

Laboratorio di linguaggio C

Massimiliano De Pierro

massimiliano.depierro@unito.it

Prima di iniziare con la programmazione

- Il corso introdurrà alla programmazione in linguaggio C in **ambiente UNIX**. Considerando i tre sistemi operativi più diffusi in ambito educativo:



- Linux e OSX Apple sono sistemi operativi UNIX-like quindi adatti per essere utilizzati in questo corso, richiedono comunque entrambi l'installazione dell'**ambiente di sviluppo in C**;
- Microsoft Windows non è UNIX-like, esistono però delle librerie di adattamento che consentono il *porting* di un certo numero di applicazioni e strumenti Linux. Coloro che desiderano seguire gli argomenti trattati e rimanere in ambiente Windows possono provare le librerie fornite dai progetti **minGW** e **cygwin**. Esiste anche la possibilità di usare il "Sottosistema di Windows per Linux" (WSL): <https://docs.microsoft.com/it-it/windows/wsl/>

Gli strumenti – L'EDITOR

- Un editor testuale *semplice*, meglio se con qualche supporto alla scrittura di programmi per elaboratori (indentazione automatica, colorazione della sintassi, bilanciamento parentesi, ...).



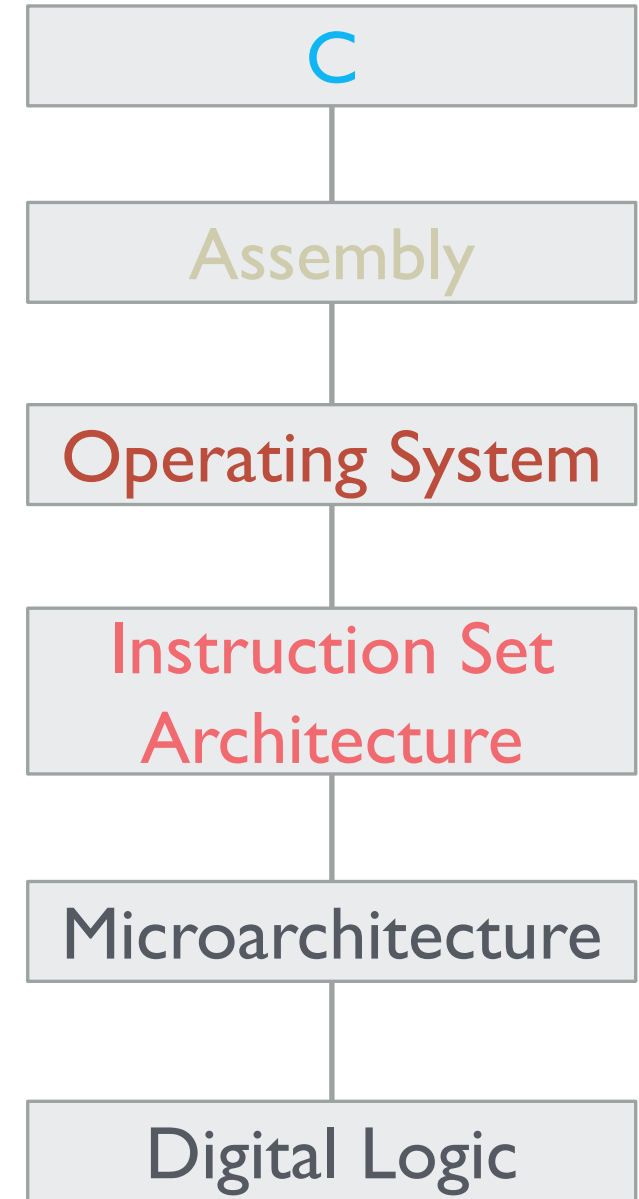
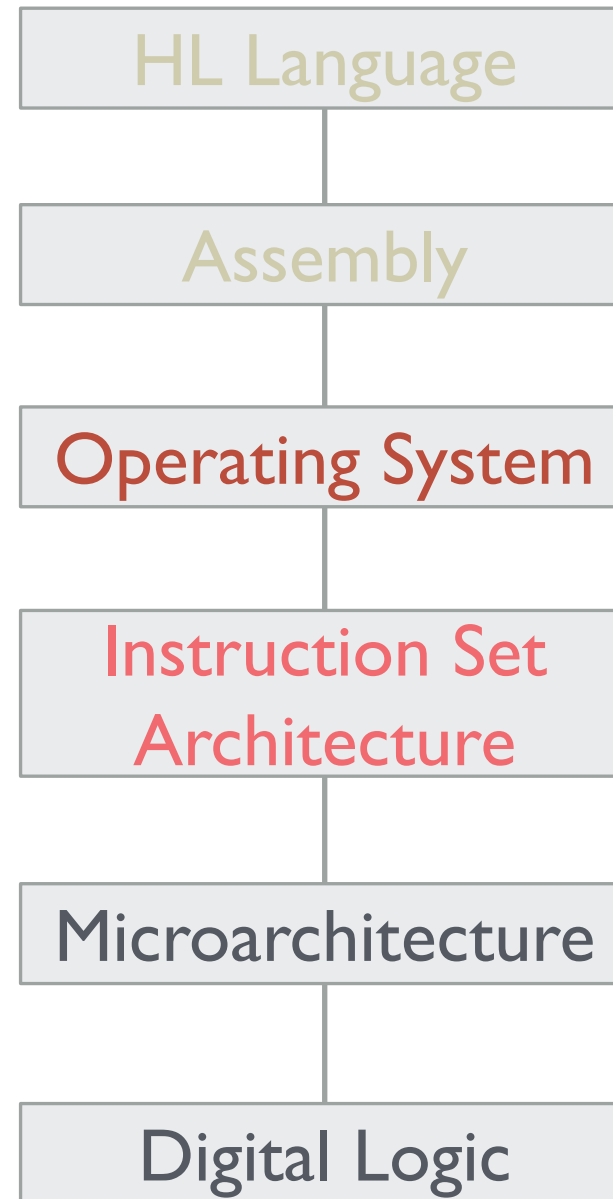
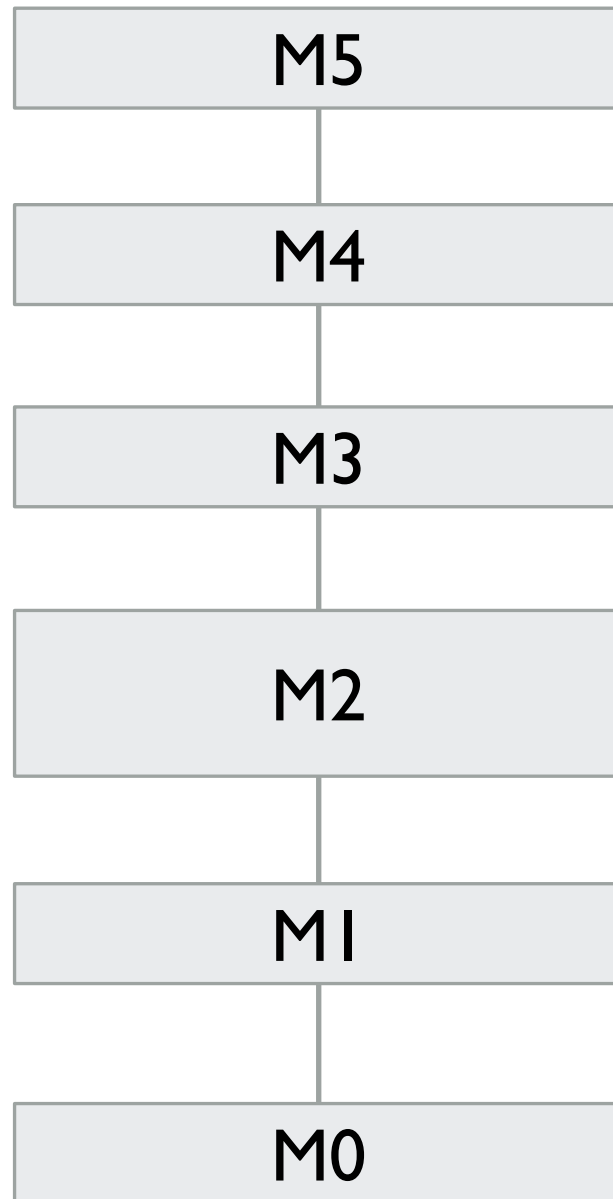
- vim
- kate
- gedit
- ...



- notepad++

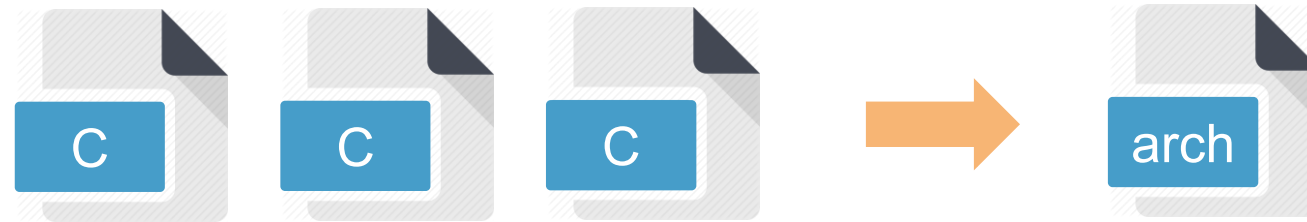


- vim
- Xcode
- ...



Prima di iniziare con la programmazione

- Nella "Organizzazione Strutturata del Sistema Computer" (vedere Tanenbaum) la **macchina C** è collocata ai livelli più alti del diagramma astrati:



- Diremo che la **macchina C** interpreta ed esegue programmi scritti in **linguaggio C**;
- In realtà l'esecuzione dei programmi C è ottenuta attraverso un processo di compilazione.

Prima di iniziare con la programmazione

- Il **C** è un linguaggio non comprensibile all'architettura hardware del processore: questa è in grado di eseguire solo programmi informatici scritti in **linguaggio macchina**;
- Un programma in C dovrà essere quindi tradotto in un programma equivalente in linguaggio macchina specifico di una **architettura target di processore** (esempio Intel x86), solo quest'ultimo potrà essere eseguito.
- Il processo di traduzione si chiama **compilazione**.



Il processo completo di produzione di un file eseguibile

- La schematizzazione illustrata nella slide precedente è una semplificazione, in realtà il processo di compilazione si compone di diversi passaggi e non avviene in maniera diretta;
- I passi in ordine sono (tra parentesi è indicato lo strumento relativo):
 1. Pre-processing dei file C (**preprocessore-C**);
 2. Traduzione o compilazione dei file sorgenti C pre-processati in sorgenti in linguaggio assembly (**compilatore-C**);
 3. Traduzione dal linguaggio assembly al linguaggio macchina (**assembler**), i file prodotti contengono *codice oggetto*;
 4. Linking dei file (**linker**).

