Programmazione 2 - C Language

- Studieremo i linguaggi imperativi → procedurali
  → ad oggetti

in particolare i linguaggi imperativi procedurali sono quelli che andremo ad approfondire maggiormente

SINTASSI E SEMANTICA

SINTASSI

La sintassi dice quali programmi sono validi e quali no per risolvere un dato problema

SEMANTICA

Indica quali operazioni vengono eseguite e come vengono cambiati gli stati del calcolatore

La Macchina Astratta

- Una **macchina astratta permette di definire un linguaggio indipendentemente dalla piattaforma in uso.** Sarà poi il compilatore ad occuparsi di eseguire il codice su ogni computer. Un esempio di compilatore è la JVM (Java Virtual Machine)
- La macchina astratta permette il paradigma

Write once, compile everywhere

La fase di traduzione in C

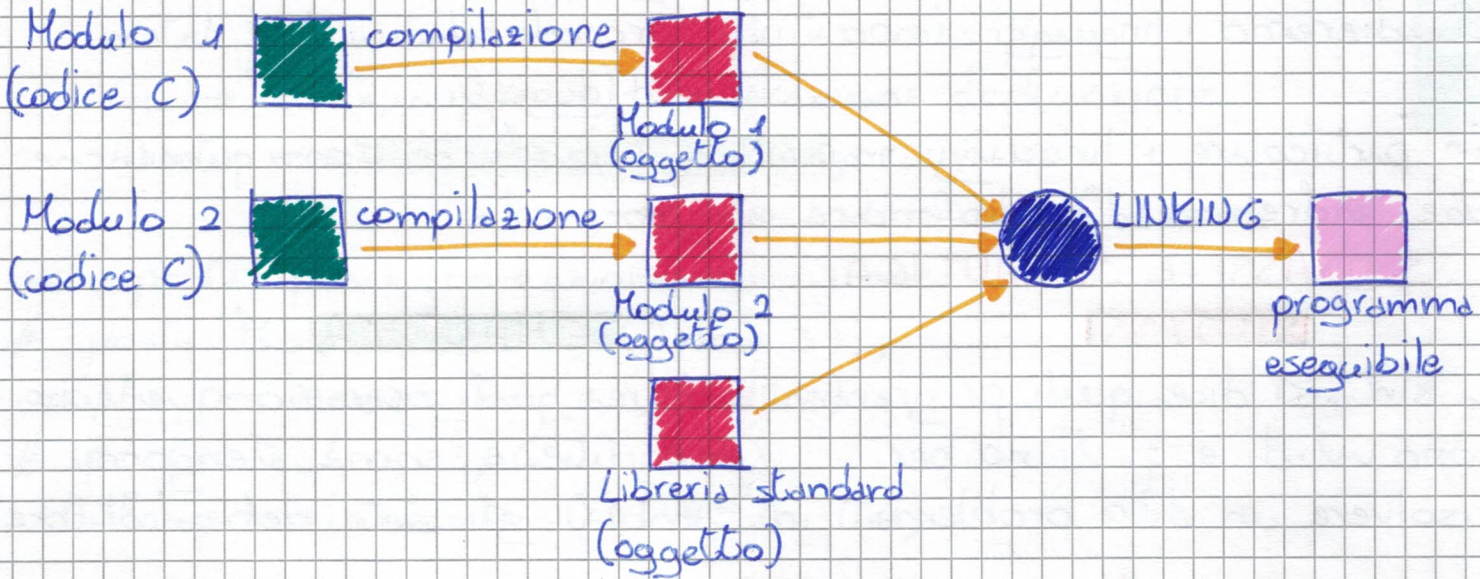
1) la compilazione del programma

- il compilatore trasforma le istruzioni del programma in linguaggio macchina
- il codice del programma C è detto **codice sorgente**
- il codice macchina ottenuto è detto **codice oggetto**
- il codice oggetto è strettamente legato all'hardware

2) la fase di collegamento (linking)

- in generale un programma complesso è composto da tante parti che si chiamano **moduli**
- ogni modulo può essere compilato separatamente e dà origine ad un **modulo oggetto**
- il collegamento tra i moduli viene fatto dal **linker**
- questo metodo consente la **creazione** e il **riutilizzo** di **librerie** standard definite dal programmatore

schema di funzionamento



Gli header

I file header contiene costanti, direttive di pre-processore, dichiarazioni di strutture, prototipi di funzioni e in alcuni casi l'implementazione dei prototipi delle funzioni.

