C89 Notes

Table of contents

Built-in Types

I data types predefiniti di C89 sono:

Data type	Descrizione
`char`	Permette di contenere un carattere della tabella ASCII che corrisponde ad un intero [0, 250]
`int`	Numeri interi
`float`	Numeri decimali a singola precisione
`double`	Numeri decimali a doppia precisione

Sintassi generale per dichiarazione singola `keywordTipo nomeVariabile1;` e dichiarazione multipla `keywordTipo nomeVariabile1, nomeVariabile2;`.

```
int socialSecurityNumber;
int age, numOfPeople;
```

É possibile inizializzare simultaneamente alla dichiarazione `keywordTipo nomeVariabile1 = valIniziale;`.

```
int carPlate = 634, numOfCars = 11;
```

> Ogni variabile deve essere **dichiarata** prima di essere utilizzata.

I caratteri alfanumerici vanno racchiusi tra apici singoli:

```
char randomAlhpaNum = '2';
randomAlphaNum = 'a';
randomAlphaNum = 'A';
randomAlphaNum = 'z';
```

Operatori aritmetici

La divisione con l'operatore `/` assume diversi significati a seconda degli operandi. Fra tipi `int` calcola il quoziente troncato della parte frazionaria, mentre fra `float` calcola il quoziente compreso di parte frazionaria. Altri operatori built-in sono `+ - * %`.

> `%` vs `/` » L'operatore aritmetico modulo (`%`) calcola una divisione troncando il quoziente, ad esempio `17 % 5` sarà diverso da `17 / 5`, i risultati sono rispettivamente `2` e `3`.

Istruzioni ingresso ed uscita

Per la scrittura su standard output si utilizza `printf()` mentre per la lettura da standard input viene usato `scanf()`.

```
int a = 3, b = 2;

printf("Hello World\n");
scanf("%d%s", &someInteger, &someString);
/* 'someInteger' e 'someString' dichiarate in precendenza! */
printf("\n %d + %d = %d", a, b, a + b);
```

Stringa di controllo	Descrizione	Carattere di stampa	Carattere di conversione
`\n`	Carriage return	°	
`\t`	Tab	°	
`%d`	Intero decimale		°
`%f`	Numero reale		°
`%c`	Carattere		°
`%s`	Sequenza di caratteri (stringa)		°

Struttura di un programma in C

```
#include <stdio.h>

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

    print("Hello World!\n");
    return 0;
}
```

> La libreria `stdio.h` (standard input/output) contiene funzioni per la gestione dell'input e dell'output fra cui `printf()` e `scanf()`.

Esercizio » Scrivere un programma che prenda in ingresso una temperatura in Fahrenheit e la trasformi in Celsius.

```
#include <stdio.h>

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

    int celsiusTemp; float fahrTemp;

    printf("Inserire valore della temperatura in gradi Fahrenheit: ");
    scanf("%f", &fahrTemp);

    celsiusTemp = (fahrTemp - 32) / 1.8;

    printf("Gradi Celsius: %d\n", celsiusTemp);

    return 0;
}
```

Operatori ed espressioni logiche

Operatore	Descrizione	Operatore logico	Operatore relazionale
==	Equal to		°
!=	Not equal to		°
> >=	Greater, greater and equal		°
< <=	Less, less and equal		°
!	NOT	°	
&&	AND	°	
||	OR	°	

La precedenza degli operatori è la seguente: `! + - & > * / % > < <= >= > > == != > && > || > =`. É possibile inoltre fare un assegnazione di tipo logico alle variabili in modo tale che una volta mettendo solamente il valore della variabile in un'unica espressione caratterizzata solamente dall'identifier della variabile, vale che per ogni valore diverso da zero assegnatovi l'espressione è vera.

```
int flag;

flag = 0; /* falso */
flag = 1; /* vero */
flag = 25; /* vero */
flag = 239; /* vero */
flag = (23 > 21) /* vero */
```

NB: da usare solo con le variabili di tipo `int`.

Durante la costruzione di tabelle di verità si noti che per n variabili logiche si hanno 2n righe. Di seguito le tabelle di verità delle espressioni logiche elementari:

NOT

A	!A
0	1
1	0

Il NOT è un operatore unario, che prende in ingresso una sola espressione.

AND

A	B	A && B
0	0	0
0	1	0
1	0	0
1	1	1

A && B è vero se e solo se sia A che B sono vere.

OR

A	B	A || B
0	0	0
0	1	1
1	0	1
1	1	1

A || B è vero se almeno una delle due è vera. NB: non è exclusive OR (XOR).

Esempio di tabella di verità composta:

A	B	C	!B	A && !B	A && !B || C
0	0	0	1	0	**0**
0	0	1	1	0	**1**
0	1	0	0	0	**0**
0	1	1	0	0	**1**
1	0	0	1	1	**1**
1	0	1	1	1	**1**
1	1	0	0	0	**0**
1	1	1	0	0	**1**

Leggi di De Morgan:

!(A && B) == !A || !B

A	B	A && B	!(A && B)
0	0	0	**1**
0	1	0	**1**
1	0	0	**1**
1	1	1	**0**

A	B	!A	!B	!A || !B
0	0	1	1	**1**
0	1	1	0	**1**
1	0	0	1	**1**
1	1	0	0	**0**

Le due tabelle di verità verificano l'equivalenza !(A && B) == !A || !B.