ARCHITECTURE SPECIFIC

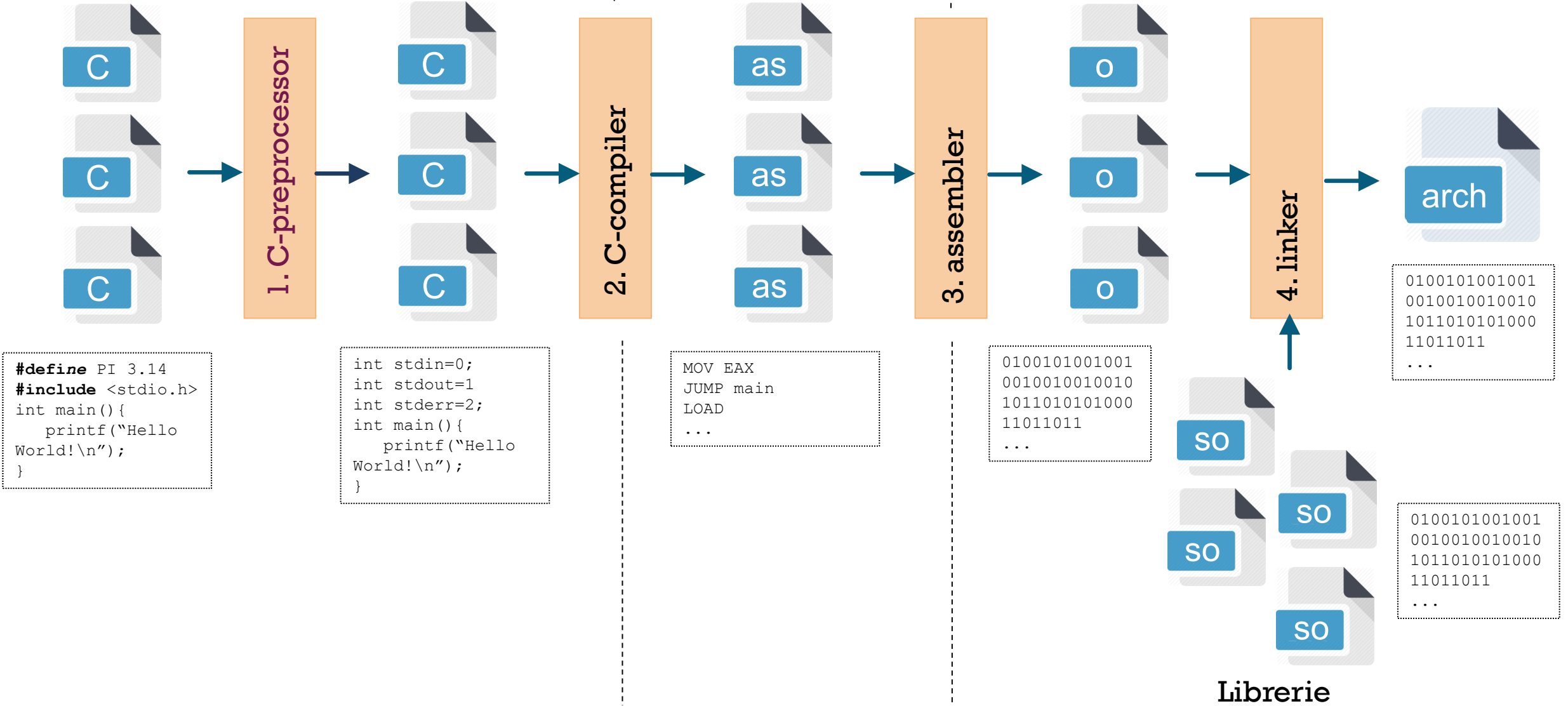
File sorgenti
C

Sorgenti C dopo
il preprocessing

File in linguaggio
assemblativo

File oggetto

File eseguibile



Gli strumenti – IL PROGRAMMA GCC

Gnu Compiler Collection (gcc.gnu.org)

- **gcc** non è un compilatore C, è uno strumento *front-end* usato in tutto il processo di produzione di un programma eseguibile a partire da sorgenti scritti in linguaggi di programmazione di alto livello, secondo lo schema illustrato nella slide precedente;
- Noi lo useremo per produrre un eseguibile di un progetto scritto in C;
- **gcc** si utilizza da linea di comando. Tutti i SO forniscono una shell di comandi (sh, bash, csh, ...). Il comando **gcc** è molto complesso: per averne una idea consultare il manuale con `'man gcc'`.
- In generale **gcc** prende in input dei file e produce in output altri file. Attraverso le **opzioni** il programmatore specifica cosa deve fare. Inoltre oltre alle opzioni **gcc** usa le estensioni dei nomi dei file forniti sulla linea di comando per capire come operare.
Esempi di compilazione:

```
gcc calcolatrice.c funzioni.c operazioni.c -o calcolatrice -lm
```



```
gcc -c calcolatrice.c
```

```
gcc -c funzioni.c
```

```
gcc -c operazioni.c
```

```
gcc calcolatrice.o funzioni.o operazioni.o -o calcolatrice -lm
```

Gli strumenti – IL PROGRAMMA GCC

Gnu Compiler Collection (gcc.gnu.org)

- **compilazione separata dei sorgenti C e linking finale:**

```
gcc -c calcolatrice.c
gcc -c funzioni.c
gcc -c operazioni.c
gcc calcolatrice.o funzioni.o operazioni.o -o calcolatrice -lm
```

```
$ edit funzioni.c
```

```
...
```

```
... modifico il sorgente C del file funzioni.c
```

```
...
```

```
gcc -c funzioni.c
gcc calcolatrice.o funzioni.o operazioni.o -o calcolatrice -lm
```

oppure

```
gcc calcolatrice.o funzioni.c operazioni.o -o calcolatrice -lm
```

Gli strumenti – IL PROGRAMMA GCC

Gnu Compiler Collection (gcc.gnu.org)

- i flag `-c` `-S` `-E` servono a fermare il processo di traduzione in una delle fasi intermedie:
- `-E` ferma la traduzione subito dopo il pre-processing.
- `-c` compile only - ferma la traduzione subito dopo l'assemblaggio;
- `-S` do not assemble - ferma la traduzione subito dopo la compilazione;

```
gcc -S calcolatrice.c
```

- l'output è scritto in un file con suffisso `.s`

```
gcc -E calcolatrice.c
```

- l'output è scritto sullo standard output.

Il C – Caratteristiche di base

- Sviluppato da Dennis Ritchie, il C nasce fra il 1969 e il 1973 come linguaggio per lo sviluppo del sistema operativo UNIX, standardizzato (ANSI C) nei primi anni '80
- Linguaggio imperativo, procedurale (Pascal, Algol60, ...)
- Linguaggio compilato, ovvero, il codice sorgente viene tradotto interamente da un compilatore in codice eseguibile. L'esecuzione può avvenire solo dopo che la compilazione è terminata senza errori

Il C – Caratteristiche di base

Un programma in sorgente C è composto generalmente da due tipi diversi di file:

- file di intestazione (**header file**): questi non contengono istruzioni C ma **dichiarazioni di dati** e **definizioni di tipo**. Il nome di questi file deve avere estensione **h** ad esempio:

`calcolatrice.h`

- file sorgente C: questi contengono principalmente istruzioni C organizzate in funzioni nonché **definizioni di dati**. Il nome di questi file deve avere estensione **c**, ad esempio:

`calcolatrice.c`

Il C – Caratteristiche di base

- **Struttura di un file sorgente C:**

inclusione di file e
direttive al preprocessore

definizione di tipi

dichiarazioni di
variabili globali

definizioni di
funzioni

funzione main()

Il C – Primo programma in C

```
#include <stdio.h>

int main(){
    printf("Hello World!\n");
}
```

- Il carattere **#** nel C introduce una **direttiva** per il **pre-processor C**, nel codice dell'esempio si tratta della direttiva **include**, ossia di *inclusione di file*; il file il cui nome segue il nome della direttiva è, in questo esempio, tra parentesi angolari ed è **stdio.h**; la direttiva stabilisce che nel punto nella quale è specificata deve essere inserito il testo contenuto nel file **stdio.h**;
- Tecnicamente il **pre-processor C** sostituisce alla direttiva il contenuto del file **stdio.h** (vedremo in seguito la funzione di questa operazione);
- Esistono altre direttive per il pre-processor. In questo corso vedremo oltre alla **include** le direttive **define** e **ifdef**.

Il C – La funzione main()

```
#include <stdio.h>

int main(){
    printf("Hello World!\n");
}
```

- Il punto di ingresso, detto **entry-point**, di un programma C è la funzione `main()`. Le funzioni C sono simili ai metodi statici studiati in Java, con la eccezione che esse non appartengono ad alcuna classe perché in C queste non esistono;
- Ogni programma C deve avere esattamente una sola definizione di funzione `main()` (diversamente da Java dove il programmatore può definire più funzioni `main()` appartenenti a classi diverse e indicare quale di esse costituirà l'entry-point al momento dell'esecuzione).
- Le funzioni C a conclusione della loro esecuzione restituiscono un **valore**; ogni valore in C è di un determinato **tipo** primitivo; il C è un linguaggio tipato. La funzione `main()` restituisce un valore di tipo **int**.
- Tra parentesi graffe segue la **definizione** della funzione; la porzione di codice tra le parentesi graffe è anche detto *testo* della funzione.

Il C – Le funzioni

```
#include <stdio.h>
#include "string.h" // contiene la definizione del tipo
                   // string (user defined)

void println(string str)
{
    printf("%s\n",str);
}

int main()
{
    println("Hello World!");
    println("Today's temperature is 30C.");
    println("Bye Bye!");

    return 0;
}
```

Il C – Le funzioni

```
#include <stdio.h>
#include "string.h" // contiene la definizione del tipo
                  // string (user defined)
```

```
void println(string str)
{
    printf("%s\n",str);
}
```

testo della funzione println

```
int main()
{
    println("Hello World!");
    println("Today's temperature is 30C.");
    println("Bye Bye!");

    return 0;
}
```

testo della funzione main

Il C – I tipi primitivi

```
#include <stdio.h>

int main(){
    printf("Hello World!\n");
}
```

- I tipi primitivi in C sono i seguenti e sono tutti tipi numerici:

int, short int, long int

unsigned int, unsigned short int, unsigned long int

float, double

char

Il C – I tipi primitivi

- Il tipo `void` è usato per dichiarare una funzione che non restituisce alcun valore alla terminazione, ad esempio:

```
void clear();
```

- Esiste infine il tipo puntatore `*` a *dato* che si accompagna al nome di un tipo primitivo. I valori assunti da una variabile di tipo puntatore sono [indirizzi di memoria](#). Ad esempio:

```
int *, void *, double *, ...
```



- Il puntatore a valori di tipo void (`void *`) indica un puntatore ad un dato generico.
- Il tipo puntatore è primitivo, posso anche definire variabili che sono puntatori a puntatori:

```
int **, void ***
```

Il C – I tipi primitivi

- Nella definizione di D. Ritchie, il C, con eccezione per il tipo `char`, non stabilisce alcuna dimensione dei dati in base al loro tipo; la dimensione è *machine-dependent*, cioè dipendente dall'architettura per la quale è generato il testo eseguibile del programma.
- Se il programmatore ha la necessità di scrivere un `programma C portabile` non deve produrre un codice C dipendente dalla dimensione dei dati, ipotizzando delle dimensioni per i tipi primitivi;
- Il tipo `char` ha dimensione 1 byte.

Il C – Definizione di variabili

```
int    i;   
short j = 2;      // definizione con inizializzazione  
int    *ptr = NULL; // definizione di un puntatore con  
                        inizializzazione  
int    k = i+j%2;  // k == ?
```

NOTE: `NULL` è una costante simbolica (un valore specificato usando un simbolo mnemonico). La costante `NULL` indica il valore di puntatore non valido. Non fa parte del C, le costanti simboliche sono definite dal programmatore.

- Lo statement C della **definizione**:

`tipo nome [= expr];`

- `expr` può essere una qualsiasi espressione C. Una espressione ha un valore ed ogni valore è di un determinato tipo primitivo. Nella inizializzazione di una variabile il tipo di `expr` deve accordarsi con il tipo della variabile --- **type matching**.

Il C – Dichiarazione di variabili

- Nell'assegnamento di un valore ad una variabile, il compilatore C ammette (quando non c'è perdita di precisione) alcune **conversioni implicite di tipo** al fine di garantire il matching. Se c'è perdita di precisione la conversione è consentita, potremmo però avere degli avvertimenti (*warnings*) durante la compilazione. Se i tipi sono incompatibili il compilatore termina con un errore di **type mismatch**.

```
int i=9;
```

```
double j = 2.0*i/2;    // j == ?  
double j = 2.0*(i/2);  // j == ?  
double j = 2.0*(i/2.0); // j == ?
```

NOTE: le espressioni sono valutate da sinistra a destra rispettando la priorità e l'associatività degli operatori e l'ordine imposto dalle parentesi.

Il C – Dichiarazione di variabili

- Scelta migliore:

```
double j = 2*i/2.0;    // Perché?
```

la prima operazione, la moltiplicazione, è tra interi mentre il risultato finale è tra double.

Il C – Definizione di variabili

```
int    i;  
short j = 2;      // definizione con inizializzazione  
int    *ptr = NULL; // definizione di un puntatore con  
                        inizializzazione  
int    k = i+j%2;  // k == ?
```

- In C una variabile non inizializzata ha un valore **indefinito**:
- Mentre in Java il compilatore genera un errore quando riscontra l'uso di una variabile indefinita, in C questo non avviene. Il compilatore potrebbe generare un **warning**.
- Quindi nella dichiarazione di k la macchina C assegna un valore alla variabile, qual è questo valore dipende dal runtime. Nella run-time del programma, k ha un valore indefinito perché usa la variabile i non inizializzata.

Il C – Le variabili e le regole di visibilità

```
{ // Blocco A
  int i;
  int k=3;
  ....
  { // Blocco A.1
    int i;
    int j;
    i+=1; // è la i definita in A.1
    ....
    k+=1;
  }
  i+=2; // è la i definita in A
  j=3;  // errore, variabile non definita
}

{ // Blocco B
  int i;
  ....
}
```

- Una variabile può essere usata solo dopo la sua **dichiarazione**.
- Una porzione di codice racchiusa tra una coppia di graffe bilanciate è chiamata *blocco*. In ogni blocco il codice andrebbe rientrato con un carattere di tabulazione aggiuntivo;
- I blocchi possono essere annidati gli uni dentro gli altri;
- Nel C ANSI89 le dichiarazioni delle variabili devono essere raggruppate al principio di ogni blocco;
- La variabile *i* definita in A è diversa dalla variabile *i* definita in B, inoltre è visibile solo dalla sua definizione fino al termine del blocco;
- La variabile *i* in A.1 **nasconde** la variabile *i* in A.