Macchina Astratta C

- segue l'architettura di Von Neumann
- stream in input: dati usati dal programma in esecuzione
- stream in output: dati prodotti dal programma
- memoria → programma: contiene il codice del programma in esecuzione
  → dati: contiene i dati manipolati durante l'esecuzione del programma

Quindi cos'è un programma C?

Un programma C è una sequenza finita di istruzioni che sono interpretabili dalla macchina astratta C.

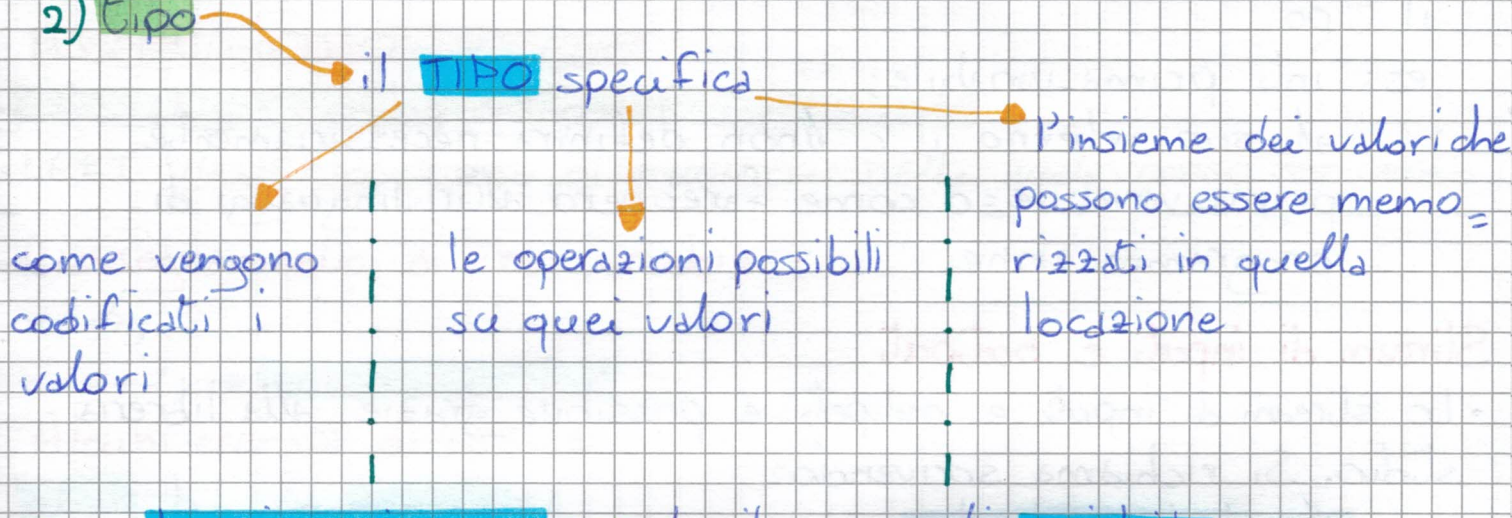
Stato della macchina astratta

Lo stato della macchina astratta C è dato da:
1) indicatore della prossima istruzione del programma da eseguire
2) contenuto della memoria dati

l'esecuzione di un'istruzione altera lo stato della macchina astratta perché modifica almeno l'indicatore della prossima istruzione da eseguire.

Come accedere alla memoria dati?

- La memoria dati è suddivisa in porzioni chiamate locazioni
- le locazioni di memoria sono caratterizzate da:
  1) indirizzo
  2) tipo



- una locazione in uso prende il nome di variabile

Le VARIABILI in C

~~VALORE~~

VARIABILE = locazione con un tipo ed un valore associati.

Il nome di una variabile è detto identificatore e permette di accedere ad una locazione di memoria senza conoscerne l'indirizzo.

Tipi elementari in C

nome	dimensione (B)	range
• char	1	-128 / 127
• unsigned char	1	0 / 255
• short	2	-32768 / 32767
• unsigned short	2	0 / 65535
• int	4	-2.147.483.648 / 2.147.483.647
• unsigned int	4	0 / 4.294.967.295
• long	8	-2^{63} / $2^{63}-1$
• unsigned long	8	0 / 2^{64}
• float	4	$1 \cdot 10^{-37}$ / $1 \cdot 10^{37}$ +/-
• double	8	$1 \cdot 10^{-308}$ / $1 \cdot 10^{308}$ +/-
• long double	16	dipende...

