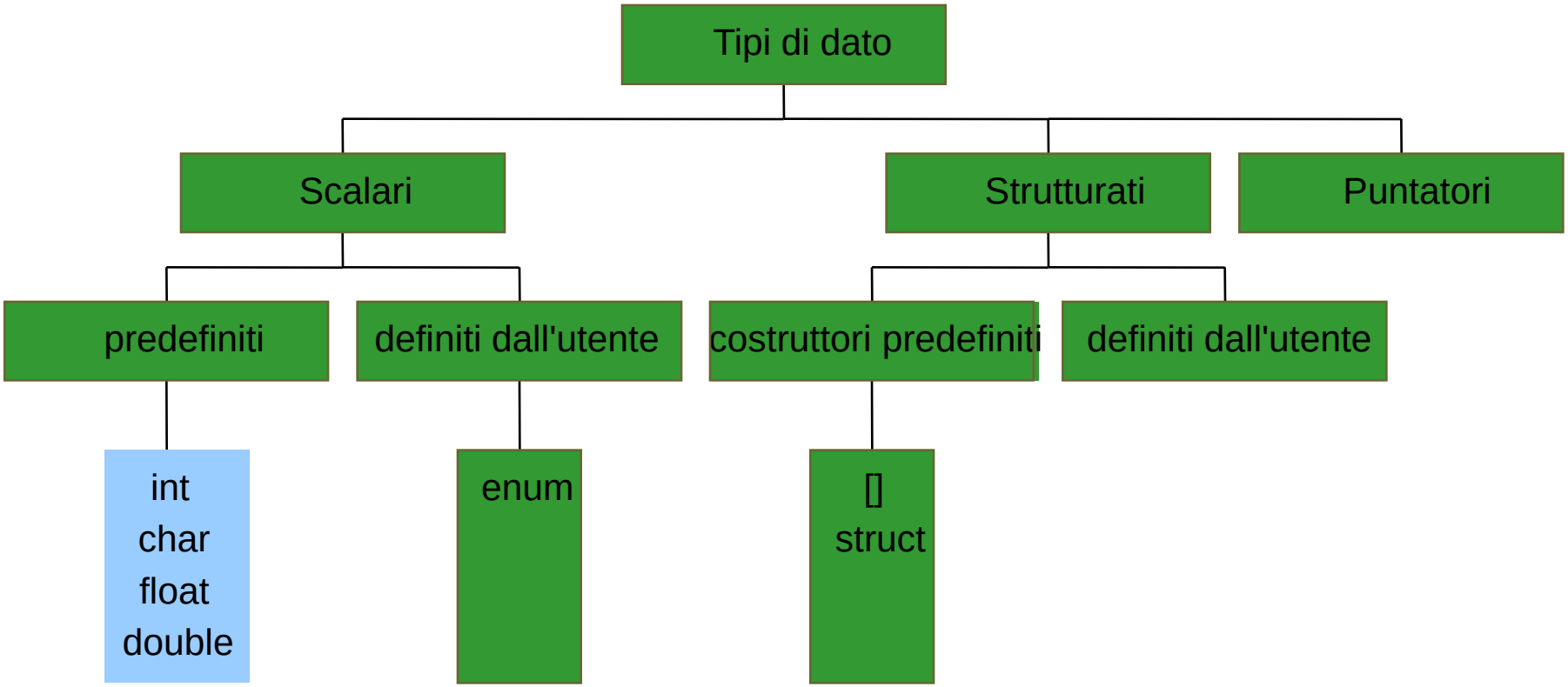
Tipi di dato

Tipi di dato


Tipi di dato
Scalari
Strutturati
Puntatori
predefiniti
definiti dall'utente
costruttori predefiniti
definiti dall'utente
int
char
float
double
enum
[]
struct

void

- void
  - è scalare
  - è predefinito
  - indica un tipo indefinito

Puntatori

- Un puntatore è una variabile che contiene un indirizzo di memoria
- Il tipo puntatore è un tipo derivato; infatti non ha senso parlare di tipo puntatore in generale ma occorre specificare a quale tipo esso punta. Tale tipo viene detto **tipo base**
- Sintassi di definizione di una variabile puntatore