Il C – Le variabili puntatore

- Una variabile è caratterizzata da una quadrupla:

(nome, tipo, **indirizzo**, valore)

- **indirizzo** identifica la locazione di memoria alla quale il valore della variabile è memorizzato.
- Il valore per l'indirizzo di una variabile è assegnato dal Sistema Operativo durante il runtime o al tempo di compilazione del programma C in programma macchina. Non è il programmatore che lo può specificare.

```
#include <stdio.h>

int i = 10;
int *ptr = &i; // & è l'operatore "indirizzo di"

int main(){
    printf("La variabile i ha valore %d\n", i);
    printf("L'indirizzo in memoria di i è %p\n", ptr);
}
```

Il C – Le variabili puntatore

```
#include <stdio.h>

int i=10;
int *ptr=&i;

int main(){
    printf("L'indirizzo in memoria di i è\t%p\n", ptr);
    printf("L'indirizzo in memoria di ptr è\t%p\n", &ptr);
}
```

```
depierro@Surface ~
$ gcc stampa_puntatore.c -o stampa_puntatore

depierro@Surface ~
$ ./stampa_puntatore
L'indirizzo in memoria di i è 0x100402010
L'indirizzo in memoria di ptr è 0x100402018
```

Il C – Le variabili puntatore

```
#include <stdio.h>

int i=10;
int *ptr=&i;

int main(){
    printf("L'indirizzo in memoria di i è\t%p\n", ptr);
    printf("L'indirizzo in memoria di ptr è\t%p\n", &ptr);
}
```

Nella testo sopra, qual è il tipo della espressione &ptr? In altre parole come completereste nei puntini la seguente definizione C?

```
..... a = &ptr;
```

seguire il link per accedere al quiz: [Quiz](#)

Il C – Le variabili puntatore

```
int i=10;
```

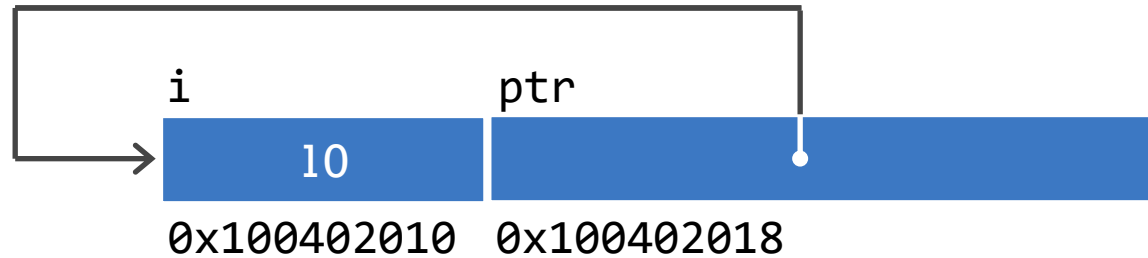
```
int *ptr=&i;
```

i	ptr
10	0x100402010
0x100402010	0x100402018

: nomi delle variabili nel programma C

: contenuto della memoria

: nomi delle variabili a runtime



Il C – Le variabili puntatore

- Il valore simbolico **NULL** è usato per assegnare un valore non valido ad una variabile puntatore:

```
#include <stddef.h>

int *ptr = NULL;
```

- Se una variabile puntatore non è inizializzata essa ha un valore indefinito;
- L'uso di un puntatore con valore arbitrario nella migliore delle ipotesi genera durante l'esecuzione del programma un accesso ad una zona di memoria illegale causandone la terminazione da parte del Sistema Operativo con un segnale che si chiama Segmentation Fault (SEGV), nella peggiore delle ipotesi causa l'accesso a una zona dei dati dello stesso programma andando quindi a modificare in maniera subdola altre variabili;
- è buona norma inizializzare sempre un puntatore a **NULL**, in questa maniera l'uso errato di un puntatore **NULL** durante l'esecuzione causa la terminazione del programma da parte del Sistema Operativo con il segnale SEGV.

Il C – Gli operatori del C

■ Aritmetici:

+ - * / %

■ Relazionali:

< <= > >= != ==

■ Logici:

|| && !

■ Altri:



*	dereferenzamento	++	--			
&	indirizzo	+=	-=	\=	*=	
>>	shift right		&	~		OR AND NOT bitwise
<<	shift left					

Il C – Le strutture di controllo del flusso

- Strutture di iterazione:



```
while ( expr ){  
    ....  
}
```

```
do{  
    ....  
}while ( expr );
```

```
for ( expr ; expr ; expr ) {  
    ....  
}
```

- `expr` determina le condizioni di terminazione dell'iterazione.

Il C – Le strutture di controllo del flusso

- `expr` scritta nel campo della *condizione* di una struttura di controllo di flusso può essere una qualsiasi espressione C (aritmetica, relazionale, logica, di assegnamento, chiamata di funzione, ...);
- le espressioni che compaiono nel campo *condizione* di una struttura di controllo sono prima valutate poi il loro valore è usato come *valore di verità*: se il valore di `expr` è diverso da 0 la condizione ha valore logico `true` altrimenti `false`;
- in C **non esiste** il tipo di dato `boolean`.

```
while ( 1 ){ // iterazione senza fine
    ....
}

while ( a ){ // itera sulle istruzioni del blocco al perdurare della
            // condizione: 'a' ha valore diverso da 0
    ....
}

while ( a=n ){ // itera sulle istruzioni del blocco al perdurare della
              //condizione: 'a' ha valore diverso da 0
    ....
}
```

Il C – Le strutture di controllo del flusso

```
while ( a==n ){ // itera sulle istruzioni del blocco al perdurare della
               //condizione: 'a' ha valore uguale a 'n'
    ....
}
```

```
for ( i=0 ; no_registered() ; i++ ){
    ....
    ....
}
```

Il C – Le strutture di controllo del flusso

- Diramazione condizionata:

```
if ( expr ){  
    ....  
}
```

```
if ( expr ){  
    ....  
}else{  
    ....  
}
```

```
if ( expr ){  
    ....  
}else if ( expr ){  
    ....  
}else if ( expr ){  
    ....  
}else{  
    ....  
}
```

Il C – Le strutture di controllo del flusso

- Le espressioni sono valutate dal compilatore, o dalla macchina C, leggendole da sinistra a destra, rispettando l'associatività e la priorità degli operatori, e rispettando l'ordine imposto dalle parentesi tonde;
- Nella **valutazione di una espressione logica** il codice è tradotto in maniera tale da terminare la valutazione non appena il risultato intermedio ne garantisce la veridicità o la falsità:

```
if ( ptr!=NULL && *ptr==10 ){  
    // blocco A  
    ....  
}else{  
    // blocco B  
    ....  
}
```

Traduzione in assembly IJVM:

```
                                ILOAD  ptr  
                                IFEQ   null_pointer  
                                ILOAD  (ptr)  
                                BIPUSH 10  
                                IF_ICMPEQ ok  
null_pointer: // traduzione del  
              // blocco B  
              GOTO next  
ok:           // traduzione del  
              // blocco A  
next:        ....
```

Il C – Le strutture di controllo del flusso

- Diramazione condizionata a n-vie con switch:

```
switch (expr){  
    case val1: ...  
                ...  
                break;  
    case val2: ...  
                ...  
                break;  
    ....  
    case valn: ...  
                ...  
                break;  
    default: ...  
                ...  
                break;  
}
```

Il C – Le strutture di controllo del flusso

- Diramazione in-condizionata:

```
        goto label;  
        ....  
        ....  
        ....  
label:   ....  
        ....  
        ....  
        ....  
        ....
```

Il C – Le strutture di controllo del flusso

- gli statement break and continue:

```
while ( expr ){  
    if (expr)  
        break;  
}
```

```
for ( expr ; expr ; expr ) {  
    if ( expr )  
        continue;  
}
```

- nel caso di strutture di controllo iterative annidate gli statement fanno riferimento alla struttura più profonda nell'annidamento.

Il C – Le funzioni C

```
#include <stdio.h>

int main(){
    int      n = 47; // total number of students
    const int b = 8; // number of snacks per pack

    printf("I need %d packs of snacks for %d students.\n",
           ceiling(n,b), n);
}
```

- Oltre alla funzione `main()` il programmatore può definire altre funzioni. Una **definizione di funzione** è costituita in ordine da:
 - tipo del valore restituito;
 - nome della funzione
 - lista dei parametri formali (tipo nome) tra parentesi tonde
 - testo della funzione
- Esempio:

```
int ceiling(int n, int b){ // definizione di funzione
    // cond: n>=0, b>0
    return (n+b-1)/b;
}
```

Il C – Le funzioni C

```
#include <stdio.h>

int ceiling(int, int);    // dichiarazione di funzione

int main(){
    int      n = 47; // total number of students
    const int b = 8; // number of snacks per pack

    printf("I need %d packs of snacks for %d students.\n",
           ceiling(n,b), n);
}

int ceiling(int n, int b){ // definizione di funzione
    return (n+b-1)/b;
}
```

Il C – Le funzioni C

```
#include <stdio.h>

int ceiling(int n, int b){ // definizione di funzione
    // cond: b>0
    return (n+b-1)/b;
}

int main(){
    int n = 47; // total number of students
    const int b = 8; // number of snacks per pack

    printf("I need %d packs of snacks for %d students.\n",
           ceiling(n,b), n);
}
```

Il C – Esercizio puntatori a stringhe costanti

Esercizio A.1 Scrivere un programma che analizza carattere per carattere la stringa costante di testo `"Hello WoRLd! Today's temperature is 34C."`: se il carattere analizzato non è una lettera dell'alfabeto o è una lettera minuscola ne fa l'eco sullo standard output, se è una lettera dell'alfabeto in maiuscolo (uppercase) la converte in minuscolo (lowercase) e ne fa l'eco. L'output deve quindi essere: `"hello world! today's temperature is 34c."`.

La stringa, allocata nei dati costanti del programma, deve essere analizzata tramite una variabile puntatore. Usare solo argomenti trattati finora per risolvere l'esercizio.

Il C – Le stringhe costanti C

- Il C **rappresenta** le stringhe come sequenze di **char**. La macchina C quando legge una stringa alloca nella **memoria dei dati costanti** tanti **char** quanti sono i caratteri della stringa. Nella rappresentazione viene aggiunto un carattere in fondo ad indicare la fine della sequenza in memoria, questo è il carattere con codifica nulla: `'\0'`.

```
char *str = "Hello World"; // str punta ad una stringa costante
```

`char *str` →

'H'	'e'	'l'	'l'	'o'	' '	'W'	'o'	'r'	'l'	'd'	'\0'
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	------

<code>*str == 'H'</code>	tipo degli operandi char
<code>*(str+6) == 'W'</code>	tipo degli operandi char
<code>*str+6 == 'N'</code>	tipo degli operandi char

- Il valore assegnato ad una stringa è l'indirizzo del primo carattere
- Un qualsiasi array di **char** che rispetti questa rappresentazione è una stringa C.

