: esempi



x è un vettore di tre puntatori a funzione che restituiscono un puntatore ad un vettore di cinque caratteri

La regola della spirale: esempi

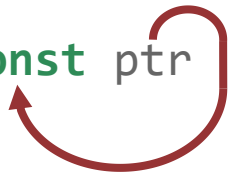
- In questo caso dobbiamo “ritoccare” quello che otteniamo!

```
volatile char * const ptr
```

ptr è un...

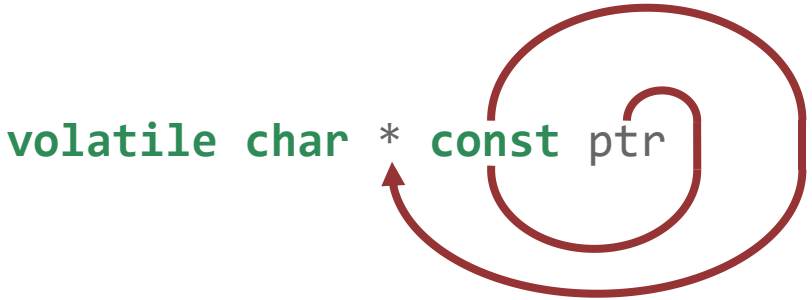
La regola della spirale: esempi

```
volatile char * const ptr
```



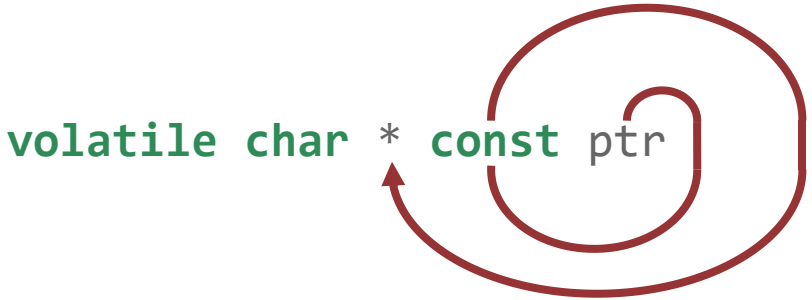
ptr è un costante...

La regola della spirale: esempi



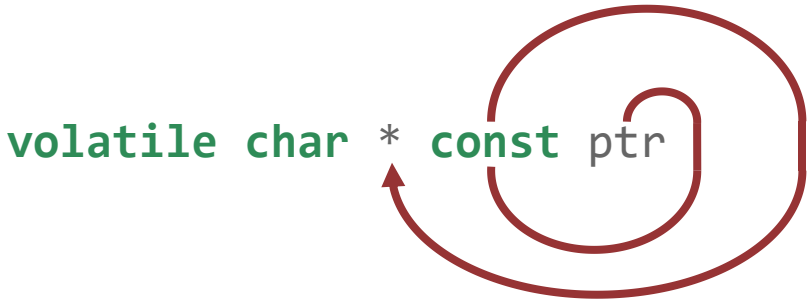
ptr è un costante puntatore...

La regola della spirale: esempi



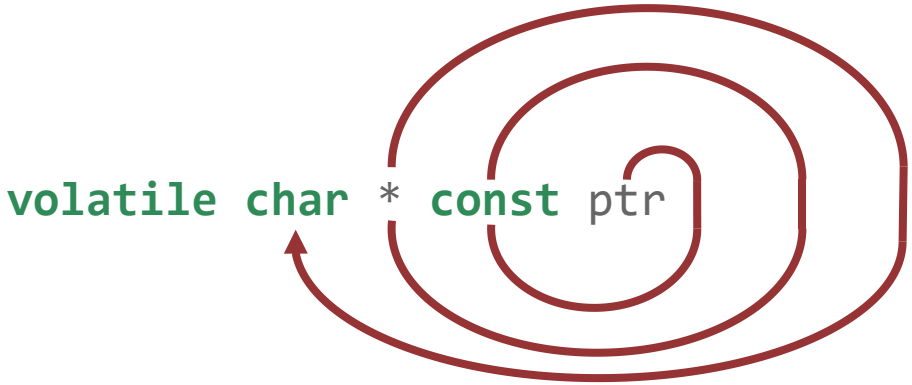
ptr è un costante puntatore (*constant pointer*)...