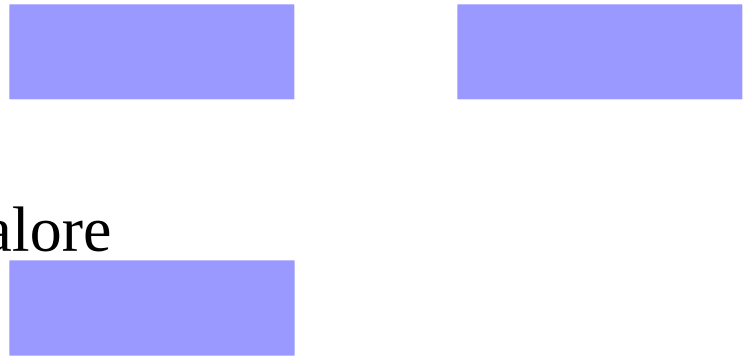
  tipobase * nomevariabile;

- Esempio
  int *p;
  float *x;
- La variabile p e x occupano lo stesso spazio in memoria, cioè quello necessario per memorizzare un indirizzo

Operatori per la manipolazione di puntatori

*	restituisce il contenuto della cella di memoria a cui punta
&	restituisce l'indirizzo dell'operando a cui è applicato

```
int valore, new;
int *p;
valore = 10;
p = &valore;
new = *p
```



Operazioni su i tipi puntatori

Assegnazione: i puntatori possono apparire nella parte destra di una espressione e il loro valore può essere assegnato ad altri puntatori, un indirizzo di memoria contenuto in una variabile puntatore può essere stampato

```
#include <stdio.h>
main ()
    {   int x, *p1, *p2;
        p1 = &x;
        p2 = p1;
        printf("%p%d", p1, *p1)
```

Confronto: sulle variabili puntatori possono essere utilizzati tutti gli operatori relazionali

Aritmetica dei puntatori

- Se p e q sono due variabili di tipo puntatore ad un certo tipo T

$$p - q$$

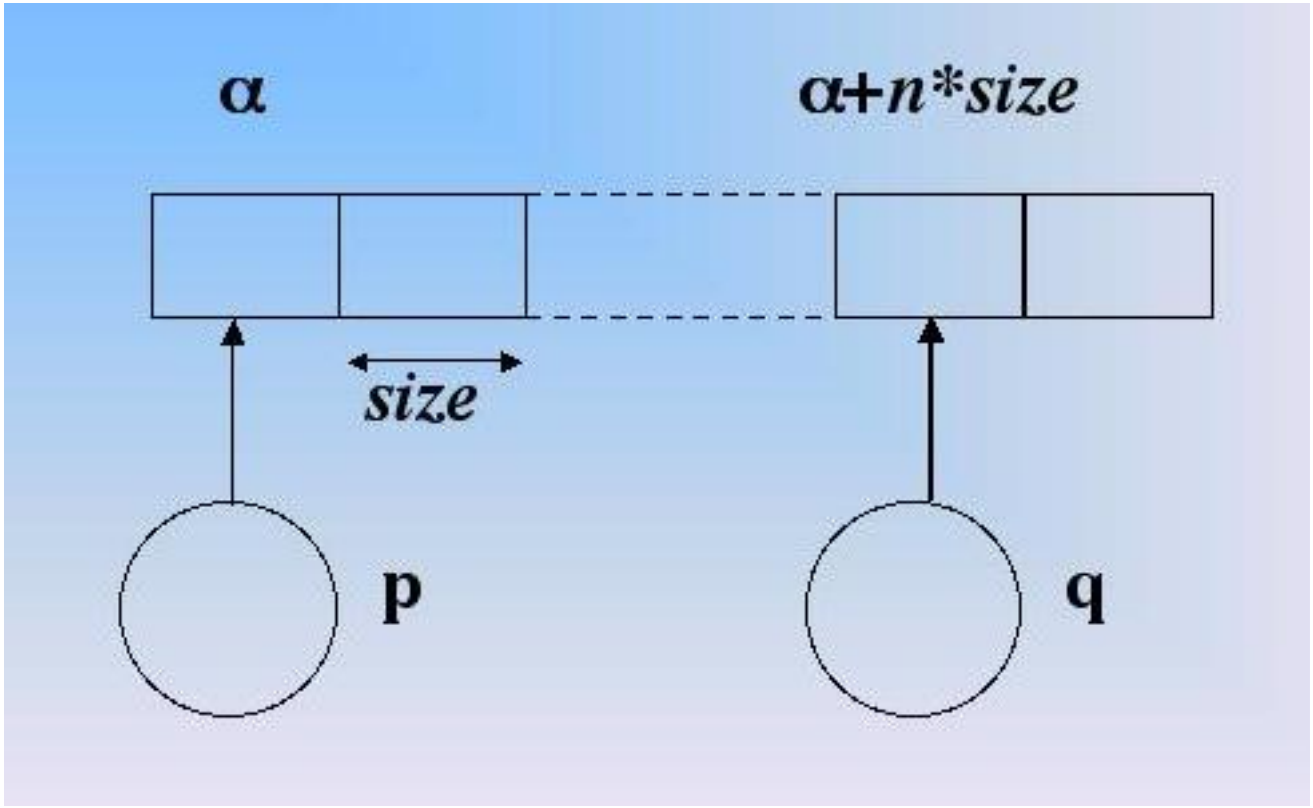
Denota un intero che rappresenta il numero di celle comprese fra p e q