Alcune specifiche sulle variabili in C:

1) scrivere una lettera o un numero tra apici equivale a scrivere un int corrispondente al codice ASCII, per esempio
2) il C non fa assunzioni sul contenuto della variabile, eccetto il tipo
   es: int prima_variabile;
   al suo interno il C non definirà necessariamente prima_variabile=0 come farebbero altri linguaggi di programmazione.

Stream di input e output

- Lo stream di input e output è possibile grazie alla libreria stdio. Si richiama scrivendo:
  `#include <stdio.h>`
- per acquisire dallo stream di input il valore di una variabile si usa la funzione scanf
  es: `scanf("%d", &prova);`
  - identificatore del tipo di variabile presa in input - SEGUA FOSTO
  - nome della variabile in cui inserire il valore ricevuto in input

NB: prima del nome della variabile ci va &

Tabella degli identificatori:

Codice	Formattazione
c	carattere
d oppure i	int con segno
f	float
s	string
u	uint
p	indirizzo di mem.
lf	long float (=double)

- per scrivere qualcosa a schermo e produrre così un output si usa la funzione printf
  es: printf("%d", prova);

- ASSEGNAMENTO = x = 12.0
- CASTING = cambio di tipo di una variabile
- LEFT VALUE = locazione di memoria nella quale posso scrivere

---

Riassegnamento di una variabile

x = 0;

x = x + 1; → incremento di 1

Alcuni esempi di codice

```
char b; /* 1 byte */
int a; /* 4 byte */
int main(){
   b = 'k';
   a = 'e';
   b = a + 1; → assegnamento }
```

} il codice appartenente ad una funzione è detto compound instruction

---

```
int x = 0;
double pi = 3.1415;
```

Le espressioni booleane in C

- In C le espressioni booleane vengono valutate in fase di esecuzione e non in fase di compilazione
- per utilizzare le variabili di tipo bool in C è necessario importare la libreria stdbool.h, oppure si possono utilizzare gli int in questo modo: val = 0 → false
  val ≠ 0 → true
- operatori booleani in C:
  ==, >, <, >=, <=, &&, ||, !val (=not)

Esempio:

```
#include <stdbool.h>
bool prova = true;
int n = 5;
if (prova == true && n <= 6){
   ...
}
```

Cicli WHILE

Algoritmo di Euclide per il calcolo del MCD

```
int a, b;
int main() {
   scanf("%d", &a); /* a>0*/
   scanf("%d", &b); /* b>0 */
   while (a!=b) {
      if (a>b)
         a=a-b;
      else /* b>a */
         b=b-a;
   }
   /* a==b */
   printf("MCD: %d \n", a);
   return 0;
}
```

while (condizione vera), esegui il compound instruction

Cicli FOR

```
for (i=1; i<=10; i=i+1)
   printf("%d", i);
```

INIT

CONDIZIONE

INCREMENTO

ISTRUZIONI

USO PREFERENZIALE FOR: le variabili che compaiono nella condizione non vengono alterate nel corpo del ciclo ma solo nelle clausole di incremento.

Cicli Do While

```
do{
   scanf("%d", &n);
} while (n<0);
```

esegue sempre l'istruzione almeno una volta

se la condizione del while è vera riesegue le istruzioni

Operatori di incremento e decremento

```
int a;
double b;
```

`a++; b++;` → incrementatori post fissi
`++a; ++b;` → incrementatori prefissi
} `a=a+1; b=b+1;`

`a--; b--;` → decrementatori post fissi
`--a; --b;` → decrementatori prefissi
} `a=a-1; b=b-1;`

Esempio: `for (i=0; i<10; i++){}`

`a+=exp;`	`a=a+exp;`
`a-=exp;`	`a=a-exp;`
`a*=exp;`	`a=a*exp;`
`a/=exp;`	`a=a/exp;`

} un altro modo per scrivere gli operatori

Side Effects

Quando valuto un'espressione vengono prodotti 2 risultati:
1) VALORE
2) SIDE EFFECTS → la valutazione di un'espressione modifica lo stato della macchina astratta; modifica il contenuto di alcune locazioni di memoria

Esempio:

```
int a;
int b;
int main(){
    a=1;
    b=a++;
    b=++a;
```

b=1 ← `b=a++;`
1) valuto l'espressione a
2) incremento di 1 la var a
} PRIMA VALUTO, POI INCREMENTO

b=3 ← `b=++a;`
1) incremento di 1 la var a
2) valuto l'espressione a
} PRIMA INCREMENTO POI VALUTO

Variabili globali e locali

- Le variabili globali vengono allocate nel processo di creazione del programma e rimangono attive durante tutta l'esecuzione del programma
- Le variabili locali vengono dichiarate all'interno di un blocco ed esistono solo durante l'esecuzione del blocco stesso