!(A || B) == !A && !B

A	B	A || B	!(A || B)
0	0	0	**1**
0	1	1	**0**
1	0	1	**0**
1	1	1	**0**

A	B	!A	!B	!A && !B
0	0	1	1	**1**
0	1	1	0	**0**
1	0	0	1	**0**
1	1	0	0	**0**

Le due tabelle di verità verificano l'equivalenza !(A || B) == !A && !B.

In C vi sono costrutti condizionali come `if`, `switch` e costrutti iterativi come `while`, `do while` e `for`.

Sintassi del costrutto `if`:

```
if(expression) {

    corpo_if;

} else {

    corpo_else;
}
```

NB: `else` è opzionale.

Sintassi costrutti `if` ed `else if` nested.

```
if(expressionOne) {

    someInstruction;
    if(nestedExpression) {

        nestedInstruction;

    } else {

        if(elseBranchExpressionNested) {

            doSomething;
        }
        anotherNestedInstruction;
    }

} else if(elseBranchExpression)  { /* another form for 'else if' */

    doAnything;
}
```

Esercizio » Scrivere un programma che prenda in ingresso un intero positivo e che determini se il numero corrisponda ad un anno bisestile o meno.

```
#include <stdio.h>

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

    int inputYear;

    printf("Inserire l'anno: ");
    scanf("%d", &inputYear);

    if( (((inputYear % 4) == 0) && ((inputYear % 4) != 100)) ||
        ((inputYear % 400) == 0) ) {

        printf("%d è un anno bisestile!", inputYear);


    } else {printf("%d NON è un anno bisestile!", inputYear);}

    return 0;
}
```

Sintassi del costrutto iterativo `while`:

```
while(expression) {

    someInstruction;
}
```

Esercizio » Stampa i primi 100 numeri naturali pari.

```
#include <stdio.h>

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

    int indexLoop = 1;

    while(indexLoop <= 50) {
```

```
        printf("%d\n", indexLoop*2);
        indexLoop++;
    }

    return 0;
}
```

Esercizio » Eseguire la somma e la media di una sequenza di numeri naturali inseriti dall'utente e continuare finchè l'utente inserisce 0.

```
#include <stdio.h>

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

    int sumOfNumbers = 0, inputNumber, inputCounter = 0;
    float average;

    printf("Inserire numero: ");
    scanf("%d", &inputNumber);

    while(inputNumber != 0) {

        inputCounter++;
        sumOfNumbers += inputNumber;
        /* == someOfNumbers = someOfNumbers + inputNumber */

        printf("\nInserire numero: ");
        scanf("%d", &inputNumber);
    }

    average = (sumOfNumbers * 1.0) / (inputCounter );

    printf("La somma totale è: %d\n", sumOfNumbers);
    printf("La media totale è: %f\n", average);

    return 0;
}
```

Il costrutto `do while` garantisce l'esecuzione del corpo del `while` almeno una volta:

```
do {

    somethingFun;

} while(expression);
```

> Ovviamente anche il corpo del `while` può essere provvisto di strutture di controllo annidate con `if` o usando lo stesso `while`.

Sintassi del costrutto iterativo `for`:

```
for(istruzioneInizializzazione; expression; istruzioneLoop) {

    corpo_for;
}
```

> Il ciclo viene regolato da una variabile di loop la quale viene inizializzata nel primo argomento del ciclo `for` (`istruzioneInizializzazione`) e l'incremento viene regolato dall'ultimo argomento del ciclo (`istruzioneLoop`). NB: la variabile di loop dev'essere dichiarata prima del costrutto iterativo `for` e non dentro il primo argomento.

Sintassi costrutto `switch case`

```
switch(integerExpression) {
```

```
    case firstValue:
        statementOne;
        break;

    case secondValue:
        statementTwo;
        break;

    case thirdValue:
        statementThree;
        break;

    default:
        defaultStatement;
        break;
}
```

`integerExpression` può essere anche un carattere (e non altro) e non solo `int`. Si noti inoltre come ogni label del `case` finisca con i due punti `:` e non col punto e virgola `;`. Per evitare l'esecuzione a cascata alla prima corrispondenza trovata, occorre inserire alla fine degli statements di ogni case la keyword `break`. Il `default` è opzionale, se nessuno degli altri casi viene matchato il `default` viene eseguito se è presente nel sorgente. Il `break` può essere utilizzato non solo negli `switch case` ma anche nei costrutti iterativi elementari come `while`, `do while` e `for`.

Utilizzo delle keyword `break` e `continue` nei costrutti iterativi elementari:

```
for(index = 0; index < 10; index++) {

    scanf("%d", &randomInteger);

    if(x < 0) {

        break;
    }

    printf("%d", &randomInteger);
}

index = 0;

while(index < 10) {

    scanf("%d", &randomInteger);

    if(x < 0) {

        continue;
    }

    printf("%d", &randomInteger);
    index++;
}
```

Nel ciclo `for` del primo esempio se il numero acquisito è minore di zero il `break` termina ed esce dal costrutto iterativo `for` anche se si trova all'interno dell'`if`. Nel ciclo `while` del secondo esempio il `continue` fa saltare le successive istruzioni sotto di lui rimanendo nel ciclo finchè l'espressione di controllo non fallisce.