Se q precede p l'intero denotato e' un numero negativo

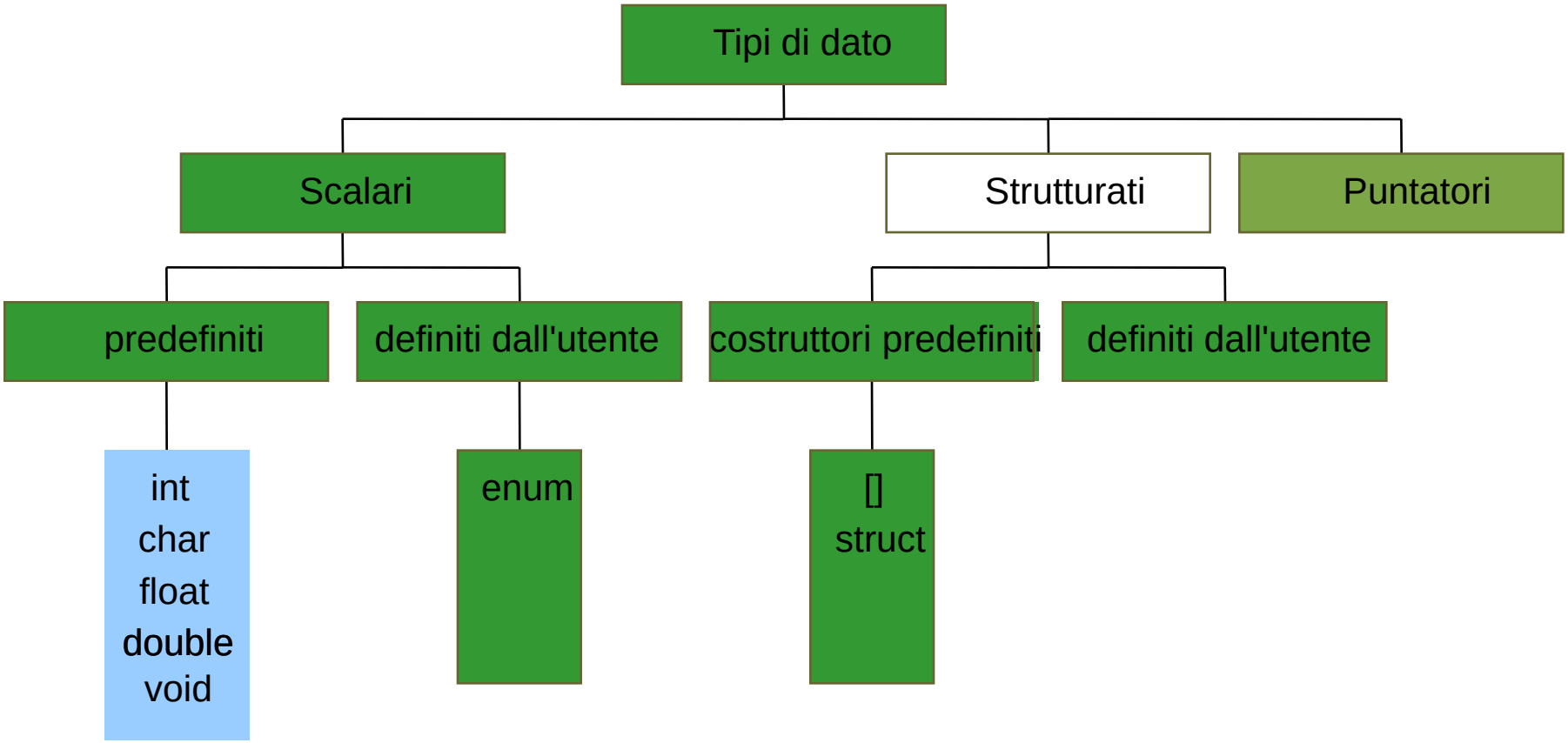
- N.b. somme di puntatori (p+q) sono illegali in quanto prive di significato