Esempio:

```
int a;
int b;                    } VARIABILI GLOBALI
int main(){
    int m, n; ...         } VARIABILI LOCALI
}
```

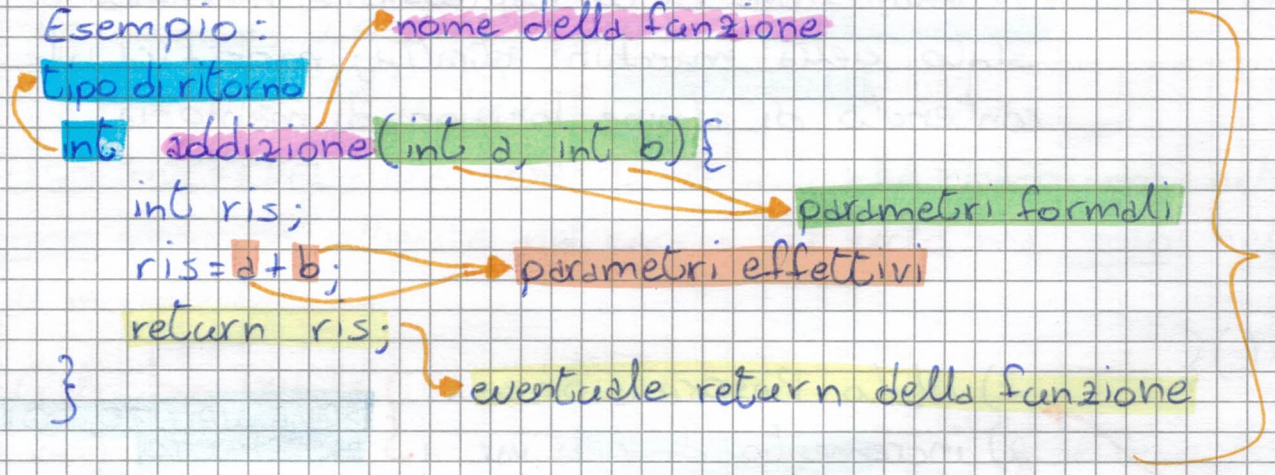
ad esempio int d; in un if

→ sono anche automatiche nel caso in cui vengano allocate e deallocate automaticamente

Le Funzioni

- Le funzioni sono uno spezzettamento del programma, così da avere tante funzioni che risolvono compiti semplici ma che se vengono utilizzate insieme possono risolvere problemi complessi

Esempio:



Le funzioni possono essere di tipo:

1) int
2) float, double } prevedono un return
3) char
4) void } non è previsto un return

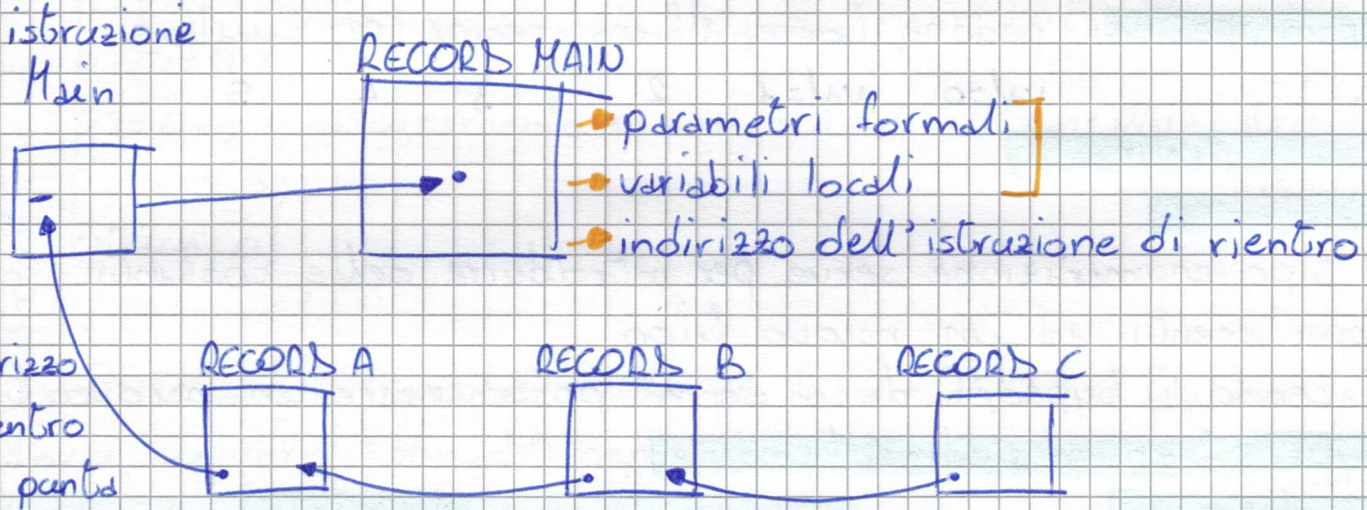
NB: il passaggio dei parametri avviene come passaggio per copia o valore, in C