Il C – Le stringhe costanti C

- **Quiz:** data la seguente definizione di variabile:

```
char c = *("Computer Structured Organization"+4);
```

- Cosa stampa la seguente istruzione?

```
printf("Book Title: %c\n", c);
```

Il C – La libreria standard del C

```
#include <stdio.h>

int main(){
    printf("Hello World!\n");
}
```

- Il programma che stampa **Hello World!** contiene la chiamata a una funzione di libreria.
- `printf()` non fa parte del linguaggio C ma è una funzione, scritta da altri sviluppatori, il cui **codice oggetto** è fornito, insieme a quello di altre funzioni, in raccolte che si chiamano **librerie**.
- `printf` non è un simbolo riconosciuto dalla macchina C, la quale si rifiuterà di procedere se non le forniamo alcune informazioni necessarie a gestirlo, per poter proseguire ha bisogno di sapere la natura del simbolo (è una funzione), quanti e di che tipo sono i parametri formali e qual è il tipo del valore restituito.
- la dichiarazione della funzione `printf()` è data nel file di intestazione **stdio.h**

Il C – Input e Output formattato

- In C esistono alcune funzioni della libreria standard adibite all'input e all'output formattato, rispettivamente sono:

```
printf(char *format, ...) // prints on standard output stdout  
scanf(char *format, ...) // reads from standard input stdin
```

- entrambe prendono un numero variabile di parametri;
- il primo parametro è detto **stringa di formato**. Nel caso della printf costituisce nell'uso più semplice il testo da stampare. Ad esempio:

```
printf("Hello World!")
```


Il C – Input e Output formattato printf()

- Alcuni caratteri particolari sono indicati nella stringa con **sequenze di escape**. Queste sono:

<code>\n</code>	line feed	<code>\r</code>	carriage return	<code>\t</code>	tabulazione	<code>\f</code>	form feed
<code>\b</code>	backspace	<code>\\</code>	carattere \	<code>\"</code>	carattere "	<code>%%</code>	carattere %

- I sistemi operativi seguono codifiche diverse per il fine-linea:

Windows:	<code>\r\n</code>	UNIX:	<code>\n</code>	Macintosh:	<code>\r</code>
----------	-------------------	-------	-----------------	------------	-----------------

- La stringa può contenere dei **caratteri di controllo** %, questi non sono stampati ma indicano delle posizioni dove stampare, secondo l'ordine di lettura, i valori di espressioni fornite nei parametri rimanenti della chiamata a funzione. Dopo il carattere di controllo % seguono uno o più caratteri che identificano **il formato** secondo il quale stampare il valore. Esempio **d** indica il formato decimale:

```
int n = 234; // total number of students
printf("In aula ci sono %d studenti.\n", n);
```

Il C – Input e Output formattato printf()

Formati di conversione:

- `%d %i` per gli interi
- `%u` per gli interi unsigned
- `%ld` per gli interi long (long decimal)
- `%c` per i caratteri
- `%s` per le stringhe
- `%f %lf` per i float e i double
- `%e %E` notazione esponenziale
- `%g %G` notazione migliore tra `%f` e `%e` (rispettivamente `%f` e `%E`)
- `%p` per valori puntatore
- I caratteri di formato possono essere preceduti da due numeri separati da un punto, `n.m`. Essi indicano quante cifre usare rispettivamente per la parte intera e decimale di un numero.
- Ad esempio:

```
double r = rand()/((double)RAND_MAX); // random number between [0,1]
printf("Random number is: %.3lf\n", r);
```

Il C – Input e Output formattato – scanf()

- La funzione scanf() legge dallo stream di **standard input** (stdin):

```
#include <stdio.h>
int scanf(const char *format, ...);
```

- scanf() scansiona l'input secondo il formato format specificato nel primo argomento. Il formato contiene delle **specifiche di conversione** che sono delle direttive su come trattare i caratteri letti dallo stream;
- I risultati di tali conversioni sono memorizzate alle locazioni puntate dai puntatori forniti attraverso gli argomenti che seguono.
- Ogni argomento puntatore deve essere di un tipo appropriato per il valore restituito dalla corrispondente specifica di conversione. Esempio:

```
int n; // size in bytes
printf("Inserisci il numero di bytes da allocare: \n");
scanf("%d", &n);
```



- Legge dallo stream di input la prima parola presente e la converte in decimale %d. Se la conversione ha successo il valore è restituito in n attraverso il puntatore alla variabile.

Il C – Input e Output formattato – scanf()

- scanf() restituisce il numero di conversioni avvenute con successo:

```
int n; // size in bytes
int r; // number of successful conversions

printf("Inserisci il numero di bytes da allocare: \n");
r = scanf("%d", &n);

if ( r==0 ){
    // error
    ....
}else{
    // a valid value is stored in n
    ....
}
```

Il C – Input e Output formattato – scanf()

- se la `scanf()` legge la fine dello stream di input restituisce al chiamante la costante simbolica `EOF`. Verificare cosa succede se si esegue il seguente programma:



```
// Per far uscire il programma dal ciclo for premere CTRL-D
long cnt;
char c;
for ( cnt=0; scanf("%c",&c)!=EOF ; cnt++)
    ;
printf("Il testo che hai inserito contiene %ld caratteri.\n", cnt);
```

- `scanf()` restituisce la costante simbolica `EOF` anche qualora avvenga un errore durante la lettura dello stream. In questo caso la **variabile globale** `errno` è impostata con un codice di errore. La variabile `errno` è dichiarata in `errno.h`.
- Molte funzioni di libreria standard del C utilizzano la variabile `errno` per comunicare situazioni di errore durante la loro esecuzione. I programmi possono usare la funzione di libreria `perror(char *msg)` per stampare un messaggio descrittivo dell'errore il cui codice è memorizzato in `errno`, attraverso la stringa `msg` il programmatore può inoltre aggiungere altro testo al messaggio predefinito.
- Per approfondimenti sull'utilizzo della funzione `scanf()`, dei codici di errore possibili, si rimanda alla pagina di manuale:

`man scanf`

Il C – Input e Output formattato – scanf()

```
int n; // size in bytes
int r; // number of successful conversions

printf("Inserisci il numero di bytes da allocare: \n");
r = scanf("%d", &n);

if ( r!=EOF ){

    if ( r==0 ){
        // error the user did not enter an integer
        ....
    }else if ( r>0 ){ {
        // a valid value is stored in n
        ....
    }else{
        // an error scanf incurred
        perror("The program is going to terminate...\n");
        ...
    }

}
```

Il C – Input e Output formattato – scanf()

```
#include <stdio.h>

int input_line_flush(){
    int c;
    while ( (c=getchar())!=EOF && c!='\n' )
        ;
    return c;
}
```

Il C – Input e Output formattato – scanf()

```
int main(){
    int i, n;
    char c;
    printf("Please enter a line starting with: \"Lun\"
           <integer number> <character>.\nThe rest of the
           line will be flushed.\n");
    while ( (n=scanf("Lun %d %c", &i, &c)) != EOF ){
        if (n!=2){
            printf("It's not good. Retry please\n");
            input_line_flush();
        }else{
            printf("OK You entered correctly. Bye bye!\n");
            return 0;
        }
    }
    printf("Bye bye!\n");
    return 1;
}
```


Il C – Esercizio su funzione scanf

Esercizio A.2 Usando la funzione scanf() della libreria standard di I/O scrivere un programma che chieda all'utente di inserire su una linea e separandoli con degli spazi bianchi dei valori numerici **interi positivi non nulli**. La lista deve essere terminata con il valore 0 (end of list). Il programma deve calcolare la media aritmetica dei valori inseriti e deve stampare il risultato sullo standard output.

```
$ Inserisci una lista di valori interi positivi non  
nulli:  
$ <input> 0  
$ La media dei valori è: <output>
```

Il C – Esercizio su funzione scanf

Esercizio A.3 Usando la funzione `scanf()` della libreria standard di I/O scrivere un programma che chieda all'utente di inserire un carattere. In base al carattere inserito il programma deve stampare: "Il carattere che hai inserito è un numero.", "Il carattere che hai inserito è una lettera dell'alfabeto." oppure "Il carattere che hai inserito non è alfanumerico.".

Esercizio A.4 Usando la funzione `scanf()` della libreria standard di I/O scrivere un programma che chieda all'utente di inserire il numero `n` di voti dei quali fare la media aritmetica. Successivamente il programma legge dallo standard input un voto per linea e al termine ne calcola la media.

Il C – Direttiva #define

- la direttiva `#define` definisce nomi simbolici per valori testuali utilizzati nel programma:

```
#define MAX_NO_STUDENTS 100
```

```
for ( i=0 ; i<MAX_NO_STUDENTS ; i++ ){  
    ...  
}
```

- il pre-processor sostituirà prima della compilazione tutti i simboli `MAX_NO_STUDENTS` con il valore testuale `100`;
- se una **costante simbolica** è utilizzata in più moduli sorgenti C è buona prassi editare un file di intestazione (.h) contenete la definizione, da includere in ogni modulo. Una successiva modifica del valore della costante richiede l'editing del solo file di intestazione.

mydefs.h:

```
#define  
MAX_NO_STUDENTS  
100
```

main.c:

```
#include "mydefs.h"  
  
....  
....  
....  
for ( i=0 ;  
i<MAX_NO_STUDENTS ;  
i++ ){  
    ....  
}  
....
```

studente.c:

```
#include "mydefs.h"  
  
....  
....  
....  
if (  
st<MAX_NO_STUDENTS){  
    ....  
}  
....  
....
```

Il C – Direttiva #define

- il campo della direttiva che specifica il valore può contenere qualsiasi testo:

```
#define MAX_NO_STUDENTS    100  
#define CLASSROOM_A_CAPACITY 40  
#define OVERFLOW          (MAX_NO_STUDENTS - CLASSROOM_A_CAPACITY)
```



- il pre-processore legge i moduli C in un passo;
- **attenzione:** il numero di caratteri del nome simbolico è limitato, nelle prime definizioni del C era 8; inoltre non è possibile usare tutti i caratteri.

Il C – Direttiva #define

- la direttiva `#define` può essere usata anche per definire `macro`:

```
#define ceiling(a,b) ((a+b-1)/b)
```

```
#include <stdio.h>
#include "/mydefs.h"

int main(){
    int      n = 47; // total number of students
    const int m = 8; // number of snacks per pack

    printf("I need %d packs of snacks for %d students.\n",
           ceiling(n,m), n);
}
```

- il pre-processore sostituisce l'uso della macro, con il corpo della definizione compiendo una riscrittura dei nomi *formali* dei parametri:

```
int main(){
    int      n = 47; // total number of students
    const int m = 8; // number of snacks per pack

    printf("I need %d packs of snacks for %d students.\n",
           ((n+m-1)/m), n);
}
```

Il C – Direttiva #define

- I file di intestazione .h possono includere altri file di intestazione. Potrebbero verificarsi delle inclusioni multiple: supponiamo che `mydefs.h` includa il file `math_macro.h`:

`mydefs.h`

```
#include "math_macro.h"  
...  
...
```



- `main.c` dovendo utilizzare sia le macro matematiche sia alcune definizioni in `mydefs.h` li include entrambi:

`main.c`

```
#include "math_macro.h"  
#include "mydefs.h"  
...
```

- quando il pre-processore elabora il file `main.c` include nel testo risultante il contenuto del file `"math_macro.h"` due volte: la prima volta per la direttiva `#include "math_macro.h"` in `main.c`, la seconda volta per la direttiva `#include "math_macro.h"` in `mydefs.h`

Il C – Direttiva #define

- **#define** in unione alla direttiva **#ifndef** può essere utilizzata per escludere l'inclusione multipla durante il pre-processing del contenuto del file.



math_macro.h

```
#ifndef _MATH_MACRO_H // Il pre-processore processa le linee che seguono solo
                      // se _MATH_MACRO_H non è definita
#define _MATH_MACRO_H // Il pre-processore quando incontra questa direttiva
...                  // colleziona la definizione del simbolo _MATH_MACRO_H
...
...
#endif
```

Il C – Ancora sulle variabili globali

- le variabili globali sono allocate in fase di compilazione e sono visibili, previa dichiarazione, anche in moduli diversi da quello nel quale sono definite;
- se definisco una variabile globale di classe `static` allora **il suo nome è visibile solo nel modulo nel quale è definita.**

main.c

```
static int index;  
int error_code;  
....  
int main(){...}  
....
```

// definizione della variabile globale error_code

studente.c

```
extern int error_code;  
....  
....  
int myfunction(){  
    ....  
    error_code=DOESNOTEXIST;  
    return -1;  
}
```

// dichiarazione di variabile definita in un altro modulo C

Il C – Le strutture dati nel C

Le strutture dati del C sono:

- gli array []

```
int a[10];
char *msg = "OK You entered correctly. Bye bye!\n";
char str[] = {'H', 'e', 'l', 'l', 'o', '\0' };
char *weekdays[] = {"Mon", "Tue", "Wen", "Thu", "Fri" };
```