Aritmetica dei puntatori

$q = p + n$



Tipi di dato


Tipi di dato
Scalari
Strutturati
Puntatori
predefiniti
definiti dall'utente
costruttori predefiniti
definiti dall'utente
int
char
float
double
void
enum
[]
struct

Tipi di dato strutturati

- In C si possono definire due tipi di dato strutturati:
  - Array []
  - Strutture struct

Array (vettori)

- Un array è una collezione finita di valori dello stesso tipo ognuna identificata da un indice

Array:definizione

Un array è una collezione finita di valori dello stesso tipo ognuna identificata da un indice

- Definizione di una variabile di tipo array

  <tipo_base> <nome_variabile> [<costante>]

- Esempi

  int V[10];

  float A[3];

Array

- Inizializzazione
  - int v[4] = {2, 7, 9, 10}
  - int v[ ] = {2, 7, 9, 10}
- Il C non effettua controllo sui limiti degli array

Esempio

Problema

scrivere un programma che, dato un vettore di N interi, determini il valore massimo

Algoritmo

Inizialmente si assume come massimo il primo elemento

Poi si confronta via via il massimo con gli elementi del vettore: nel caso se ne trovi uno maggiore si aggiorna il valore del massimo

Al termine massimo coincide con il massimo del vettore

```
#define DIM 4
main()
{    int v[DIM] = {43, 12, 7, 86};
     int i, max=v[0];      /*inizializza massimo al primo elemento */
     for (i =1 ; i<DIM; i++)
          if (v[i]>max) max=v[i];
     printf("%d", max);
}
```

```
#define DIM 4
main()
{    int v[DIM];
     int i, max;
     for (i =0 ; i<DIM; i++) scanf("%d", &v[i]); /*leggi il vettore*/
     for (max = v[0], i =1 ; i<DIM; i++)
          if (v[i]>max) max=v[i]
     printf("%d", max);
}
```

Operatori di deferencing

- L'operatore * applicato ad una variabile puntatore accede alla variabile da esso puntata
- L'operatore **[]** applicato ad un nome di una variabile array e ad un intero accede all'i-esimo elemento dell'array
- Sono entrambi operatori di deferencing

$$V[0] \equiv *V$$

ARITMETICA DEI PUNTATORI

- Oltre a ***v ≡ v[0]**, vale anche:

***(v + 1) ≡ v [1]**

…

***(v + i) ≡ v [i]**

- Espressioni della forma **p + i** vanno sotto il nome di aritmetica dei puntatori e denotano l'indirizzo posto i celle dopo l'indirizzo denotato da p (celle, non bytes).

Conclusione

- Gli operatori * e [ ] sono intercambiabili:

  ***(v + i) ≡ v [i]**

- Ne basterebbe uno solo. Il C li fornisce entrambi solo per un fatto di comodità. Operare sui vettori scrivendo *(v + i) risulterebbe poco pratico.
- Internamente il compilatore C converte ogni espressione contenente [ ] nella corrispondente espressione con *.

ARRAY MULTIDIMENSIONALI

- E' possibile definire anche delle matrici e, piu' in generale, array a piu' dimensioni. Esempio:

int matrice [N][M]

- N indica il numero di righe (numerate da 0 ad N – 1)
- M indica il numero di colonne (numerate da 0 ad M – 1)

ARRAY MULTIDIMENSIONALI

Esempio (somma elementi di una matrice)

```
main(){
 float m[4][4] = { {1,2,3,4},
     {5,6,7,8}, {4,3,2,1}, {9,8,7,6}};
 float somma = 0;
 int i,j;
 for (i=0;i<4;i++)
  for (j=0;j<4;j++) somma += m[i][j];
}
```