Costanti

Dichiarando una costante viene stabilito un valore definitivo e non rimpiazzabile ad un identificatore. Nel caso in cui si provi ad assegnare un valore ad una costante dopo averla già inizializzata il compilatore darà errore. Le costanti sono utili per non avere un codice ripetitivo. La sintassi generica per definire una costante è `const typeKeyword identifier = value;`.

```c
const float pi = 3.14;
const int fixedNumber = 23;
const char serialChar = 'G';
```

Array

Gli array sono sequenze di variabili omogenee, ovvero tutte le variabili della sequenza sono dello stesso type ed ogni elemento della sequenza è individuato da un indice. La sintassi del costruttore generale è `tipo identificatoreArray[Dimensione]`.

```c
int carPlate[7];
char codiceFiscale[16];
float coordinates[30];
```

> Le dimensioni specificate degli array sono numeri fissi e non è possibile modificarle durante l'esecuzione.

Ogni elemento dell'array è una variabile del tipo dell'array ed è possibile accedere ad essi specificandone un indice fra parentesi quadre `[]`:

```c
int someIntegers[34];
someIntegers[0]; /* primo elemento */
someIntegers[25];
someIntegers[33]; /* ultimo elemento */
```

Esempi di operazioni su array:

```c
/* assegnazione */
someIntegers[2] = 23;
someIntegers[31] = 12 % 4;
someIntegers[20] = someIntegers[20 + 6];

/* operazioni logiche */
someIntegers[4] == someIntegers[13];
someIntegers[15] > someIntegers[9];

/* operazioni aritmetiche */
someIntegers[0] = someIntegers[17] / someIntegers[16];

/* operazioni I/O */
int index;
scanf("%d", &someIntegers[index]);
printf("Valore all'indice %d = %d", index, someIntegers[index]);
```

Errore da non fare con il costruttore array:

```c
int dimension;

scanf("%d", &dimension);
float wrongArr[dimension];
```

> Questo metodo di costruzione non è standard ANSI C 89, spesso infatti si ricorre alla direttiva di precompilazione `define` per dichiarare la dimensione di un array.

Esercizio » Chiedere all'utente di inserire un array di interi (dimensione definita da `effectiveController`) ed un numero intero `integerFlag`. Il programma salva gli elementi inseriti in `arrOne` e copia tutti gli elementi di `arrOne` che sono maggiori di `integerFlag` in un secondo vettore `arrTwo`. La copia deve avvenire nella parte iniziale di `arrTwo` senza lasciare buchi.

```c
#include <stdio.h>

#define ARR_ELEMENTS 30

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

    int arrOne[ARR_ELEMENTS], arrTwo[ARR_ELEMENTS], effectiveController = 10,
        integerFlag, indexLoop, subIndexLoop = 0;
```

```
    printf("Inserire l'integer flag: ");
    scanf("%d", &integerFlag);

    /* riempimento di entrambi gli array*/
    for(indexLoop = 0; indexLoop < effectiveController; indexLoop++) {

        printf("\nInserire l'elemento dell'indice %d: ", indexLoop);
        scanf("%d", &arrOne[indexLoop]);

        /* condizione di riempimento del secondo array*/
        if(arrOne[indexLoop] > integerFlag) {

            arrTwo[subIndexLoop] = arrOne[indexLoop];
            subIndexLoop++;
        }
    }

    /* stampa array */
    printf("\n[ ");

    for(indexLoop = 0; indexLoop < effectiveController; indexLoop++) {

        printf("%d  ", arrOne[indexLoop]);
    }

    printf("]\n");
    printf("\n[ ");

    for(indexLoop = 0; indexLoop < subIndexLoop; indexLoop++) {

        printf("%d  ", arrTwo[indexLoop]);
    }

    printf("]\n");



    return 0;
}
```

Array di caratteri e stringhe

In c le stringhe sono realizzate mediante array di caratteri. La differenza fra una stringa ed un array di caratteri è il terminatore della stringa `'\0'`. Se mi viene chiesto di gestire stringhe al più `N` caratteri devo allocare un vettore con dimensione `N+1`.

```
#define N 50

char firstAnswer[N + 1];
```

> Viene definito un array di `50` caratteri di tipo `char` più un `'\0'` che rappresenta la fine della stringa. NB: non esiste il tipo predefinito `string`.

Una modalità di riempimento dell'array di caratteri per creare una stringa può avvenire tramite un ciclo `for` (anche se non funziona e bugga) o tramite `scanf()`.

```
#define N 100

char randomString[N];

/* primo metodo */
scanf("%s\n", randomString);
```

```c
/* secondo metodo */
scanf("%[^\n]", randomString);

printf("%s\n", randomString);
```

Il primo metodo acquisisce la stringa fino al primo spazio mentre mentre nel secondo metodo acquisisce fino all'inserimento del primo invio. NB: `randomString` è l'indirizzo del primo elemento `&randomString[0]`, `scanf()` quindi non ha bisogno della `&`. Entrambi delimitano automaticamente la parte significativa dal terminatore `'\0'`.