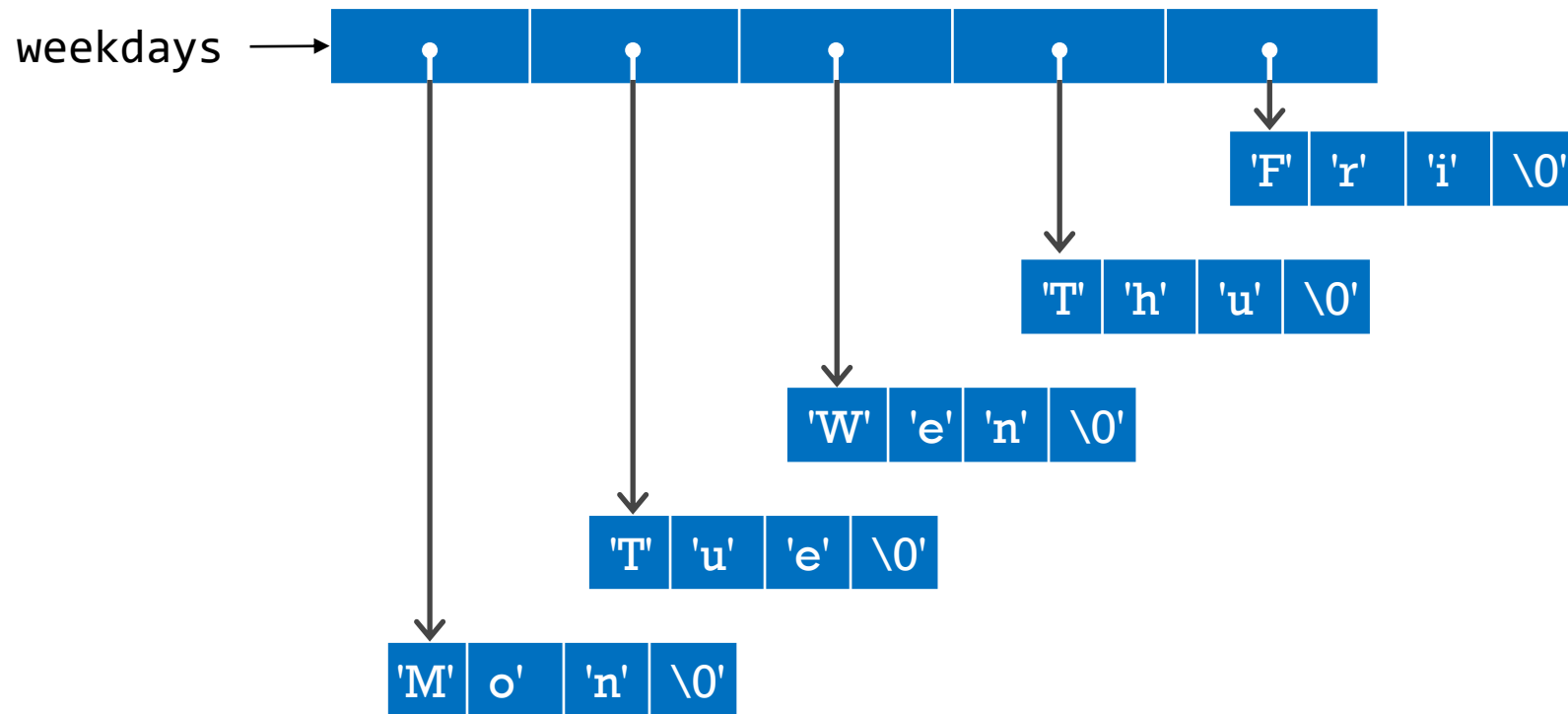
- le struct

```
// definizione di una struttura
struct student {
    int    matr;
    char *first_name;
    char *last_name;
    int    voti[40];
    struct address adr;
};

// definizione di dati con struttura student
struct student pippo, pluto, paperino;
```

- Strutture dati diverse (esempio liste, alberi, etc etc) devono essere implementate astrattamente tramite rappresentazioni ad array o a struct.

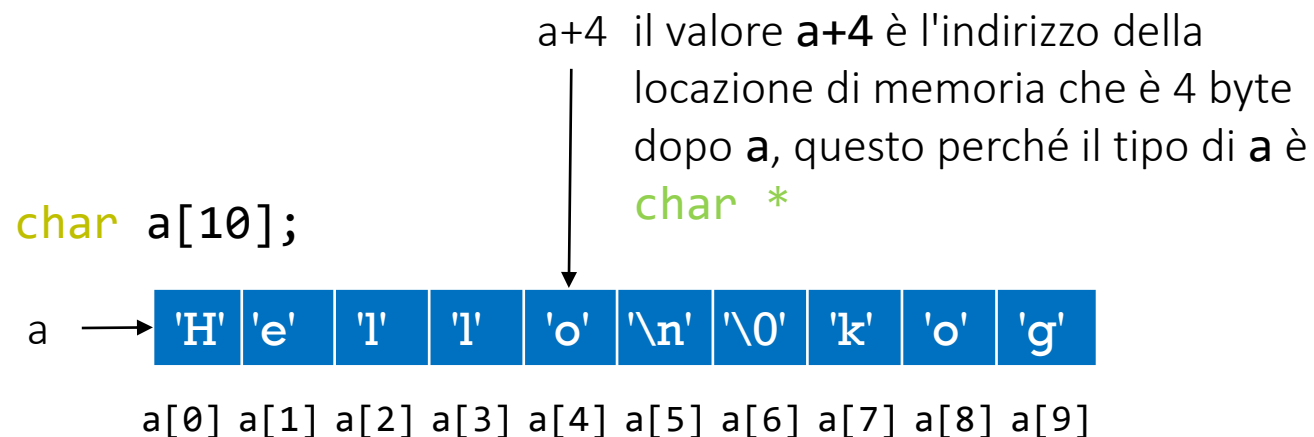
Il C – Gli array



- Allocazione alternativa della memoria per l'array di stringhe:

```
char *weekdays_b = "Mon\0Tue\0Wen\0Thu\0Fri\0";
```

Il C – Gli array



- il nome di un array in C è l'indirizzo del primo elemento quindi le seguenti espressioni sono tra loro equivalenti:

`*a, a[0]` : elemento in posizione 0

`a[3], *(a+3)` : elemento in posizione 3

`&a[4], a+4` : indirizzo dell'elemento in posizione 4

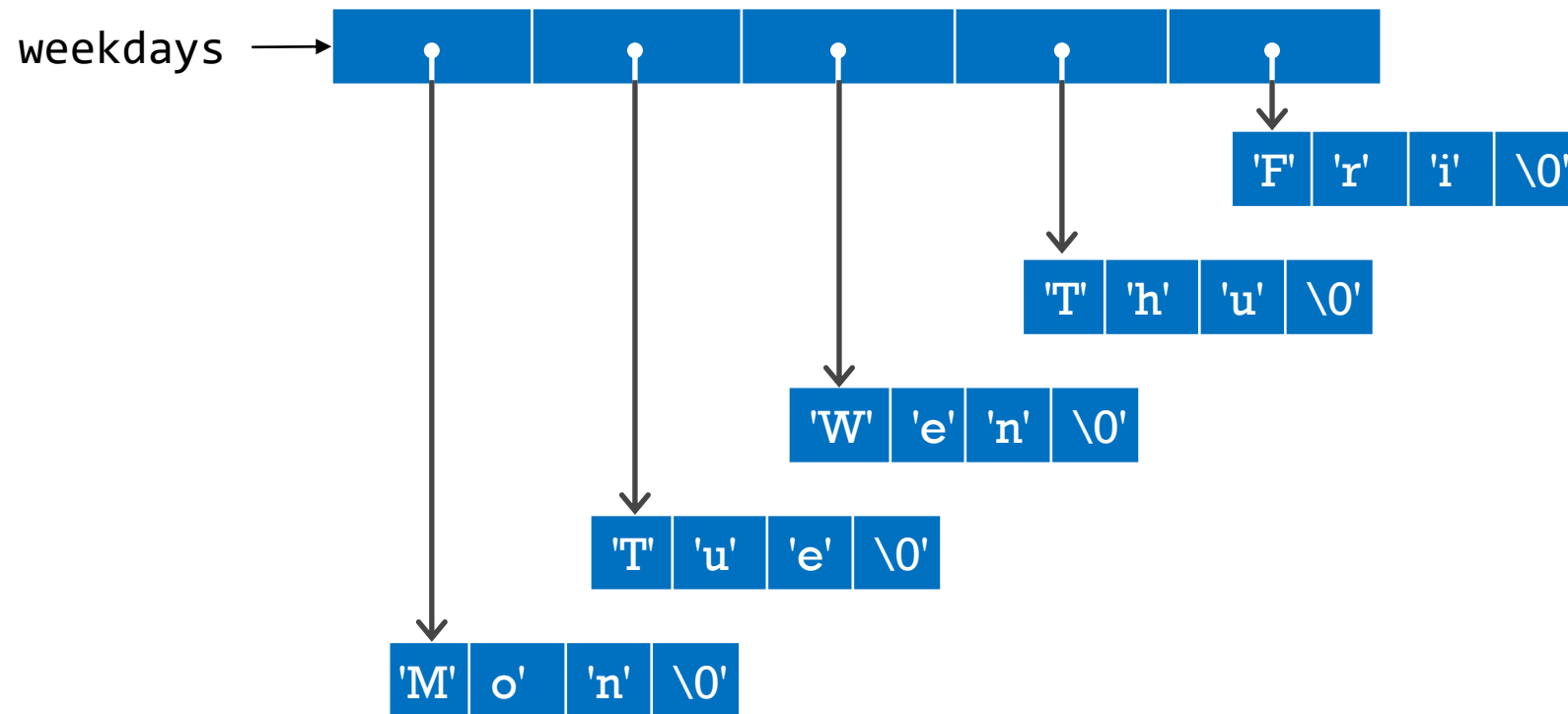
- a non si può modificare perché il nome di un array è un `const`, si può però modificare una variabile puntatore, definita nel seguente modo, usata per accedere agli elementi dell'array direttamente, senza passare attraverso la lettura di un indice:

`char *aptr = a+1;`

Il C – Gli array

- Allocazione della memoria per l'array di stringhe:

```
char *weekdays[] = {"Mon", "Tue", "Wen", "Thu", "Fri"};
```



Il C – Gli array

- uso di un puntatore per scorrere un array:

```
#define MAX_ITEMS 10
int main(){
    int a[MAX_ITEMS];
    int *aptr;
    for ( aptr=a ; aptr<(a+MAX_ITEMS) ; aptr++ )
        *aptr=0;
    printf("Ho azzerato l'array!\n");
}
```

- usando una variabile indice sarebbe stato:

```
#define MAX_ITEMS 10
int main(){
    int a[MAX_ITEMS];
    int i;
    for ( i=0 ; i<MAX_ITEMS ; i++ )
        a[i]=0; // *(a+i)=0
    printf("Ho azzerato l'array!\n");
}
```

Il C – Gli array

- Gli array possono essere passati come argomenti delle funzioni. La funzione `min(int a[])` che segue determina il minimo tra i valori memorizzati nell'array passato come argomento:

```
int min(int a[]){
    int i;
    int min=a[0];
    for ( i=1 ; i<sizeof(a)/sizeof(int) ; i++ ) // SBAGLIATO: ERRORE LOGICO
        if (a[i]<min) min=a[i];
    return min;
}
```

- Per determinare quanti byte di memoria occupa un array si può usare l'operatore del C `sizeof` applicato al nome. **Questo non è però possibile se il nome dell'array è il parametro formale di una funzione**, è il chiamante che deve passare anche la dimensione:

```
int min(int a[], int n){
    int i;
    int min=a[0];
    for ( i=1 ; i<n ; i++ )
        if (a[i]<min) min=a[i] ;
    return min;
}
```

Il C – Gli array

- Nelle chiamate di funzione gli array sono passati per riferimento. La funzione precedente poteva essere difatti anche dichiarata come:

```
int min(int *a, int n);
```

```
int min(int *a, int n){  
    int min=*a;  
    int *ptr;  
    const int *end = a+n;  
    for ( ptr=a+1 ; ptr<end ; ptr++ ){  
        int e = *ptr;  
        if (e<min) min=e;  
    }  
    return min;  
}
```

Esercizio A.5

Scrivere un programma che legge dallo standard input un testo e conta in esso la frequenza di ogni carattere dell'alfabeto (senza distinzione tra stile minuscolo e maiuscolo) e la frequenza degli spazi bianchi. **Facoltativo:** usando il carattere * come mattone elementare dare una rappresentazione ad istogramma orizzontale delle frequenze delle lettere dell'alfabeto, esempio:

```
a: *****  
b: *****  
c: *****  
d: ****  
...
```

Premessa agli esercizi seguenti A.6 e A.7

Redigere un file di nome `string_utilities.c` contenente una collezione di funzioni per operare su stringhe secondo le specifiche degli esercizi A.6 e A.7. Redigere il corrispondente file di intestazione `string_utilities.h`. Testare il funzionamento delle funzioni su stringa scrivendo la funzione `main()` in un modulo C separato.