Array multidimensionali

<tipo_base> <variabile> [dim1][dim2] … [dimN]

int m[10][20][30]

int x[4][2] = {1,1,2,2,3,3,4,4}

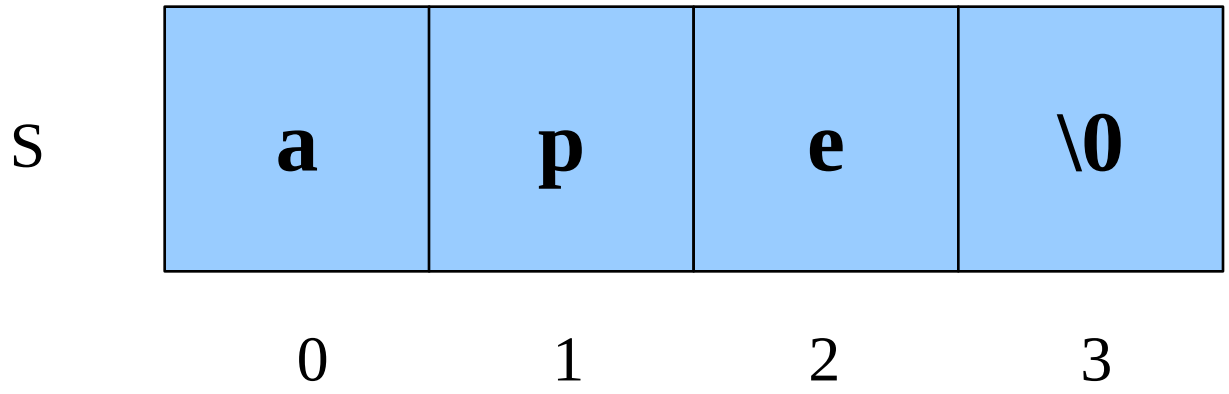
Array multidimensionali

- Perché specificare la dimensione dell'array prima?

- Un array multidimenzionale è un array di array
- Ogni elemento dell'array è in effetti un altro array
- Bisogna distinguere dove finisce una riga e inizia la riga successiva

Stringhe di caratteri

- Una stringa di caratteri in C è un array di caratteri terminato dal carattere '\0'.



- Un vettore di N caratteri può ospitare stringhe lunghe al più N-1 caratteri, perchè una cella è destinata al terminatore '\0'

Stringhe di caratteri

- Una stringa di caratteri si può inizializzare, come ogni altro array, elencando le singole componenti:

```
char s[4] = {'a', 'p', 'e', '\0'};
```

- Oppure anche, più brevemente, con la forma compatta seguente:

```
char s[4] = “ape”;
```

Il carattere di terminazione '\0' è automaticamente incluso in fondo. Quindi, ATTENZIONE ALLA LUNGHEZZA!

Data una stringa di caratteri, calcolarne la lunghezza

Ipotesi: la stringa è “ben formata”, ossia termina con ‘\0’

Specifica: scandire la stringa elemento per elemento, fino a trovare il terminatore ‘\0’

```
main ()
{    char s[] = “Stringa di prova”;
int lung=0;
for ( ; s[lung] != ‘\0’; lung++);
printf (“la lunghezza della stringa e’%d”, lung);
}
```

Data una stringa di caratteri, copiarla in un altro array di caratteri (di lunghezza non inferiore)

Ipotesi: la stringa è “ben formata”, ossia termina con ‘\0’

Specifica: scandire la stringa elemento per elemento, fino a trovare il terminatore ‘\0’ copiando l’elemento nella posizione corrispondente dell’altro array

```
main ()
{    char s[] = ”Stringa di prova”;
char s2[40];
int i;
for (i=0; s[i] != ‘\0’; i++) s2[i] = s[i];
s2[i]= ‘\0’;
}
```

Data una stringa di caratteri, copiarla in un altro array con eventuale troncamento se la stringa e piu' lunga

Ipotesi: la stringa è "ben formata", ossia termina con '\0'

Specifica: scandire la stringa elemento per elemento, fino a trovare il terminatore '\0' o fino alla lunghezza dell'array di destinazione

```
#define N 6

main ()
{    char s[] = "Stringa di prova";
char s2[N];
int i;
for (i = 0; s[i] != '\0' && i<N-1; i++) s2[i] = s[i];
s2[i]= '\0';
}
```