É possibile calcolare la lunghezza di una stringa con un ciclo usando la flag `'\0'`:

```c
char randomString[30];
int countChar = 0;

scanf("%[^\n]", randomString);

while(randomString[countChar] != '\0') {countChar++;}

printf("Lunghezza stringa: %d\n", countChar);
```

Un metodo più semplice è utilizzare la funzione `strlen()`, contenuta nella libreria `string`:

```c
#include <string.h> /* !! importare libreria !! */

int countChar = strlen(randomString);
```

Esercizio » Verificare se due stringhe coincidono per lunghezza e se coincidono per ogni elemento.

```c
#include <stdio.h>
#include <stdlib.h> /* libreria per exit() */
#include <string.h> /* libreria per strlen() */

#define C 100

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

    char firstString[C], secondString[C];
    int arrChecker, flag;

    printf("Inserire due stringhe: ");
    scanf("%[^\n] %[^\n]", firstString, secondString);

    /* confronto lunghezza */
    if( strlen(firstString) == strlen(secondString) ) {

        /* confronto indice per indice */
        for(arrChecker = 0; arrChecker < ((int) strlen(firstString)); arrChecker++) {

            if(firstString[arrChecker] != secondString[arrChecker]) {

                printf("Stringhe lunghe uguali ma diversi caratteri\n");
                exit(0);
            }
        }

        printf("Stringhe identiche, %d caratteri\n", (int) strlen(firstString));

    } else {

        printf("Stringhe diverse!\n");
    }

    return 0;
```

```
}
```

É possibile eseguire la copia del contenuto di un array ad un altro chiamando `strcpy()` con gli array come argomenti. Il primo argomento è l'array copiato, mentre il secondo argomento è l'array copiante. Per accodare le stringhe si usa `strcat()`; ha lo stesso verso degli argomenti come per copiare. Per acquisire interattiva di una stringa completa di testo si può utilizzare la funzione di libreria `gets()` per ovviare alle limitazioni sui separatori possedute da `scanf()`. I sottoprogrammi citati in questo paragrafo fanno parte della liberia `string` a parte `scanf()` e `gets()` che fanno parte di `stdio`.

```
char myCoolLine[80];

printf("Type something cool => ");
gets(myCoolLine);
```

Array multidimensionali

Si possono definire array con più di una dimensione mediante il classico costruttore array.

```
int randomMatrix[10][12];
char matrixWords[3][20];
int triMatrix[21][15][4];
```

Ricorsione

- Uno o più casi base hanno soluzione immediata e non ricorsiva
- Tutti gli altri casi possono essere ridefiniti in funzione di problemi che siano più vicini ai casi base.
- Applicando questo processo di ridefinizione viene chiamato il sottoprogramma ricorsivo: alla fine il problema si riduce alla semplice soluzione di un caso base.

Di solito gli algoritmi ricorsivi utilizzano un costrutto condizionale strutturato nel modo seguente:

```
Se questo è un caso base {

    risolvilo

} Altrimenti {

    Ridefinisci il problema utilizzando la ricorsione
}
```

Prendiamo un esempio semplice di moltiplicazione, dobbiamo moltiplicare `6 * 3`, ipotizzando di avere la tabellina delle somme e non delle moltiplicazioni. Inoltre sappiamo che l'elemento neutro nella moltiplicazione è `1`, quindi se ci imbattiamo nel caso base di moltiplicazione per `1`, siamo in grado di risolverlo. Il problema principale può essere scomposto in sue sottoproblemi:

1. Moltiplica 6 per 2
2. Somma 6 al risultato del problema 1

Conoscendo solo la tabellina delle somme, possiamo risolvere il problema 2 ma non il problema 1. Notando bene il primo problema ora è più vicino al caso base di quanto lo fosse il problema iniziale e possiamo scomporre il problema 1 in un dei sottoproblemi analoghi a questi di prima.

1. Moltiplica 6 per 2 1.1 Moltiplica 6 per 1 1.2 Somma 6 al risultato del problema 1.1
2. Somma 6 al risultato del problema 1.

In codice:

```
#include <stdio.h>

int ricMultiply(int m, int n) {

    int res;

    /* caso base */
    if(n == 1) {

        res = m;
```

```
    } else {

        /* passo ricorsivo */
        res = m + ricMultiply(m, n - 1);
    }

    return(res);
}

int main(int argc, const char* argv[]) {

    printf("%d\n", ricMultiply(6, 3));

    return (0);
}
```

Altro esempio spiegato di utilizzo di ricorsione in questo caso per trovare quanto volte un carattere è presente in una stringa:

```
#include <stdio.h>

int count(char ch, char* str) {

    int res;

    if(str == '\0') {

        res = 0;

    } else {

        if(ch == str[0]) {

            res = 1 + count(ch, &str[1]);

        } else {

            res = count(ch, &str[1]);
        }
    }

    return(res);
}


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

    char str[] = {"POSSESSOPALLE"};
    char target = 'S';

    printf("%d\n", count(target, str));
}
```

NB: nel sottoprogramma `count()` si può notare come non si utilizzi un indice nella chiamata ricorsiva. Si utilizza un costrutto del genere: `&str[1]` che potrebbe essere all'inizio controintuitivo poichè può sembrare che ci stiamo riferendo sempre alla stessa cella di memoria, ovvero la quella con indice `[1]`. Invece si noti come nella chiamata ricorsiva si utilizzi l'ndirizzo corrente iniziale di zero riferendoci alla cella di memoria successiva. Nella chiamata a venire, l'indirizzo `&str[1]` memorizzerà nel puntatore `const char* str` l'indirizzo della cella subito accanto, poichè non stiamo lavorando sull'indirizzo dell'array originale poichè non sarebbe in linea con le regole della visibilità delle variabili locali definite dal linguaggio C, ma stiamo lavorando al puntatore della cella che viene richiamato ogni volta sull'indirizzo seguente e non chiaramente sull'indirizzo globale `[1]` come si potrebbe pensare.