La regola della spirale: esempi



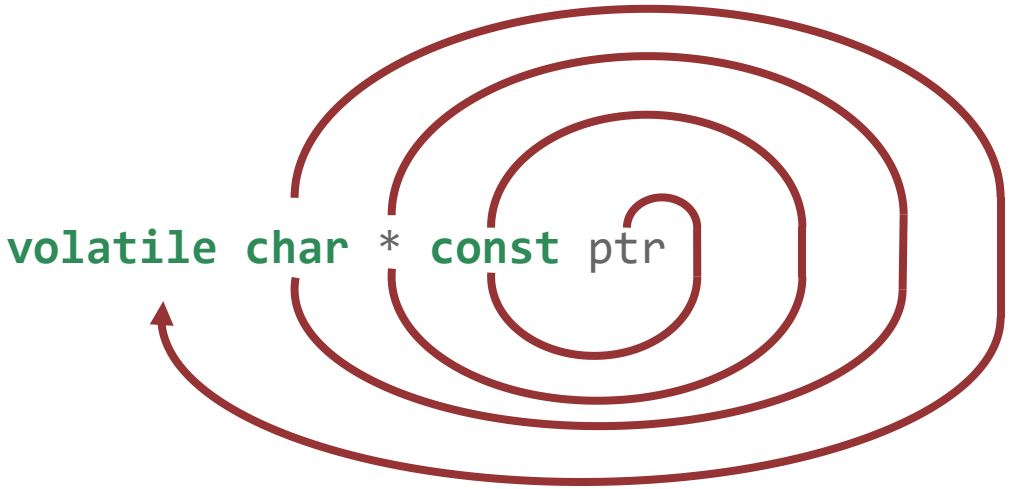
ptr è un puntatore costante a...

La regola della spirale: esempi



ptr è un puntatore costante a un carattere...

La regola della spirale: esempi



ptr è un puntatore costante a un carattere volatile

Funzioni variadiche

- Una *funzione variadica* è una funzione di *arietà* variabile
  - accetta un numero variabile di argomenti
- In C, nella definizione di tale funzione, si utilizzano tre puntini di sospensione come ultimo parametro
- In questo caso, le variabili vengono passate tramite stack
- La funzione variadica deve esplicitamente recuperare le variabili dallo stack una ad una, specificandone correttamente il tipo, sfruttando le definizioni in stdarg.h
- Un esempio è la funzione printf(), che determina in funzione della stringa di formato il tipo di ciascun parametro
- L'utilizzo delle funzioni variadiche può dare adito a problemi di sicurezza

Funzioni variadiche

file: variadic.c

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

int somma(int n, ...)
{
        int nSum = 0;

        va_list int_ptr;
        va_start(int_ptr, n);

        for (int i = 0; i < n; i++)
                nSum += va_arg(int_ptr, int);

        va_end(int_ptr);

        return nSum;
}

int main(int argc, char **argv) {
        printf("10 + 20 = %d\n", somma(2, 10, 20));
        printf("10 + 20 + 30 = %d\n", somma(3, 10, 20, 30));
        printf("10 + 20 + 30 + 40 = %d\n", somma(4, 10, 20, 30, 40));

        return EXIT_SUCCESS;
}
```

Scrivere codice elegante: l'esempio di una lista

```
struct node_t {
  int key;
  node_t * next;
};

void remove_entry(struct node_t **const head, struct node_t const *const entry)
{
  struct node_t *prev = NULL;
  struct node_t *walk = *head;
  while(walk != entry) {
    prev = walk;
    walk = walk->next;
  }

  if(!prev) {
    *head = entry->next;
  } else {
    prev->next = entry->next;
  }
}
```

Scrivere codice elegante: l'esempio di una lista

```
struct node_t {
  int key;
  node_t * next;
};

void remove_entry(struct node_t **head, struct node_t const *const entry)
{
  while((*head) != entry) {
    head = &(*head)->next;
  }
  *head = entry->next;
}
```

Esempio riassuntivo: Duff Device

file: duff.c

```
int duff_device(char *from, char *to, int count)
{
        int n = (count + 7) / 8;
        switch (count % 8) {
                case 0:
                do {
                        *to++ = *from++;
                case 7:
                        *to++ = *from++;
                case 6:
                        *to++ = *from++;
                case 5:
                        *to++ = *from++;
                case 4:
                        *to++ = *from++;
                case 3:
                        *to++ = *from++;
                case 2:
                        *to++ = *from++;
                case 1:
                        *to++ = *from++;
                } while (--n > 0);
        }
        return count;
}
```