Data due stringhe di caratteri decidere quale precede l'altra

Poiché si possono avere tre risultati utilizziamo un intero:
negativo se s1 precede s2
positivo se s2 precede s1
zero se s1 e uguale a s2

```
main ()
{    char s1[] = ”Prima stringa”;
char s2[] = ”Seconda stringa”;
int stato,i;
for ( i=0; s1[i] != ‘\0’ && s2[i] != ‘\0’ &&
          s1[i] == s2[i]; i++);
stato= s1[i]-s2[i];
}
```

Stringhe di caratteri

- Nell'esempio della copiatura:

  Data una stringa di caratteri, copiarla in un altro array di caratteri (di lunghezza non inferiore).

si è deciso di copiare la stringa nell'array “carattere per carattere”

Avremmo potuto fare diversamente?

Perchè non copiarla tutta in un colpo?

Stringhe di caratteri

- Perchè non fare così?

```
int main (){
 char s[ ] = "nel mezzo del cammin ...”;
 char s2 [40];
    s2 = s;
}
```

NON FUNZIONA!

Libreria sulle stringhe

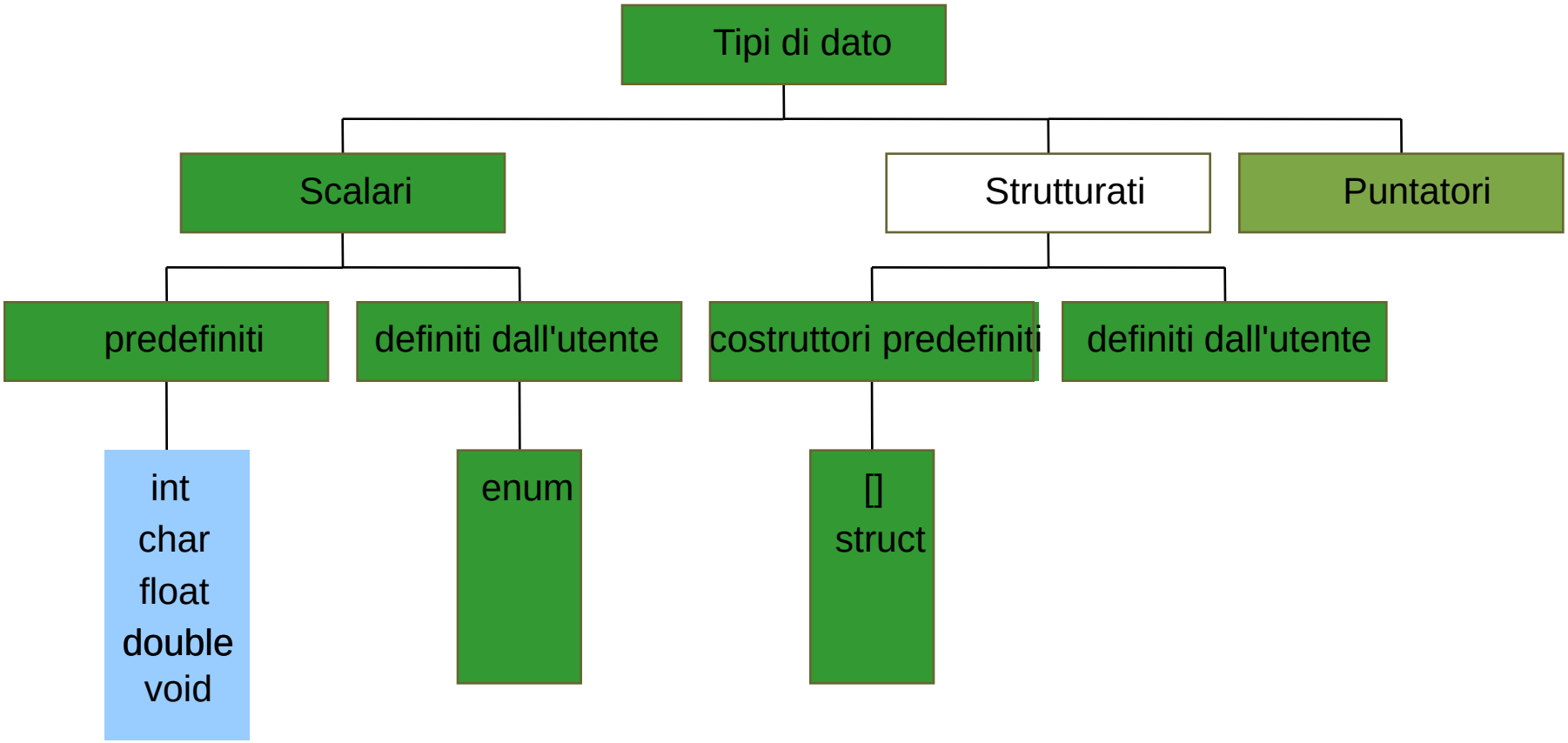
Il C fornisce una libreria per operare sulle stringhe:

```
#include <string.h>
```