Altro esercizio sulla ricorsione:

```c
/*
    Scrivere una funzione ricorsiva che calcola la somma di tutti gli interi
    compresi tra due argomenti passati come parametri.
*/

#include <stdio.h>

int sumBetween(int a, int b) {

    if(a == b) {

        return (a);

    } else {

        return (a + sumBetween(a + 1, b));
    }
}

void wrapper(void) {

    printf("%d\n", sumBetween(3, 12));
}

int main() {

    wrapper();

    return(0);
}
```

Strutture e Unioni

Gli array aggregano variabili omogenee in una sequenza precisa. Le `struct` permettono di aggregare variabili eterogenee in una sola variabile ed è una sorta di "contenitore" per variabili disomogenee di tipi più semplici. Le variabili aggregate nella struct sono dette campi della struct.

```c
struct nomeStruttura{

    keywordTipoUno nomeCampoUno, sameTypeField, anotherField;
    keywordTipoDue nomeCampoDue;
    ...
    keywordTipoN nomeCampoN;

};
```

> É possibile inoltre dichiarare una o più variabili dalla stessa struttura con al fondo `} nomeUno, nomeDue, ... ;`.
> NB: una struttura non può avere come membri variabili che hanno come tipo la struttura stessa ma può contenere puntatori a tali variabili.

```c
struct anotherStructure {

    keyword base1, membroDue;
    struct anotherStructure membroUno; /* ERRORE */
    struct anotherStructure *membroUno; /* OK */
};
```

Dichiarazione Struct

la dichiarazione alloca spazio di memoria a differenza della sola definizione del tipo strutturato e come con i datatype builtin di C si può inizializzare in modo separato.

```
struct dataContainer {

    ...

} mazzo[32], data, copy, *cartaP;
```

L'inizializzazione di una `struct` può avvenire in modo completo o separando dichiarazione ed inizializzazione come detto in precedenza

```
/* comleta */
struct dataContainer data = {"Kevin", "Malone", 25};

/* con questa inizializzazione si possono mettere anche in un ordine diverso */
struct dataContainer data = {.name = "Kevin", .surname = "Malone", .age = 25};

/* memory copy */
struct dataContainer copy = data;

/* inizializzazione separata */
struct dataContainer data;
data.name = "Morty";
data.surname = "Mortimer";
...
```

Quando una struct viene passata come parametro in ingresso ad un sottoprogramma, tutti i valori dei suoi campi vengono copiati nel corrispondente parmetro formale del sottoprogramma. NB: differentemente dai tipi di dato strutturati come gli array, i quali non possono essere utilizzati come valore di ritorno di un sottoprogramma, le struct possono invece essere ritornate.

```
planet_t get_planet(void) {

    planet_t planet;

    scanf("%s%lf%d%lf%lf", planet.name,
                           &planet.diameter,
                           &planet.moons,
                           &planet.orbit_time,
                           &planet.rotation_time);

    return(planet);
}
```

Accesso alle struct

Può avvenire per accesso diretto o tramite puntatore. Nel primo caso si utlizza l'operatore punto `.` come utilizzato nel'esempio di codice precedente e tramite un puntatore si utilizza l'operatore freccia `->`.

```
/* accesso diretto */
struct dataContainer data;
printf("%s", data.surname);

/* accesso via puntatore */
struct dataContainer *dataPointer = &data;
printf("%s", dataPointer->name);
```

> `dataPointer->name` e `(*dataPointer).name` sono equivalenti.

Si può passare una struttura interamente o solo i singoli membri. Utilizzando il metodo classico si usa l'operatore `.` mentre se si passano le strutture per riferimento si deve passare l'indirizzo della variabile `struct` (`&`) e la funzione riceve come parametro formale un puntatore alla variabile `struct`. NB: per riassegnare valori stringa in una `struct` devo ricorrere a `strcpy()`.

Tipi di dato User-Defined

le keyword `typedef` permette di definire nuovi tipi di C. La dichiarazione di nuovi tipi va prima del `main()`. Qualche esempio:

```c
#define LEN_STR 20
#define NUM_PIANI 20 Accesso alle struct
#define NUM_UFFICI 40

typedef struct {

    char name[LEN_STR + 1], surname[LEN_STR + 1];
    int mortgage;
    int index;
} Worker;

typedef struct {

    int surface;
    int exposition;
    Worker occuping;
} Office;

Office tower[NUM_PIANI][NUM_UFFICI];
```

Array di Struct

Il modo per creare questo particolare tipo di strutture è definire un tipo strutturato che collezioni le informazioni riguardanti il settore di dati di interesse e poi dichiarare un array di quel tipo.

```c
#define MAX_STU 50
#define NUM_PTS 10

typedef struct {

    int id;
    double avg;

} student_t;

typedef struct {

    double x, y;

} point_t;

...

{
    student_t stuList[MAX_STU];
    point_t polygon[MAX_PTS];
```

Per accedere ai vari campi ad esempio `stuList`, i dati del primo studento saranno nella struttura `stuList[0]`, i cui campi sono `stuList[0].id` e `stuList[0].avg`. Esempio nella tabella:

index	.id	.avg
stuList[0]	645189864	29.3
stuList[1]	987456561	25.9
stuList[...]	...	...
stuList[49]	146454546	28.5

Unioni

Le unioni sono molto simili alle struct poichè il formato della definizione di unione è analoga a quella della struttura.

```c
typedef union {

    int x, y;
    char z[32];

} Data;
```

Questo tipo ha in alternativa dentro l'unione `x` o `y` o `z[32]` e dispone la memoria in base al dato più grande, in questo caso `z[32]`. Quindi la differenza principale fra le Union e le Struct è che le prime predispongono lo spazio solamente per uno solo dei membri di quella struttura, che si riferisce al dato di dimensione più grande.