Creazione e distruzione delle variabili locali

- Quando eseguo delle chiamate a delle funzioni vengono create delle variabili in un ambiente (la funzione invocata) che viene distrutto quando ne termina l'esecuzione, pertanto anche le variabili create con l'invocazione della funzione vengono distrutte.

Record di attivazione (di una funzione)

Main → A → B → C → B



- Ad ogni chiamata di funzione viene creato un record di attivazione che verrà distrutto una volta terminata l'esecuzione della funzione
- Record creati e distrutti secondo l'idea FIRST IN, LAST OUT: l'ultimo creato sarà il primo ad essere distrutto

Costanti ed Enumerazioni

1) Costanti

- Le costanti in C possono essere di qualsiasi tipo, e non è possibile modificare il valore assegnatoli al momento dell'inizializzazione
- Esempio:

```
const double pi = 3,1415;
```

2) Enumerazioni

- Esempio:

```
enum pedine = {pedone, torre, alfiere, regina, re, cavallo}
              val=0   val=1   2        3       4    5

enum pedine x;
x = alfiere;
```

- il tipo enumerazione serve per introdurre delle costanti appartenenti ad un nuovo tipo
- facendo il typedef di un enum posso creare un nuovo tipo

```
typedef enum pedine tpedine;
tpedine x;
```

dichiarati

- i tipi possono essere ~~importati purché vengano importati~~ dopo l'include delle librerie:

```
#include <stdio.h>
{dichiarazione dei tipi
```

Le strutture in C - Struct

```
struct punto2D {
    double x;
    double y;
};
int main() {
    struct punto2D p1, p2;
    p1.x = 12.3;
    p1.y = 0.7;
    p2 = p1;
    p2.x = 2.3;
    return 0;
}
```

campi della struct

assegnamento di campi

. operatore di selezione

assegnamento di una struct ad un'altra struct

- Posso usare il typedef anche per le struct, così da non dover scrivere "struct" ogni volta che ne definisco una:

```
typedef struct punto2D tpunto2D;
```

↳ dopo aver definito la struct

- **È possibile nidificare una struct dentro ad un'altra.**

Esempio:

```
struct triangolo{
    tpunto2D p1;
    tpunto2D p2;
    tpunto2D p3;
};
typedef struct triangolo ttriangolo;
int main(){
    ttriangolo t;
    t.p1.x = 0.5;
    t.p1.y = 0.5;
    t.p2.x = 3.0;
    t.p2.y = 2.5;
    ...
```

Gli Array / Vettori

- gli **array** sono degli **insiemi di variabili** dello **stesso tipo**
  - **contigue**
  - **accessibili tramite** un **indice** di tipo **int**
- Esempio:

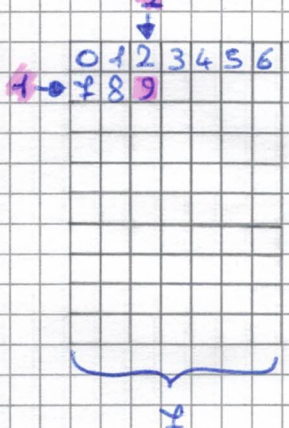
```
int main(){                  // in C deve essere una costante
    double a[100];
    a[0] = 2.3;
    a[1] = a[0] + 2.0;
                 // indice di tipo int con 0 ≤ i ≤ len-1
```

- posso inizializzare gli array in modi diversi:
  1) `int a[] = {2,4,5,7,9}` → inizializza un array con n elementi e lunghezza n
  2) `int a[10];` → inizializza un array a 0 con 10 elementi
  3) `int a[10] = {2,4,5,7,9}` → inizializza un array con 10 elementi: 5 già definiti e gli altri a 0
- per **confrontare** due **array** o **copiarne** uno in un altro è necessario **eseguire l'operazione elemento per elemento** e **non** è possibile farlo direttamente **sugli array interi**