Include funzioni per:

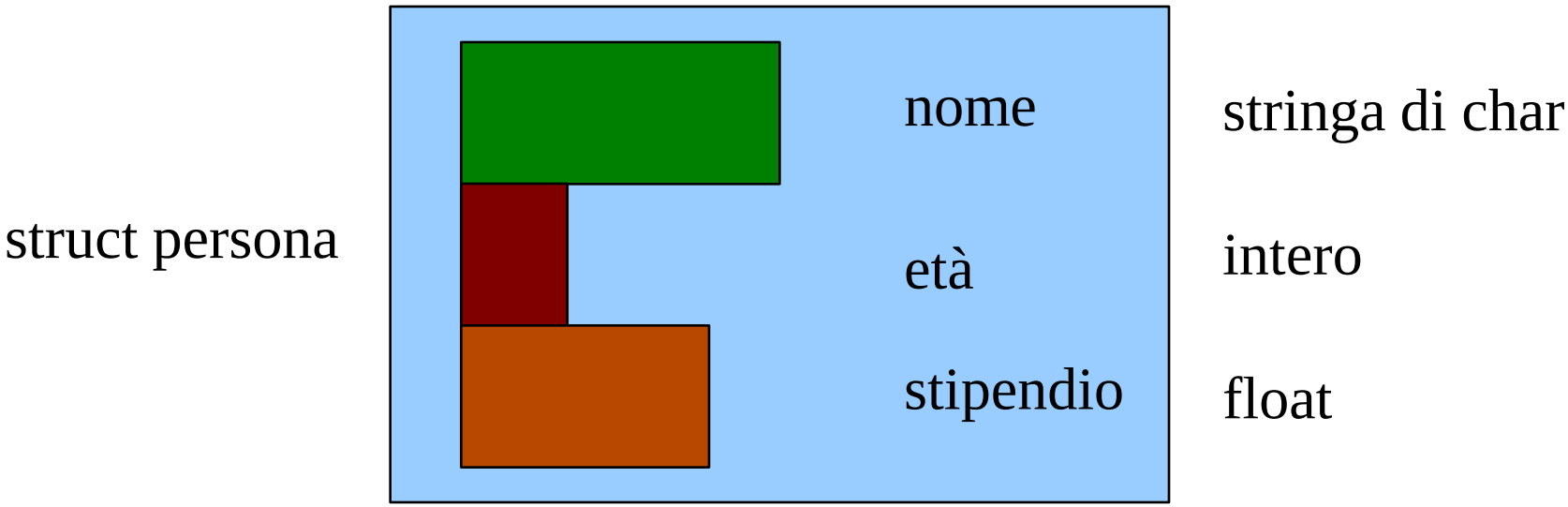
- Copiare una stringa in un'altra (strcpy)
- Concatenare due stringhe (strcat)
- Confrontare due stringhe (strcmp)
- Cercare un carattere in una stringa (strchr)
- Cercare una stringa in un'altra (strstr)
- ...

Tipi di dato


Tipi di dato
Scalari
Strutturati
Puntatori
predefiniti
definiti dall'utente
costruttori predefiniti
definiti dall'utente
int
char
float
double
void
enum
[]
struct

Strutture

Una struttura è una collezione finita di variabili non necessariamente dello stesso tipo ognuna identificata da un nome.



STRUTTURE

Una *struttura* è una *collezione finita di variabili non necessariamente dello stesso tipo*, ognuna identificata da un *nome*.