Sottoprogrammi

I sottoprogrammi venogono definiti per tipo che returnano o se non returnano alcun tipo. Le variabili locali definite in un sottoprogramma restano inglobate nel sottoprogramma a meno che non ne venga esportato il valore con una istruzione `return`. Non è violazione di standard balzare la function prototype mettendo in stack i sottoprogrammi prima del main in modo tale che il compilatore conosca già la pre-definition dei vari sottoprogrammi.

```c
double anotherSubProgram(double callerOne) {/* definizione */}
void reimbursement_calc(void) {/* definizione */}

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

    double numOne;

    reimbursement_calc();
    anotherSubProgram(numOne);

    return 0;
}
```

Puntatori e programmazione modulare

```c
float temp;
float *point = &temp; /* salva l'indirizzo di temp nel puntatore point */
```

> Identifica `point` come una variabile puntatore, in particolare un puntatore a `float`. Questo significa che possiamo memorizzare l'indirizzo di memoria di una variabile di tipo `float` in `point`.

Vari utilizzi dei pointers anche con i files che non ho voglia di spiegare, *let the code explains itself*:

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

void separate(double valueToAnalyze, double *fracP, char *signP, int *wholeP) {

    if(valueToAnalyze < 0) {

        *signP = '-'

    } else if(valueToAnalyze == 0) {

        *signP = ' ';

    } else {

        *signP = '+';

    }

    *wholeP = floor(fabs(valueToAnalyze));
    *fracP = fabs(valueToAnalyze) - floor(fabs(valueToAnalyze));
}
```

```
void separate_value() {

    double value, frac;
    char sign;
    int whole;

    printf("Inserire un valore da analizzare => ");
    scanf("%lf", &value);

    separate(value, &frac, &sign, &whole);

    printf("Parts of %.4f\n Sign: %c\n Whole number magnitude: %d\n Fractional part: %.4f\n", value, sign, who
}

void files_read_n_write(void) {

    FILE *fWritePointer;
    FILE *fReadPointer;
    char myLine[100];

    fWritePointer = fopen("writingFile.txt", "w");
    fReadPointer = fopen("anotherFile.txt", "r");

    /* write to a file */
    fprintf(fWritePointer, "HOLA SOY DORA\n");

    /* reading from a file */
    while(!feof(fReadPointer)) {

        fgets(myLine, 100, fReadPointer);
        puts(myLine);
    }

    fclose(fReadPointer);
    fclose(fWritePointer);
}

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

        /*files_read_n_write();*/
        separate_value();


    return 0;
}
```

Sintassi di inizializzazione di un array di puntatori:

```
char *strP[10];
```

Aritmetica dei puntatori

In C sono possibili le operazioni aritmetiche sui puntatori, ad esempio nel caso in cui dovessimo incrementare il nostro pointer di `+ 1` l'indirizzo a cui punterebbe sarà alla prossima cella di memoria seguente. La quantità `int` risultante aggiunta sarà in relazione al data type del puntatore a cui stiamo sommando. Di conseguenza il puntatore sarà incrementato della dimensione del data type a cui punta, valore che può essere recuperato tramite la funzione di libreria `sizeof(type_pointer)`.

```
newAddress = *point + (i * sizeof(int))
```

> `i` indica il numero di incremento del pointer. Si può anche scrivere solo `*(point + 1)` per accedere nella cella dopo di memoria, in questo modo possiamo incrementare di x posizioni.

Scorrimento array tramite puntatori:

```c
#include <stdio.h>

#define LIMIT 10

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

    int addresses[LIMIT] = {23, 324, 12, 76, 321, 43, 45, 98, 109, 34}, indexLoop;

    for(indexLoop = 0; indexLoop < LIMIT; indexLoop++) {

        printf("Contenuto con indice %d\t Con puntatore %d\n", addresses[indexLoop], *(addresses + indexLoop))
    }

    return 0;
}
```

Files

Per utilizzare i file nel linguaggio C è necessario dichiarare un puntatore al suo descrittore, aprire e poi chiudere lo stream. Nell'apertura riceve una stringa che contiene in nome del file da aprire

```c
/* dichiarazione */
FILE *someText

/* apertura in scrittura */
someText = fopen("myFile.txt", "w");

/* scrittura con controllo */
if(someText) {

    fprintf(someText, "my cool and awesome line");
}

/* chiusura */
fclose(someText);
```

Utilizzando `fscanf()` che legge da file la sintassi è la stessa dello `scanf()` classico a parte l'aggiunta del puntatore al primo parametro. Se non ci sono più valori da leggere nel file il sottoprogramma restituisce `EOF` (cioè -1). Possiamo utilizzare come controller di ciclo la funzione `feof(someText)` (`someText` o qualsiasi altro puntatore a file) per determinare la fine del file; il sottoprogramma restituisce 1 se abbiamo raggiunto la fine del file, altrimenti zero. NB: possiamo utilizzare come espressione di controllo `!feof(someText)` o anche sfruttando il valore di `fscanf()`: `fprintf(someText, "my cool and awesome line") > 0`.

Esempio di backup di creazione di un file per fare il backup di un file esistente:

```c
#include <stdio.h>
#define STR_SIZE 80

void backup(void) {

    char single_c, in_name[STR_SIZE], out_name[STR_SIZE];
    FILE *in, *out;

    printf("Enter name of file you want to backup >> ");