Il C – Le stringhe C – Esercizi

Esercizio A.6 Implementare le seguenti funzioni su stringhe C:

1. `int slength(char s[])`: restituisce la dimensione in caratteri della stringa passata come argomento, `-1` in caso di errore;
2. `char *scut_last_word(char s[])`: cancella logicamente dalla stringa passata come argomento l'ultima parola senza ri-allocarla; restituisce il puntatore alla stringa stessa;
3. `int string_is_empty(char s[])`: il metodo restituisce `1` (true) se la stringa `s` è vuota, altrimenti `0`; `-1` in caso di errore;

Esercizio A.7 Implementare le seguenti funzioni su stringhe C:

1. `int string_is_palindrome(char s[])`: restituisce `1` (true) se la stringa `s` è una palindrome, altrimenti `0`;
2. `int string_compare(char s1[], char s2[])`: confronta le stringhe `s1` e `s2` e restituisce `1` se `s1` lessicograficamente è prima di `s2`, `0` se `s1` e `s2` sono identiche, `-1` se `s1` è lessicograficamente dopo di `s2`;
3. `void string_wipe_whitespaces(char s[])`: il metodo elimina gli spazi bianchi ridondanti nella stringa `s`;
4. `int string_how_many(char c, char s[])`: restituisce il numero di occorrenze del carattere `c` nella stringa `s`.

Il C – Esercizio puntatori a stringhe costanti

Esercizio A.8 Scrivere un programma il quale legga dallo standard input una linea alla volta e restituisca per ogni parola nella linea la dimensione in caratteri. Dopo averne consultato la pagina di manuale usare la funzione `fgets()` per ottenere una linea dallo standard input e memorizzarla in un array di `char`, successivamente processare l'array per ottenere i conteggi.

Esempio di sessione:

Inserisci una linea:

> Hello, how are you?

le dimensioni delle parole inserite sono:

> 6 3 3 4

Inserisci una linea:

> <input>

Il C – Le strutture C – struct

- Segue un esempio di definizione di struct C:

```
// definizione di struct point per descrivere un punto nel piano Euclideo
struct point {
    double    x;        // abscissa coordinate
    double    y;        // ordinate coordinate
    char      name[4];   // label, 3 characters max
};
```

- La definizione di una struct non alloca memoria. Essa è costituita dalla parola chiave **struct**, da una **etichetta** opzionale, utile a nominare la definizione nel seguito, e tra parentesi graffe dalla dichiarazione dei **campi** che compongono la struttura;
- Nel codice che segue sono invece **allocate** tre variabili rispettivamente di nomi a, b e o strutturate secondo struct point; mentre a e b non sono inizializzate, o è inizializzata al momento della definizione:

```
// definizione di dati con struttura struct point
struct point a, b;
struct point o = { .0, .0, {'0','\0', , } };
```

Il C – Le strutture C – struct

```
// assegnamento di valori ai campi di variabili strutturate struct point
a.x = 2.4; a.y = .0; a.name[0] = 'A'; a.name[1] = '\0';
b.x = 9.3; b.y = 4.2; b.name[0] = 'B'; b.name[1] = '\0';
```

- Posso definire array di struct. Esempio:

```
// definizione ed inizializzazione di un array di tipo struct point
struct point retta[2] = { { 0, .0, {'C', '\0', ,} }, { 3.2, 2.9, {'D', '\0', ,} } };
// Per accedere ai campi di una variabile strutturata si usa la dot-notation:
// nome-variabile.nome-campo
```

- L'operatore di assegnamento è definito tra variabili struct sempre che queste siano state definite usando la stessa etichetta (non posso operare assegnamenti tra struct con diversa etichetta sebbene abbiano campi identici):

```
a = 0;
retta[0] = b;
```

Il C – Le strutture C – struct

- Diversamente dal trattamento degli array, il valore assegnato al nome di una variabile di tipo struct è il valore del dato e non l'indirizzo, eccetto nell'assegnamento quando il nome è scritto a sinistra dell'operatore (lvalue);
- Le strutture sono **passate per valore** nelle chiamate di funzioni:

```
// funzione con argomenti dei valori strutturati 'struct point'
double distance(struct point a, struct point b){
    double d1 = b.x-a.x;
    double d2 = b.y-a.y;
    return sqrt( d1*d1+ d2*d2);
}
```

```
// uso della funzione distance
d = distance(retta[1], o);
```

- se distance modificasse a o b allora modificherebbe delle copie locali senza avere alcun effetto sui dati del chiamante, nell'esempio retta[1] o o.

Il C – Le strutture C – struct

- Una funzione può restituire una struct. Questa è restituita per valore.

```
// funzione con argomenti dei valori strutturati 'struct point'
struct point vsum(struct point a, struct point b){
    struct point s;
    s.x = a.x + b.x;
    s.y = a.y + b.y;
    s.name[0]='\0';
    return s;
}
```

Il C – Le strutture C – struct

- Si può ottenere l'indirizzo di una struct con l'operatore &. Attraverso l'indirizzo una struct può quindi essere **passata per riferimento** ad una funzione. Ad esempio:

```
// Imposta casualmente le coordinate del punto passato per riferimento
void random_point(struct point *a){
    a->x = rand()/((double)RAND_MAX);
    a->y = rand()/((double)RAND_MAX);
}
```

- Nel chiamante:

```
struct point p;
p.name[0] = 'P'; p.name[1]='\0';
random_point(&p);
```

- la chiamata a funzione modificherà la variabile p attraverso il puntatore.
- Riassumendo: se a è un indirizzo di una struttura struct point allora per accedere ai campi della struttura tramite il puntatore si usa la notazione ->:

```
a->name[1] // carattere memorizzato alla posizione 1 dell'array name
(*a).x    // con dereferenzamento del puntatore accede al campo x
*(a->name) // è il primo carattere memorizzato nell'array name a->name[0]
struct point b;
b = *a;    // assegna a 'b' il dato memorizzato all'indirizzo 'a'
```

Il C – Le strutture C – struct

- Si può ottenere la dimensione in byte di un tipo struct o di un dato di quel tipo usando l'operatore sizeof. Ad esempio:

```
int n = sizeof(struct point);  
  
struct point a;  
int n = sizeof(a);
```

- Inoltre per i vincoli di allineamento dei dati, in generale è:

```
sizeof(struct point) >= sizeof(double) + sizeof(double) + sizeof(char)*4
```


Il C – Il tipo enumerativo: enum

- In C è possibile definire degli insieme di valori simbolici per enumerazione usando il costrutto enum:

```
enum nome_insieme { valore [, valore]* };
```

```
// Definizione di un insieme 'day' per enumerazione  
enum day { MON, TUE, WEN, THU, FRI, SAT, SUN };
```

- Una variabile di tipo enum day potrà assumere i valori simbolici forniti nella definizione. Ad esempio:

```
// Definizione di una variabile con valori su un insieme enumerativo  
enum day rest = MON;  
....  
....  
if ( rest==WEN) {  
....  
}
```

Il C – Il tipo enumerativo: enum

- Il compilatore assegnerà dei valori interi positivi crescenti ai valori simbolici specificati nella enumerazione a partire da 0. Quindi:

```
enum day a = WEN+1; // a ha valore iniziale 3 quindi THU
```

- Il programmatore può modificare il valore numerico iniziale dal quale partire nella definizione dell'enumerazione. Ad esempio:

```
// Definizione di un insieme 'day' per enumerazione  
enum day { MON=1, TUE, WEN, THU, FRI, SAT, SUN };
```

Il C – Il tipo enumerativo: enum

- Usando l'enumerazione è possibile definire un insieme di valori simbolici booleani:

```
// Definizione di un'enumerazione di valori booleani
enum boolean { FALSE, TRUE };
```

```
// Definizione di una variabile enum boolean
enum boolean is_digit = FALSE;
....
....
if ( c <= '9' && c >= '0' ) {
    is_digit = TRUE;
    ....
}
```

- è importante aver scritto nell'enumerazione per i booleani prima il valore FALSE e poi il valore TRUE. In questa maniera è possibile usare direttamente il valore simbolico come valore di verità nelle espressioni:

```
if ( is_digit ) {
    ....
}
```

Il C – Definizione di nuovi tipi: `typedef`

- In C è possibile definire nuovi tipi di dato usando l'operatore `typedef`:

```
typedef tipo_base tipo_derivato;
```

- Alcuni esempi:

```
typedef unsigned int matricola;

typedef struct _student { matricola id;
                           char *first_name;
                           char *last_name;
                           unsigned int voti[40];
                        } student;
```

- In ogni caso, il C non esegue un type-matching stretto sui tipi derivati. Quindi il nuovo tipo `matricola` sopra definito segue le regole di matching usate per gli `unsigned int`.

```
// definizione di variabili di tipo student
student qui, quo, qua;
struct _student paperino;
```

Il C – Definizione di nuovi tipi: `typedef`

- Esempio di definizione del tipo di dato boolean:

```
typedef enum _boolean {FALSE, TRUE} boolean;
```

- nella definizione il *tipo base* è '`enum {FALSE, TRUE}`' il *tipo derivato* è `boolean`.

```
boolean is_digit = FALSE;  
enum _boolean flag = TRUE;
```

Il C – Le union

- La definizione di una union C è simile alla definizione di una struct C ma la semantica è differente.

```
union val {  
    int i;  
    double f;  
    char c;  
};
```

```
union val t;
```

- La union dell'esempio può essere usata sia per memorizzare i valori della variabile t.i di tipo int, sia i valori della variabile t.f di tipo double, sia i valori della variabile t.c di tipo char, **ma in maniera esclusiva**

```
for ( t.i=0 ; t.i<NO_STUDENTS ; t.i++){  
    ....  
}  
....  
t.c=getchar();  
if ( t.c == 'A' ){  
    ....  
}
```

Il C – Le union

- il compilatore alloca per la variable `t` uno spazio sufficientemente grande da accomodare un qualsiasi campo della union. Nell'esempio se `int` è di 4 byte e `double` è di 8 byte allora una variabile di tipo union `val` sarà allocata in uno spazio di 8 byte.
- è compito del programmatore farne un uso corretto;

```
t.i = 0;  
t.c = 'A';  
for ( ; t.i < NO_STUDENTS; t.i++) { // uso logicamente errato  
    ....  
    ....  
}
```

Il C – Ancora sulle funzioni e i puntatori

- In C è possibile definire una variabile di tipo **puntatore a funzione**. Una tale variabile memorizza l'indirizzo della prima istruzione del testo della funzione alla quale fa riferimento.

```
// Definizione della funzione min per un array di float
float min(float a[], int n){
    // cond: n the size is such that n>0
    int i;
    float min = a[0];

    for (i=1;i<n;i++)
        if (a[i]<min) min=a[i];
    return min;
}
```

```
// Dichiarazione della funzione min per un array di float
extern float min(float [], int);

float (*f_ptr)(float [], int); // Definizione del puntatore f_ptr a funzione.
f_ptr = min;                  // f_ptr è inizializzata con l'indirizzo della
                              // funzione min, la scrittura 'f_ptr = &min;'
                              // è equivalente

float m;
float a[] = {3.4, 5.9, 0.0, 6.7 };
m = f_ptr(a, sizeof(a)/sizeof(float)); // chiama la funzione puntata da f_ptr
                                         // la scrittura 'm = (*f_ptr)(a,...);' è
equivalente
```


Il C – Ancora sulle funzioni e i puntatori

- Definire un puntatore a funzione `fptr` affinché possa essere usato per indirizzare la funzione `getchar`:

```
int (*fptrt)(void);
```

- Definire un puntatore a funzione `fptr` affinché possa essere usato per indirizzare la funzione `putchar`:

```
int (*fptr)(int);
```

- Definire un puntatore a funzione `fptr` affinché possa essere usato per indirizzare la funzione `fgets`:

```
char *(*fptr)(char *, int, FILE *);
```

Il C – Ancora sulle funzioni e i puntatori

- E' possibile definire **array di puntatori a funzione**. Nell'esempio che segue viene definito un array di puntatori a funzioni matematiche definite esternamente e che operano su array di **float**.

```
// Alcune definizioni simboliche degli indici delle operazioni su array di float
#define MIN  0
#define MAX  1
#define MEAN 2

// Dichiarazione delle funzioni matematiche su array di float
extern float min(float [], int);
extern float max(float [], int);
extern float mean(float [], int);

// Definizione di un array che è un indice per le funzioni su array di float
float (*operations[])(float [], int) = {min,max,mean};

// Esempio d'uso per il calcolo della media
float a[] = {3.4, 5.9, 0.0, 6.7 };
const int n = sizeof(a)/sizeof(float);
float mean;
mean = (*operations[MEAN])(a,n);    // calcola la media dei valori in a[]
```

Il C – Ancora sulle funzioni e i puntatori

- In C è possibile passare una funzione F come parametro durante una chiamata di una funzione G;
- La funzione F è passata **per riferimento**;
- Nella dichiarazione della funzione G bisogna specificare il tipo del parametro formale come puntatore a funzione di tipo appropriato. Ad esempio:

```
void setup( int (*)(void) , short);
```