Array multi dimensionali

- `double mat[DIM1][DIM2];` → righe: $0 \le r \le DIM1-1$; colonne: $0 \le c \le DIM2-1$
- un array 2x2 (bidimensionale) compone una matrice

Linearizzazione degli array



Come ottengo il 9? `m[r][c]` — eseguito dal pc

n° riga * n° colonne + n° colonna = r * 7 + c

↳ Esempio stampa di una matrice:

```
int stampaMat(double a[][DIM2], int r, int c){
                  *
        .
        .
        .
    a[r][c]
```

*tutte le dimensioni successive alla prima devono essere specificate per permettere la linearizzazione

*tutte le dimen

Le stringhe in C

- Le **stringhe** in C possono essere viste come degli **array di cha**
  ma con una differenza sostanziale rispetto agli array:
  le stringhe **non possono contenere il carattere** '\0' → 1 Byte
  indica la fine della stringa — carattere di escape

- Esempio di stringa:

```
char str[250];
str[0]='a';
str[1]='b';
str[2]='c';
str[3]='\0'; → fine stringa
```

- costruzione di una stringa da standard input e stampa:

```
scanf("%s", str);
printf("%s", str);
```

- esempio di funzione su una stringa:

```
void upcase(char str[]){
    int i;
    for (i=0; str[i]!='\0'; i++){
        if (str[i]>='a' && str[i]<='z'){
            str[i]=str[i]-'a'+'A';
        }
    }
}
int main(){
    char my[]={'a','b','c','\0'};   ="abc" con 4B (3+1)
                ="abc" → possibile solo nell'inizializzazione
```

- assegnamento e confronto "diretto" non sono possibili:

```
if (my=="abc")
my="ab";
```

} sono errati → non confronta le stringhe / → non compila

String copy

- la **strcpy** è una funzione di **libreria** inclusa in **string.h**
- funzionamento:

```
void strcpy(char str1[], char str2[]){
    int i;
    for (i=0; str2[i]!='\0'; i++){
        str1[i]=str2[i];
    }
    str1[i]='\0';
```

COPIA UNA STRINGA IN UN'ALTRA

- sintassi: **strcpy(char dest, char src);**

String compare

- anche la **strcmp** è una funzione di **libreria** inclusa in **string.h**
- strcmp può dare 3 tipi di output differenti:
  1) str1-str2<0 → str1 viene prima di str2
  2) str1-str2=0 → str1 e str2 sono uguali
  3) str1-str2>0 → str1 viene dopo di str2

COMPARA 2 STRINGHE

- sintassi: **strcmp(char str1, char str2);**

Numeri pseudocasuali

- la **stdlib.h** contiene la funzione **rand()** che restituisce un int compreso tra 0 e RAND_MAX
  ↳ costante di libreria
- Esempio di estrazione di un numero compreso tra 0 e 1:

```
int r;
double n;
r=rand();
u=r*1.0/RAND_MAX;
```

- Estrazione di un numero compreso tra 20 e 40:

```
r=rand()%21+20;
```

40-20+1=21 possibili scelte

- con gli esempi appena visti in realtà i numeri random saranno sempre gli stessi per via del seme del generatore di numeri sempre uguale
- possiamo "risolvere" usando il clock come seme:

```
srand(time(NULL));  → setta il seme generatore sul clock
rand();
```

La ricorsione

- Per la ricorsione è necessario trovare un caso base al quale dovremo "arrivare". È necessario "figurare" il problema, astrarlo, non si può pretendere di immaginare passo per passo cosa succederà come normalmente faremmo con la programmazione iterativa.
- Esempio di funzione ricorsiva:

```
int ordinato(int *vet, int dim){
    if (dim==0 || dim==1){
        return 1;
    }
    else{
        if (vet[dim-2]<=vet[dim-1] && dim>1){
            return ordinato(vet, dim-1);
        }
        else{
            return 0;
        }
    }
}
```

I puntatori e gli indirizzi di memoria

- una variabile possiede nome, tipo, valore } allocati in una locazione di memoria a

variabile

x: | α | β | γ | δ |

1B 2B 3B 4B

es: una variabile di tipo int è composta da 4B; a indicherà il primo dei 4B

- Come scopro l'indirizzo di memoria di x? Con &

&(x) → ritorna l'indirizzo di memoria di x

espressione — variabile

- Come faccio a memorizzare gli indirizzi? Ci sono delle variabili che lo permettono? Che operazioni sono definite sui tipi indirizzo? Lo scopriremo!