    /* si acquisisce il nome del file da copiare e aprirlo in lettura */
    for( scanf("%s", in_name); (in = fopen(in_name, "r")) == NULL; scanf("%s", in_name) ) {

        printf("Cannot open file %s, for input\nRe-enter file name >> ", in_name);
    }
```

```
    printf("Enter name for the backup copy >> ");

    /* si acquisisce il nome del file su cui fare il backup */
    for( scanf("%s", out_name); (out = fopen(out_name, "w")) == NULL; scanf("%s", out_name) ) {

        printf("Cannot open file %s, for output\nRe-enter file name >> ", out_name);
    }

    /* copia un carattere alla volta */
    for( single_c = getc(in); single_c != EOF; single_c = getc(in) ) {

        putc(single_c, out);
    }

    fclose(in);
    fclose(out);

    printf("Copied %s to %s.\n", in_name, out_name);

}

int main() {

    backup();

    return (0);
}
```

Binary files write and read

I binary files occupano meno spazio dei `*.txt` e sono più veloci. La sintassi per scrivere e leggere un binary file si differenziano poichè con i binary usiamo i sottoprogrammi `fwrite()` e `fread()`.

```
int arr[10] = {...};
FILE *binFile;

binFile = fopen("myBinFile.bin", "wb");

if(binFile) {

    /* il terzo parametro è quante volte devo scrivere l'oggetto */
    fwrite(arr, sizeof(int), 1, binFile);
}

fclose(binFile)
```

`fread()` ha la stessa sintassi con qualche differenza ovvero la stringa di lettura invece che di scrittura per esempio. Di seguito un esercizio di esempio di utilizzo dei binary files.

```
#include <stdio.h>

#define LIMIT 500

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

    int arrBin[LIMIT], readFrom[LIMIT], indexLoop;
    FILE *binWrite, *binRead;

    for(indexLoop = 0; indexLoop < LIMIT; indexLoop++) {

        arrBin[indexLoop] = indexLoop * 2;
    }
```

```
    binWrite = fopen("valuesWrite.bin", "wb");

    if(binWrite) {

        fwrite(arrBin, sizeof(arrBin), 1, binWrite);
    }

    fclose(binWrite);

    binRead = fopen("valuesWrite.bin", "rb");

    if(binRead) {

        fread(readFrom, sizeof(readFrom), 1, binRead);
    }

    for(indexLoop = 0; indexLoop < LIMIT; indexLoop++) {

        printf("[%d] ", readFrom[indexLoop]);
    }

    return 0;
}
```

Tabella argomenti `fopen()`:

File access mode string	Meaning	Explanation	Action if file already exists	Action if file does not exist
`"r"`	read	Open a file for reading	read from start	failure to open
`"w"`	write	Create a file for writing	destroy contents	create new
`"a"`	append	Append to a file	write to end	create new
`"r+"`	read extended	Open a file for read/write	read from start	error
`"w+"`	write extended	Create a file for read/write	destroy contents	create new
`"a+"`	append extended	Open a file for read/write	write to end	create new

Allocazione dinamica stack e heap

In C è possibile allocare dinamicamente uno spazio di memoria in un area denominata heap tramite il sottoprogramma `malloc()` contenuto nella libreria `stdlib.h`: `tipo *p = malloc (size)` (controllare p con un `if` che non sia `NULL`) ed è buona prassi castare nel tipo che ci interessa la chiamata a `malloc()` dato che è un puntatore a `void` in questo modo: `(int*) malloc(size)`; `size` è la quantità di memoria da allocare ed è in genere espressa come `n*sizeof(tipo)`. Nella stessa libreria per deallocare dello spazio di memoria non più necessario si utilizza `free(indirizzo)`. La `calloc()` prende come parametri il numero di elementi e la dimensione di un singolo elemento. La principale differenza fra `malloc()` e `calloc()` è che la seconda quando riserva la memoria nello heap va ad azzerare ogni cella prima di restituire il puntatore. La chiamata a `realloc()` prende come parametri l'array a cui vogliamo cambiare la dimensione e la nuova dimensione totale in byte ovvero sempre `n*sizeof(tipo)`.

```
#include <stdlib.h>

int numElem = 20;

/* alloco spazio e casto nel tipo che mi interessa */
int *custom = (int*) malloc(numElem * sizeof(int));

/*
```

```
 * alloco spazio e azzero le celle. primo parametro numero
 * elementi e secondo parametro size del singolo tipo
 */
int *custom = (int*) calloc(numElem, sizeof(int));

/* modifico la dimensione di custom */
int *custom = (int*) realloc(custom, 12 * sizeof(int));

/* lo usiamo un po' come fclose() */
free(custom);
```

Programmazione in grande

Un modo per riutilizzare il codice è organizzarlo in delle librerie personali attraverso i file di intestazione che solitamente includiamo nel codice per utilizzare dei sottoprogrammi di libreria come ad esempio `ctype.h`. Le caratteristiche principali che deve avere un file di intestazione sono dei blocchi che commentano la funzionalità della libreria e che spieghino la finalità e funzionalità dei vari sottoprogrammi, delle direttive `#define` per definire le macro costanti cosicchè possano essere utilizzate dal programma e prima delle definizioni dei prototipi delle funzioni si utilizza la convenzione `extern` e ciò indica al compilatore che la definizione del sottoprogramma in quel file di intestazione verrà fornita al linker.