- dichiara setup una funzione che non restituisce alcun valore e che ha due parametri formali, rispettivamente: il primo di tipo `int (*)(void)` quindi puntatore a una funzione senza argomenti che restituisce un valore `int`; il secondo di tipo `short`;
- naturalmente, nella definizione di setup bisogna dare dei nomi ai parametri formali:

```
void setup( int (*handler)(void), short parameters){  
    ....  
    ....  
}
```

Il C – Ancora sulle funzioni e i puntatori

- Esempio di chiamata di setup:

```
// Dichiarazione di puntatore a funzione definito esternamente
int (*handler)(void);
handler = getchar;           // remainder: int getchar(void)

// Esempio di chiamata di setup
setup(handler, 0xa01f);
```

- oppure:

```
setup(getchar, 0xa01f);
```

Il C – Ancora sulle funzioni e i puntatori

- Un esempio:

```
// Funzioni di gestione dei segnali previsti dall'applicazione
extern void s_overflow_handler(int);
extern void s_underflow_handler(int);
extern void s_division_by_zero_handler(int);
extern void s_out_of_memory_handler (int);
....
....
```

```
void signal_handling(int sender, void (*specific_handler)(int)){

    // Pre
    ....
    ....

    // Ad-hoc Signal Management
    specific_handler(sender);

    // Post
    ....
    ....
}
```

```
signal_handling(buffer, s_out_of_memory_handler);
```

Il C – Operatori bitwise e Shift

- Gli operatori bitwise in C sono:

or-bitwise |

and-bitwise &

not-bitwise ~

- essi lavorano a livello dei bit che formano i valori. Alcune esempi:

`0xA3 & 0xB1 = 1010 0011 & 1011 0001 --> 1010 0001 = 0x21`

`0xA3 | 0xB1 = 1010 0011 | 1011 0001 --> 1011 0011 = 0xB3`

`~0x4F = 0100 1111 --> 1011 0000 = 0xB0`

- Operatore di shift-right >> e shift-left <<. Esempi:

`0xFF>>2 = 1111 1111 >> 2 --> 0011 1111 = 0x3F`

`1101 >> 1 -> 0110 val & 1 1101`

`0011 mask`

Scrivere una funzione che prende un valore int come argomento e ne restituisce il resto della divisione per 2 (usare operatori bitwise per calcolare il risultato).

```
return val&1;
```

Il C – L'accesso ai file

- Vedremo due modi per accedere ai file: **1.** attraverso l'uso di funzioni che operano direttamente le chiamate di sistema e che usano i **descrittori dei file**; **2.** con funzioni di più alto livello che lavorano sul dato di tipo FILE ed operano una bufferizzazione del flusso dei dati.
- L'accesso diretto avviene attraverso l'uso di alcune funzioni della libreria standard `unistd`. Esse sono:

```
int    open(const char *pathname, int flags, mode_t mode);  
  
ssize_t read(int fd, void *buf, size_t count);  
  
ssize_t write(int fd, void *buf, size_t count);  
  
off_t  lseek(int fd, off_t offset, int whence);  
  
int    close(int fd)
```

- consultare le pagine di manuale per esplorare alcune altre funzioni ad esse correlate ed i file di intestazione da includere.

Il C – L'accesso ai file attraverso le System Call: **open**

- Prima di poter leggere o scrivere un file è necessario aprirlo.

- La **open** :

```
int      open(const char *pathname, int flags [, mode_t mode] );
```

- `pathname` è il nome del file da aprire;
- i `flags` specificano alcune modalità per l'apertura e l'accesso al file. Alcuni di questi sono sono:

`O_CREAT`, `O_TRUNC`

flag di creazione del file, influenzano la semantica della modalità di apertura;

`O_APPEND`

flag di stato del file, influenza la semantica di successive operazioni di lettura e scrittura;

`O_RDONLY`, `O_WRONLY`, `O_RDWR`

flag della modalità di accesso.

```
open("./Divina Commedia.txt", O_RDONLY);
```

```
open("/home/depiero/app/strings.c", O_RDWR | O_CREAT | O_TRUNC );
```


Il C – L'accesso ai file attraverso le System Call: **open**

- In caso di successo la open restituisce il **numero di descrittore di file** che il sistema operativo ha associato all'istanza di apertura, altrimenti -1 ed errno è impostata con il codice dell'errore (vedere il man per i tipi di errore); .

Il C – L'accesso ai file read

- La read :

```
ssize_t read(int fd, void *buf, size_t count);
```

- cerca di leggere all'offset corrente count byte dal *file descriptor* fd e li memorizza all'indirizzo buf;
- in caso di successo restituisce il numero di byte effettivamente letti, 0 in caso di *fine del file*; altrimenti in caso di errore -1 ed errno è impostata con il codice dell'errore (vedere il man per i tipi di errore);

```
#define BUFFER_SIZE 1024

int div_com = open("../Divina Commedia.txt", O_RDONLY);

if ( div_com != -1 ){
    char text[BUFFER_SIZE];
    ssize_t n = read(div_com, text, BUFFER_SIZE);
    ....
}else{ .... }
```

Il C – L'accesso ai file read

- Un esempio di utilizzo errato:

```
int div_com = open("../Divina Commedia.txt", O_RDONLY);

if ( div_com != -1 ){
    char *text;
    size_t n = read(div_com, text, 1024);
    ....
}else{ .... }
```

- L'indirizzo text passato alla read non indica alcuna area di memoria allocata allo scopo. Addirittura text ha un valore arbitrario (pericoloso). Meglio sarebbe stato inizializzarlo a NULL.

Il C – L'accesso ai file `write`

- La `write`:

```
ssize_t write(int fd, void *buf, size_t count);
```

- cerca di scrivere all'`offset` corrente `count` byte sul *file descriptor* `fd` leggendoli dal buffer `buf`;
- in caso di successo restituisce il numero di byte effettivamente scritti; in caso di errore restituisce `-1` ed `errno` è impostata con il codice dell'errore (vedere il man per i tipi di errore);

```
int f = open("/home/depiero/app/strings.c", O_RDWR | O_APPEND );

if ( f != -1 ){
    // Aggiunge in fondo al file strings.c la stringa "Hello World"
    char *text = "Hello World";
    size_t n = write(f, text, 11);
    ....
}else{ .... }
```

Il C – L'accesso ai file `lseek`

- La `lseek` (posizionamento):

```
off_t  lseek(int fd, off_t offset, int whence);
```

- posiziona l'offset di lettura del file descriptor `fd`, in base al valore di `whence`:

`SEEK_SET`

L'offset del file è posto a `offset` bytes;

`SEEK_CUR`

L'offset del file è posto `offset` bytes in avanti, dal valore corrente;

`SEEK_END`

L'offset del file è posto alla dimensione del file più il valore `offset`.

```
int f = open("/home/depiero/app/strings.c", O_RDWR);
if ( f != -1 ){
    // Equivale ad aprire in append il file
    lseek(f, 0, SEEK_END);
    ....
}else{ .... }
```

Il C – L'accesso ai file `lseek`

- in caso di successo `lseek` restituisce il nuovo valore dell'offset associato al file descriptor `fd`, altrimenti restituisce `-1` ed `errno` è impostata con il codice dell'errore (vedere il man per i tipi di errore);
- il posizionamento dell'offset oltre la dimensione del file introduce un "hole", eventuali letture in un "hole" restituiscono caratteri con codifica nulla `'\0'`.

```
int f = open("/home/depiero/untitled.rtf", O_RDWR);
....
// Pone l'offset all'inizio del file, andando a sovrascriverne il
contenuto
lseek(f, 0, SEEK_SET);
....
....
// Pone l'offset 4 byte in avanti dall'attuale posizione
lseek(f, 4, SEEK_CUR);
....
```

Il C – L'accesso ai file `close`

- Se l'accesso ad un file è terminato allora è importante chiuderlo. Questa operazione rilascia risorse nel sistema operativo e garantisce che tutte le modifiche fatte al file siano registrate nel file system. Inoltre eventuali lock sul file saranno rilasciati.

- La `close` :

```
int    close(int fd);
```

- in caso di successo `close()` restituisce 0, altrimenti restituisce -1 ed `errno` è impostata con il codice dell'errore (vedere il man per i tipi di errore).

Il C – L'accesso ai file attraverso stream `stdio`

- Un altro modo per accedere ad un file è aprire uno `stream` ed associarlo al file. Uno stream è acceduto attraverso un puntatore a `FILE`, tipo derivato definito nella libreria `stdio`.

- La `fopen` apre uno stream e lo associa al file con nome `pathname`:

```
FILE *fopen(const char *pathname, const char *mode);
```

- `mode` specifica la modalità di apertura del file:

fopen() mode	open() flags
r	O_RDONLY
w	O_WRONLY O_CREAT O_TRUNC
a	O_WRONLY O_CREAT O_APPEND
r+	O_RDWR
w+	O_RDWR O_CREAT O_TRUNC
a+	O_RDWR O_CREAT O_APPEND

Rapporto tra stringa `mode` della `fopen()` e `flags` nella `open()`.

Il C – L'accesso ai file attraverso stream `stdio`

```
FILE *f = fopen("../Divina Commedia.txt", "r");
```

- La `fopen` restituisce un puntatore allo stream, il tipo per tale puntatore è `*FILE`. Esso deve essere usato in successive operazioni sul file. Alcune di queste sono:
`fread()` `fwrite()` `fgets()` `fgetc()` `fclose()` `fseek()` `fscanf()` `fprintf()`
- In generale le funzioni della libreria `stdio` su stream restituiscono EOF se raggiungono la fine del file. Si rimanda alla pagina di manual `man` per ulteriori dettagli sul loro utilizzo.

Il C – L'accesso ai file attraverso stream `stdio`

- Nel file `stdio.h` sono dichiarati i seguenti stream:

```
extern FILE *stdin; // stream collegato allo standard input,  descrittore 0
extern FILE *stdout; // stream collegato allo standard output, descrittore 1
extern FILE *stderr; // stream collegato allo standard error,  descrittore 2
```

- Quindi le due chiamate sono del tutto equivalenti:

```
int c;
c=fgetc(stdin);
```

```
int c;
c=getchar();
```

Il C – Allocazione dinamica della memoria dati

- Alcune funzioni della libreria standard `stdlib` sono per l'allocazione dinamica della memoria dati. Esse sono principalmente le seguenti, la loro dichiarazione è nel file di intestazione `#include <stdlib.h>`:

```
void *malloc(size_t size);
```

```
void *calloc(size_t nmemb, size_t size);
```

```
void *realloc(void *ptr, size_t size);
```

```
void free(void *ptr);
```

- La `malloc()` alloca `size` bytes dallo *heap* e restituisce in caso di successo un puntatore alla memoria allocata. La memoria **non è inizializzata** a 0.
- La `calloc()` alloca memoria per un array di `nmemb` elementi ciascuno di **dimensione `size` bytes**. In caso di successo restituisce un puntatore alla memoria allocata. La memoria **è inizializzata** a 0. Fa il controllo dell'overflow di `nmemb*size`;
- La `realloc()` modifica la dimensione di un blocco di memoria precedentemente allocato e puntato da `ptr` alla nuova dimensione `size` bytes. `ptr` deve essere un puntatore precedentemente restituito da una chiamata a `malloc()` o `calloc()` o `realloc()`.

Il C – Allocazione dinamica della memoria dati

Il contenuto della memoria reallocata viene mantenuto se la nuova dimensione è maggiore, la parte aggiuntiva **non viene azzerata**, altrimenti viene troncato.

- La `free()` de-alloca il blocco di memoria puntato da `ptr` e precedentemente allocato con una `malloc()`, una `calloc()` o una `realloc()`.