---

int x;

&x → espressione di tipo int*

→ indirizzo di un int

int a;
int *b;
b=&a;

} b è un PUNTATORE alla variabile a, ovvero contiene l'indirizzo di memoria del primo byte di a, e il tipo di b ci permette di sapere quanti byte oltre al primo saranno riservati alla variabile a

NB: gli errori di tipo in C sono in compilazione

NB2: i puntatori sono sempre grandi uguali poiché contengono sempre un indirizzo di memoria

Dereferenziazione di un indirizzo

```
int a, *pa;
pa = &a;
*pa = 14; = *(&a) = 14; = a = 14;
```

Come faccio a scrivere in a usando l'indirizzo di memoria? Con * (star)

: in = espressione di tipo type
out = variabile allocata presso l'indirizzo e di tipo type

Riassumendo:

*exp → var
type* → type

&var → exp
type → type*

- Altra cosa: se voglio aggiungere 1 ad a potrei fare così:
  *pa = *pa + 1;

Puntatori e passaggio dei parametri

```
void swap(int *pa, int *pb){
    .....
}
int main(){
    int x, y;
    x = 3;
    y = 5;
    swap(&x, &y);
    .....
}
```

come parametri passo dei puntatori, quindi gli indirizzi di memoria delle variabili

PASSAGGIO PER COPIA DI INDIRIZZO

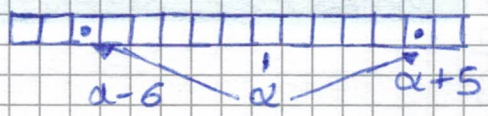
passo gli indirizzi di memoria delle variabili

Aritmetica dei puntatori

- sommare con int a type* → type*, int → type*

es: int *pa; //pa=a
pa+5;
pa-6;

} mi sposto di n unità di memoria rispetto a pa, dove unità sta per Byte occupati dal tipo di variabile del puntatore



- sottrazione tra type* e type* dello stesso tipo
  ↳ type*, type* → int
  Restituisce il numero di variabili comprese tra pb e pa (pb-pa)
- > >= < <= == → type*, type* → bool

Variabili dinamiche

- Le variabili dinamiche non hanno un nome e vi possiamo accedere solo tramite gli indirizzi di memoria
- non vengono create da dichiarazioni ma da istruzioni eseguibili (executable statement)
- la loro deallocazione è a carico del programmatore
- le variabili dinamiche non deallocate formano il garbage

Allocazione delle variabili dinamiche

- possibile grazie a stdlib.h

```
int main(){
    int *p;
    p=(int *) malloc(sizeof(int));
```

cast esplicito da void* a int*

passo alla malloc il n° di B necessari ad allocare l'oggetto desiderato e restituisce il primo Byte della variabile allocata con tipo void*

sizeof(type) → size_t
↓
int grande a sufficienza per contenere la + grande dimensione in B degli oggetti allocabili sull'architettura del pc in uso

NB: la MALLOC serve per ALLOCARE

- con la FREE DEALLOCO la memoria allocata dalla malloc. Per deallocare correttamente devo passare alla free la stessa variabile che era stata allocata dalla malloc (indirizzo del primo byte)
- es: free(p);
- ASSERT: #include <assert.h>
  assert(p!=NULL); → TRUE → continuo col programma
  → FALSE → segnalo l'errore di errata allocazione

Array dinamici

```
int main(){
    double *d;
    d=(double*)malloc(sizeof(double)*20);
    free(d);
```

variabile

alloco un array di tipo double e grande lunghezza 20

- È possibile perché la malloc viene valutata in fase di esecuzione
- per ogni malloc deve essere una free()

Allocazione di struct

```
struct mys{
    int d;
    char x;
    double vet[10];
};
int main(){
    struct mys *p;
    p=(struct mys *)malloc(sizeof(struct mys));
    p->d=14;
    p->vet[2]=13.20;
```

"->" = (*p) = dereferenziazione di p

Array multidimensionali

```
double *p; int r, c;
p=(double *)malloc(sizeof(double)*r*c);
free(p);
```

- array di array → dovrò fare una malloc per ogni array dell'array

Le liste

- Una lista di elementi di tipo T è:
  1) la lista vuota
  2) un elemento di tipo T seguito da una lista di elementi di tipo T

  es: 2 → 4 → 5 → 0 → -6 → ∧ (lista vuota)

- per avere un oggetto con la struttura di una lista dobbiamo fare la struct corretta:

```
struct cell {
    int elem;
    struct cell *next;
};
typedef struct cell *tlista;
```