```
extern int n_arrcpy(int *dest, const int *src);
```

Nella direttiva al preprocessore `#include` abbiamo finora sempre usato le parentesi angolari (`< >`) per includere i file di intestazione contenuti nella libreria standard di C. Per indicare al preprocessore un file di intestazione custom si utilizzano le virgolette per indicare che la libreria è stata creata dal programmatore: `#include "customlib.h"`. Nelle macro costanti inoltre è importante precedere oltre al nome stesso anche il nome della libreria o comunque differenziare da macro costanti di altre librerie che possono avere un nome simile se non uguale così da evitare dei conflitti. Esempio:

```
/* arrUtils.h */
#define ARR_UTILS_MAX_SIZE 40
```

Variabili globali

Il linguaggio C permette la dichiarazione di variabili globali le quali vengono dichiarate nello stesso modo di una variabile classica con l'unica differenza che sono nel livello più alto al di fuori di ogni sottoprogramma cosicchè la visibilità di esse comprenda tutto il codice sorgente. Nonostante spesso sia inevitabile l'utilizzo di variabili globali, un accesso ad esse senza restrizioni è generalmente sconsigliato poichè considerato dannoso e va contro le prerogativa del codice e dei suoi sottoprogrammi che devono avere accesso solamente ai dati che hanno necessità di conoscere, seguendo l'interfaccia documentata e rappresentata dal prototipo del sottoprogramma. Una soluzione è quella di rendere le variabili delle costanti

```
#include <...>

extern const int months = 12; /* per essere utilizzata da altri sorgenti */

int main(int argc, char* argv[]) { ... }
```

Classi di memorizzazione static e register

```
int funFunction(int *tempValue) {

    static int once = 0;
    int many = 0;
}
```

> Per ogni invocazione di `funFunction()` la variabile `many` viene inizializzata a zero ed ogni volta che `funFunction()` termina, `many` viene rilasciata. Al contrario, la variabile `static once` viene inizializzata e rilasciata una sola volta. Se `funFunction()` modifica il valore di `once`, tale valore modificato viene mantenuto nelle chiamate a seguire su `funFunction()`. NB: Utilizzare una variabile locale `static` per poi richiamare sulla stessa funzione dov'è contenuta è una pratica di programmazione sconsigliata poichè complica notevolmente la comprensibilità del codice.

Per quanto riguarda classi di memorizzazione `register` è strettamente legata alla classe di memorizzazione `auto` e può essere applicata solo a parametri e variabili locali. Una variabile `register` avverte solo al compilatore che si accederà molto frequentemente a tale variabile e quindi sarebbe opportuno che venisse allocato uno spazio (caratterizzato da tempi di accesso

molto piccoli) all'interno dell'unità centrale di elaborazione. Dei buoni candidati possono essere ad esempio delle variabili che scorrono array di grandi dimensioni;

```
static double manyValues[69][420];
register int row, col;
```

Compilazione condizionale

Un utilizzo delle direttive condizionali sono per tenere traccia dei comportamenti dei sottoprogrammi e per testing, molto utile così nel momento in cui non servono più basta cambiare il parametro di una macro che determina l'esecuzione di procedimenti di testing. Di seguito un semplice esempio di programma ricorsivo che traccia i valori di ogni chiamata ricorsiva:

```
/*
    (TRACE) non necessariamente in questo sorgente, può anche non esserci ma essere
    definito da una libreria che abbiamo incluso con una direttiva al preprocessore
 */
#define TRACE 1

int ricMultiply(int m, int n) {

    int res;

    #ifdef TRACE
        printf(": Entering >> m = %d, n = %d\n", m, n);
    #endif

    /* caso base */
    if(n == 1) {

        res = m;

    } else {

        /* passo ricorsivo */
        trace++;
        res = m + ricMultiply(m, n - 1);

        #ifdef TRACE
            printf(": Leaving >> m = %d, n = %d, res = %d\n", m, n, res);
        #endif
    }

    return(res);
}
```

> Altre direttive sono: `#ifndef` (contrario di `#ifdef`), `#elif, #else, #undef` (elimina la definizione di un particolare nome).

Passare parametri al programma principale

Digitando l'eseguibile da terminale, binary che creiamo con il comando per chiamare il compilatore `gcc -o somePgrm somePgrm.c`, possiamo dargli dei parametri che vengono accettati dai parametri formali del nostro programma principale ovvero `int main(int argc, char* argv[])`:

```
>> somePgrm old.txt new.txt
```

Ciò significa che `argv` avrà riempito i primi due elementi nel suo array.

```
argv[0] = "old.txt"
argv[1] = "new.txt"
```

> Possiamo quindi con questi parametri usarli nel nostro codice accedendo agli elementi di `argv[]`.

Definire macro costanti con parametri

```
#define NOME_MACRO(parametro1, parametro2) /* corpo della macro */
```

NB: prima delle parentesi non ci deve essere uno spazione poichè in quel caso avverrebbe una assegnazione ed la componente assegnata diverrebbe tutto quello che vi sta dopo lo spazio. Di seguito un esempio completo.

```
#include <stdio.h>

#define LABEL_PRINT_INT(label, num) printf("%s = %d", (label), (num))

int main(void) {

    int r = 5, t = 12;

    LABEL_PRINT_INT("rabbit", r);
    printf(" ");
    LABEL_PRINT_INT("tiger", t + 2);

    return (0);
}
```

```
(output)
>> rabbit = 5 tiger = 14
```

Il passaggio che fa quando si riferisce alla macro si chiama espansione della macro.

Per scrivere la definizione di una macro su più righe si mettono dei backslash \ a fine riga tranne l'ultima.