In caso di errore o se nessuna allocazione è stata fatta le funzioni `malloc()`, `calloc()` e `realloc()` restituiscono `NULL`.

```
// Allocazione dinamica di un array di int
int *voti = (int *)calloc(40, sizeof(int) );
if (voti!=NULL){
    ....
    voti[i]=30; // *(voti+i)=30;
}
....
free(voti);
```

Il C – Allocazione dinamica della memoria dati

```
// Allocazione dinamica di una struct student
struct student *donald = (struct student *)malloc(sizeof(struct student));

if (donald!=NULL){
    donald->id = 348;
    donald->first_name = "Donald";
    donald->last_name = "Duck";
    ....
}
....
free(donald);
```



Il C – Implementazione di una lista in C

- Le liste sono strutture dati di natura dinamica, hanno due operazioni principali che sono l'*inserimento* e il *prelievo*. Nel seguito useremo le struct C e l'allocazione dinamica dei dati per implementare in maniera efficiente un dato di tipo list. Implementeremo dapprima una lista di clienti, il tipo di dato si chiamerà list_cl:

cl_list.h

```
#include "client.h"

// Definizione del tipo 'l_node' usato per rappresentare i nodi della lista
typedef struct _node {
    unsigned int    id;
    client          person;
    struct _node    *next;
} l_node;

// Definizione del tipo list_cl (lista di client)
typedef struct {
    l_node *head;
    l_node *tail;
} list_cl;

#define L_EMPTYLIST_CL {NULL,NULL}
```

Il C – Implementazione di una lista in C

```
// Definizione di una lista di studenti inizialmente vuota  
list_cl class = L_EMPTYLIST_CL;
```

cl_list.h

```
// Operazioni sulla lista  
list_cl  l_add_cl(list_cl l, client p);  
client   l_rem_cl(list_cl l);
```

■ Definizione del tipo client:

client.h

```
#include "address.h"  
  
// Definizione del tipo 'client'  
typedef struct {  
    char    cf[16];  
    char    *first_name;  
    char    *last_name;  
    address adr;  
} client;
```

Il C – Implementazione di una lista in C

```
list_cl class;
```

```
class.head
```

```
class.tail
```

id person next

HEAD

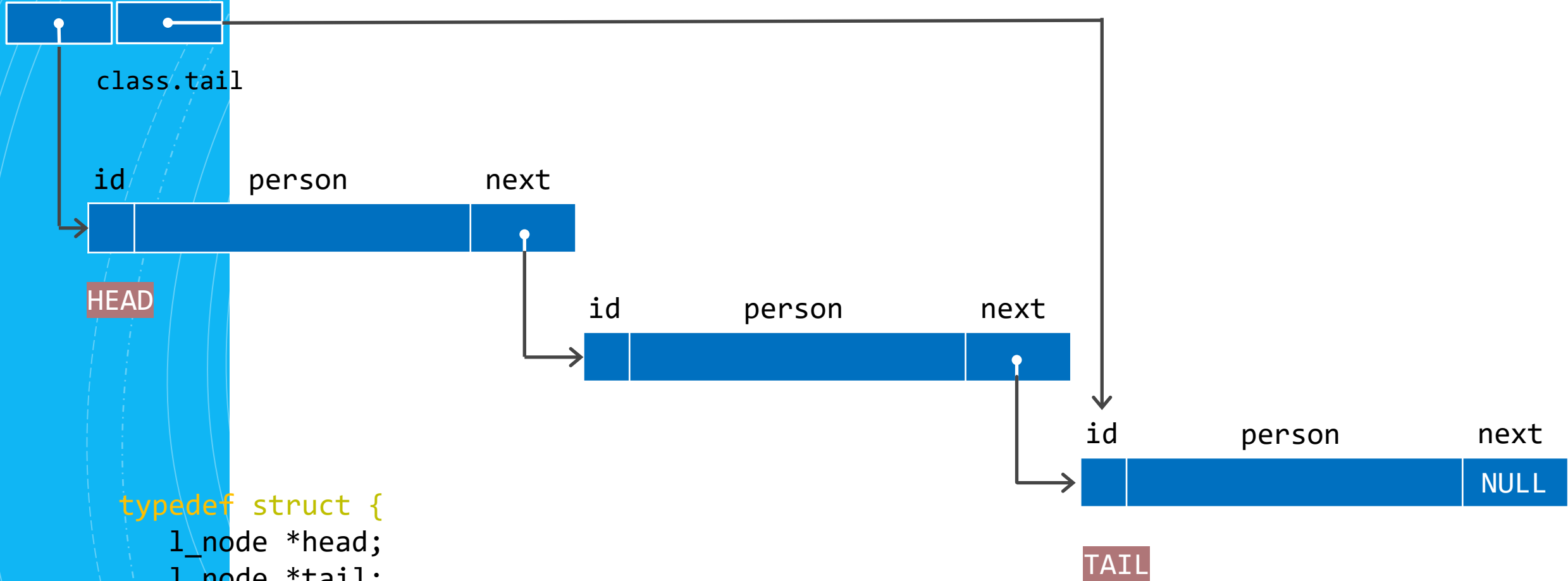
id person next

id person next

NULL

TAIL

```
typedef struct {  
    l_node *head;  
    l_node *tail;  
} list_cl;
```



Il C – Implementazione di una lista in C

Esercizio A.9 Implementare le seguenti funzioni sul dato `list_cl`:

1. `list_cl l_add_cl(list_cl l, client p)`: aggiunge il cliente `p` in coda alla lista `l`; restituisce la lista stessa, la lista vuota in caso di errore;
2. `client l_rem_cl(list_cl l)`: restituisce il cliente in testa alla lista `l`, il cliente vuoto se la lista è vuota;
3. `int l_is_empty(list_cl l)` restituisce `1` se e solo se la lista `l` è vuota;
4. `list_cl l_clear(list_cl l)` rimuove tutti gli elementi dalla lista `l` rilasciando le risorse allocate per essi; restituisce la lista vuota;
5. `int l_length(lists_cl l)` restituisce il numero di elementi nella lista `l`.

C – An implementation of a list abstract data type

Exercise A.9 Implement the following functions operating on the `list_cl` data type:

1. `list_cl l_add_cl(list_cl l, client p)`: adds client `p` at the end of list `l`; returns the list itself, the empty list if an error occurred;
2. `client l_rem_cl(list_cl l)`: returns the client at the head of list `l`, the empty client if the list has not got any client;
3. `int l_is_empty(list_cl l)`: returns `1` if and only if list `l` is empty;
4. `list_cl l_clear(list_cl l)`: removes all the elements from list `l` releasing the resources allocated for them; returns the empty list;
5. `int l_length(lists_cl l)` returns the number of elements in list `l`.

Esercizi sulle liste semplicemente collegate

Esercizio A.10 Si considerino le seguenti definizioni di tipo, dove **data** è un tipo definito esternamente:

```
#include "data.h"

typedef struct _node {
    data *d;
    struct _node *next;
} node;

typedef node *list;

#define L_EMPTYLIST NULL
```

La lista è rappresentata tramite un puntatore all'elemento in testa. Definire la seguente funzione la quale inserisce alla posizione **i** della lista **l** il dato **d**:

```
list list_add_at(unsigned int i, data *d, list l)
```

Contare gli elementi della lista a partire dalla posizione 0. Quindi: **list_add_at(0, d, coda)** inserisce in testa alla lista **coda** il dato puntato da **d**.

Esercizi sulle liste semplicemente collegate

Esercizio A.12 Si considerino le seguenti definizioni di tipo:

```
typedef struct _node {  
    int val;  
    struct _node *next;  
} node;
```

```
typedef node *list;
```

```
#define L_EMPTYLIST NULL
```

Il tipo `list` rappresenta una lista di valori interi. La lista è implementata tramite un puntatore all'elemento in testa. Definire la seguente funzione:

```
list *list_split(list l)
```

la quale divide la lista `l` in due sotto-liste tali che la prima contenga solo gli elementi di `l` con valore pari e la seconda contenga solo gli elementi di `l` con valore dispari. Le due liste così ottenute sono restituite dalla funzione in un array.

Esercizi sulle liste semplicemente collegate

Esercizio A.13 Si considerino le seguenti definizioni di tipo:

```
typedef struct _node {  
    int val;  
    struct _node *next;  
} node;
```

```
typedef struct {  
    node *head;  
    node *tail  
} list;
```

```
#define L_EMPTYLIST NULL
```

Il tipo **list** rappresenta una lista di valori interi. La lista è implementata tramite due puntatori: uno all'elemento in testa (campo **head**) ed uno all'elemento in coda (campo **tail**). Definire la seguente funzione:

```
list list_invert(list l)
```

la quale inverte l'ordine degli elementi in **l**. Ad esempio se **l** è **(2,5,3,8,1)** allora **list_invert(l)** restituisce una lista così composta **(1,8,3,5,2)**. Valutare di risolvere l'esercizio definendo la una funzione ricorsivamente.

Esercizi: stack di elementi

Esercizio A.14 Implementare un dato astratto Stack con le due operazioni classiche di push (inserimento in cima) e pop (prelievo dalla cima).

```
stack stack_push(stack s);  
stack stack_pop(stack s, void **d);
```

Lo stack può contenere elementi di tipo diverso o non specificato. E' necessario usare quindi puntatori generici ai dati mantenuti nella struttura a pila. Il secondo parametro della funzione stack_pop è necessario per *passare all'indietro* al chiamante il dato prelevato dallo stack.

S

UGGERIMENTI:

```
typedef struct _node {  
    void *data;  
    struct _node *below;  
} node;  
  
typedef struct {  
    node *base;  
    node *top;  
} stak;  
  
#define STACK_EMPTY {NULL,NULL}
```

Esercizio su accesso a file

Esercizio A.11 Scrivere un programma di nome **print** e dal seguente uso:

```
print <filename1> [<filename2> ...]
```

il quale stampi sullo standard output il contenuto dei file testuali i cui nomi sono passati come argomenti nell'ordine specificato dall'utente.

Usare le funzioni **open**, **read**, **write** e **close**.

In caso un file non esista stampare un messaggio di errore (usare **errno** e **perror**).

Il C—Ancora sulle variabili locali

- è **un errore** restituire una variabile locale per riferimento. Esempio:

```
int *function(void){  
    int a[10];  
    ....  
    ....  
    return a;  
}
```

- quando la funzione `function()` termina le sue variabili locali sono deallocate, usare il riferimento nel chiamante produrrebbe degli errori logici.
- È possibile cambiare la **classe di memorizzazione** di una variabile locale in `static`. La variabile così definita non è deallocata al termine della funzione sebbene continui ad avere una visibilità locale. Tale variabile oltre a mantenere il valore può essere restituita per riferimento.

Il C—Ancora sulle variabili locali

- Esempio:

```
void function(void){  
    static int i=0;  
    printf("i:%d  ",i);  
    i++;  
}  
  
int main(){  
    for (int k=0 ; k<10 ; k++)  
        function();  
}
```

output:

```
i:0  i:1  i:2  i:3  i:4  i:5  i:6  i:7  i:8  i:9
```

Il C – variabili locali e funzioni *static*

- La **classe di memorizzazione** *static* applicata ai **simboli globali** (funzioni e variabili globali) restringe la loro visibilità al solo modulo C dove sono definiti:

myutilities.c

```
static int counter=0;

static int handler(void){
    ....
    ....
}
```

- sia la variabile `counter` sia la funzione `handler()` sono simboli **non visibili** ai moduli esterni, quindi il seguente modulo darebbe nel linking un **errore** di *simbolo non definito*:

main.c

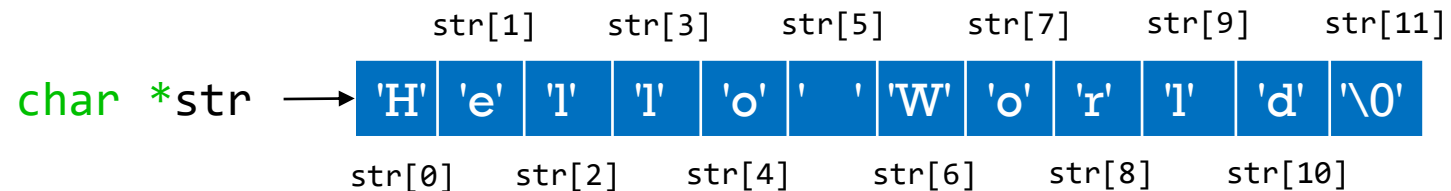
```
extern int counter;
extern int handler(void);

int main(){
    ....
    ....
}
```

Il C – Le stringhe C

- Il C **rappresenta** le stringhe come array di **char**. Nella rappresentazione viene aggiunto un carattere in fondo ad indicare la fine della sequenza in memoria, questo è il carattere con codifica nulla: `'\0'`.

```
char *str = "Hello World"; // str punta ad una stringa costante
```



```
*str == 'H'           char
str[0] == 'H'         char
*(str+6) == 'W'        char
str[6] == 'W'         char
*str+6 == 'N'          char
&str[6] == str+6       char *
```

- Attenzione:** `sizeof(str)` non calcola la dimensione della struttura puntata, quindi l'array, ma la dimensione del puntatore, questo perché `str` non è un nome **const** di array. Anche `sizeof(*str)` non produce la dimensione della struttura puntata ma la dimensione del tipo **char**.
- Un qualsiasi array di **char** che rispetti questa rappresentazione è una stringa C.