IMPORTANTE

- cosa possiamo fare con una lista?
  1) aggiungere un elemento in testa → PREPEND
  2) leggere il contenuto della testa
  3) aggiungere un elemento in coda → APPEND
  4) data una lista non vuota considerare la coda della lista

Prepend

```
int prepend(tlista *pl, int e){
    tlista nuova;
    nuova = (tlista)malloc(sizeof(struct cella));
    if (nuova){ /* nuova != NULL */
        nuova->elem = e;
        nuova->next = *pl;
        *pl = nuova;
        return e; /* tutto ok */
    }
    else{
        return 0;
    }
}
```

Append

```
int append(tlista *pl, int e){
  tlista pc;
  tlista nuova;
  pc=*pl;
  nuova=(tlista)malloc(sizeof(struct cella));
  if (nuova){
    nuova->next=NULL;
    nuova->elem=e;
    while (pc->next){
        pc=pc->next;
    }
    pc->next=nuova;
    return 1;
  }
  else{
    return 0;
  }
}
```

Stampa

```
void stampalista(tlista l){
   while (l){
      printf("%d\n", l->elem);
      l=l->next;
   }
}
```

Liste circolari

- Le **liste circolari** sono delle liste caratterizzate dal fatto che **una volta arrivati all'ultimo elemento** la lista non sarà giunta alla fine ma **si tornerà al primo elemento**
- la struct cella sarà sempre la stessa ma cambia il typedef:
  typedef struct cella *tlistacirc;

Inserisci

```
int inserisci(tlistacirc *pl, int e){/*inserisce tra il 1° e il 2° elemento*
  tlistacirc nuova;
  nuova=(tlistacirc)malloc(sizeof(struct cella));
  if (nuova){
     nuova->elem=e;
     if (*pl==NULL){
        *pl=nuova;
        nuova->next=nuova;
     }
     else{
        nuova->next=(*pl)->next;
        (*pl)->next=nuova;
     }
     return 1;
  }
  else{
     return 0;
  }
}
```

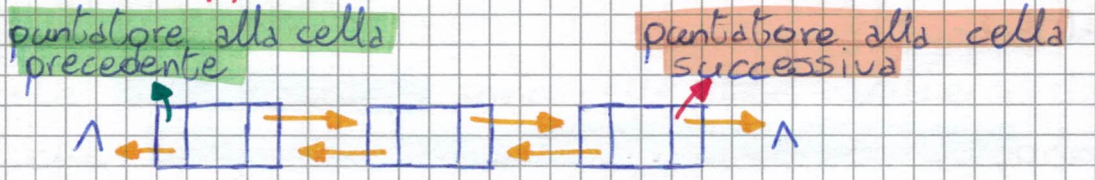
Stampa

```
void stampa(tlistacirc l){
    if (l){
        tlistacirc supp;
        supp=l;
        do{
            printf("%d\n", l->elem);
            l=l->next;
        }while (l!=supp);
    }
}
```

Distruzione

```
void distruggi(tlistacirc l){
    if (l){
        tlistacirc supp, prossimo;
        supp=l;
        do{
            prossimo=l->next;
            free(l);
            l=prossimo;
        }while (l!=supp);
    }
}
```

Liste doppiamente concatenate



- Le liste doppiamente concatenate differiscono dalle liste normali perché anziché avere un solo puntatore per cella che permette di accedere alla cella successiva, hanno 2 puntatori: uno per la cella precedente e uno per la cella successiva
- struttura:

```
struct cella {
    int elem;
    struct cella *prev;
    struct cella *next;
};
typedef struct cella *tlistadl
```

Prepend

```
int prepend(tlistadl *ph, tlistadl *pt, int e){
    tlistadl nuova=(tlistadl) malloc(sizeof(struct cella));
    if (nuova){
        nuova->elem=e;                    // aggiungo la cella vuota di inizio lista (^)
        nuova->next=*ph; nuova->prev=NULL;
        *ph=nuova;
        if(nuova->next){                  // se la lista era vuota linko la nuova cella come next e prev
            nuova->next->prev=nuova;
        }
        else{
            *pt=*ph;
        }
        return 1;
    }
    else{
        return 0;
    }
}
```

Distruzione

```
void distruggi(tlista t){
    distruggi(t->next);
    distruggi(t);
}
```

Rimozione

```
int rimuovi(tlista *l, int e){
   if (*l==NULL){
      return 0;
   }
   else{
      int k;
      k=rimuovi(&((*l)->next), e);
      if ((*l)->elem==e){
         tlist supp=*l;
         (*l)=(*l)->next;
         free(supp);
         return k+1;
      }
      else{
         return k;
      }
   }
}
```