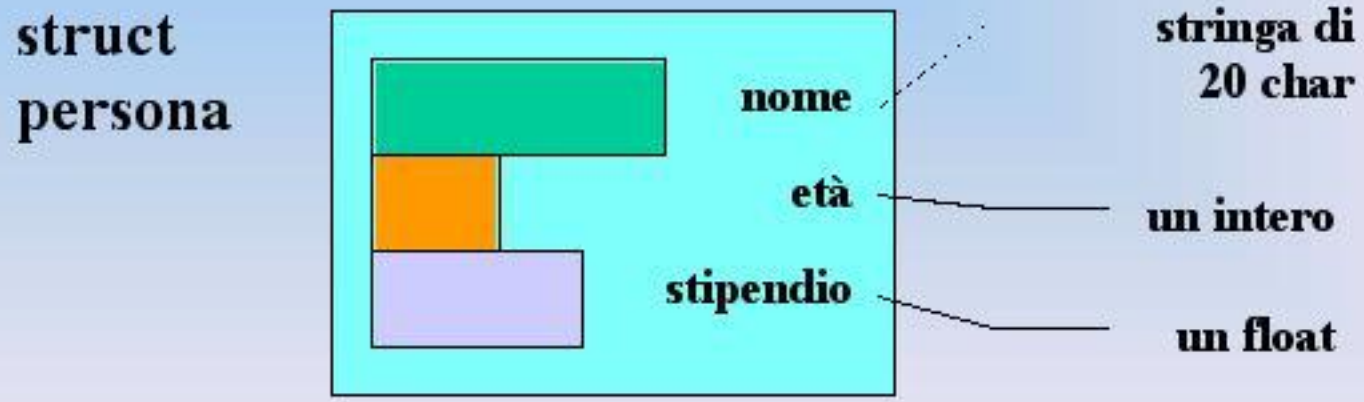
Definizione di una *variabile* di tipo *struttura*:

```
struct [<etichetta>] {
    { <definizione-di-variabile> }
} <nomeStruttura> ;
```

STRUTTURE - ESEMPIO

```
struct persona {
    char nome[20];
    int  eta;
    float stipendio;
} pers ;
```

Definisce una variabile `pers` strutturata nel modo illustrato.



ESEMPI

```
struct punto {
   int  x, y;
} p1, p2 ;
```

p1 e p2 sono fatte ciascuna da due interi di nome x e y

```
struct data {
   int  giorno,mese,anno;
} d ;
```

d è fatta da tre interi di nome giorno, mese e anno

STRUTTURE

Una volta definita una variabile struttura, *si accede ai singoli campi mediante la notazione puntata.*

Ad esempio:

```
p1.x = 10;   p1.y = 20;
p2.x = -1;   p2.y = 12;
d.giorno = 25;
d.mese = 12;
d.anno = 1999;
```

Ogni campo si comporta e si usa come una normale variabile.

UN ALTRO ESEMPIO

```
main(){
   struct frutto {
      char nome[20]; int  peso;
   } f1;
   struct frutto f2 ;

   ...
}
```

Non occorre ripetere l'elenco dei campi perché è *implicito nell'etichetta* `frutto`, che è già comparsa sopra.

ESEMPIO

```
main(){
  struct frutto {
     char nome[20]; int  peso;
  } f1 = {"mela", 70};
  struct frutto f2 = {"arancio", 50};

  int peso = f1.peso + f2.peso;

}
```

Non c'è alcuna ambiguità perché ogni variabile di nome `peso` è definita nel proprio environment.

UNA PRECISAZIONE

A differenza di quanto accade con gli array, ***il nome della struttura rappresenta la struttura nel suo complesso.***

Quindi, è possibile:

- ***assegnare una struttura a un'altra*** **(f2 = f1)**
- ***che una funzione restituisca una struttura***

E soprattutto:

- ***passare una struttura come parametro a una funzione significa passare una copia***

ASSEGNAMENTO FRA STRUTTURE

```
main(){
   struct frutto {
      char nome[20]; int  peso;
   } f1 = {"mela", 70};
   struct frutto f2 = {"arancio", 50};

   f1 = f2;
}
```

Equivale a copiare `f2.peso` in `f1.peso`, e `f2.nome` in `f1.nome`.

RIFLESSIONE

Se una struttura, anche molto voluminosa, viene copiata elemento per elemento...

.. perché non usare una struttura per incapsulare un array?

In effetti:

- il C non rifiuta di manipolare gli array come un tutt'uno "per principio": è solo la conseguenza del modo in cui si interpreta il loro nome
- **quindi, *"chiudendoli in una struttura" dovremmo riuscirci!***

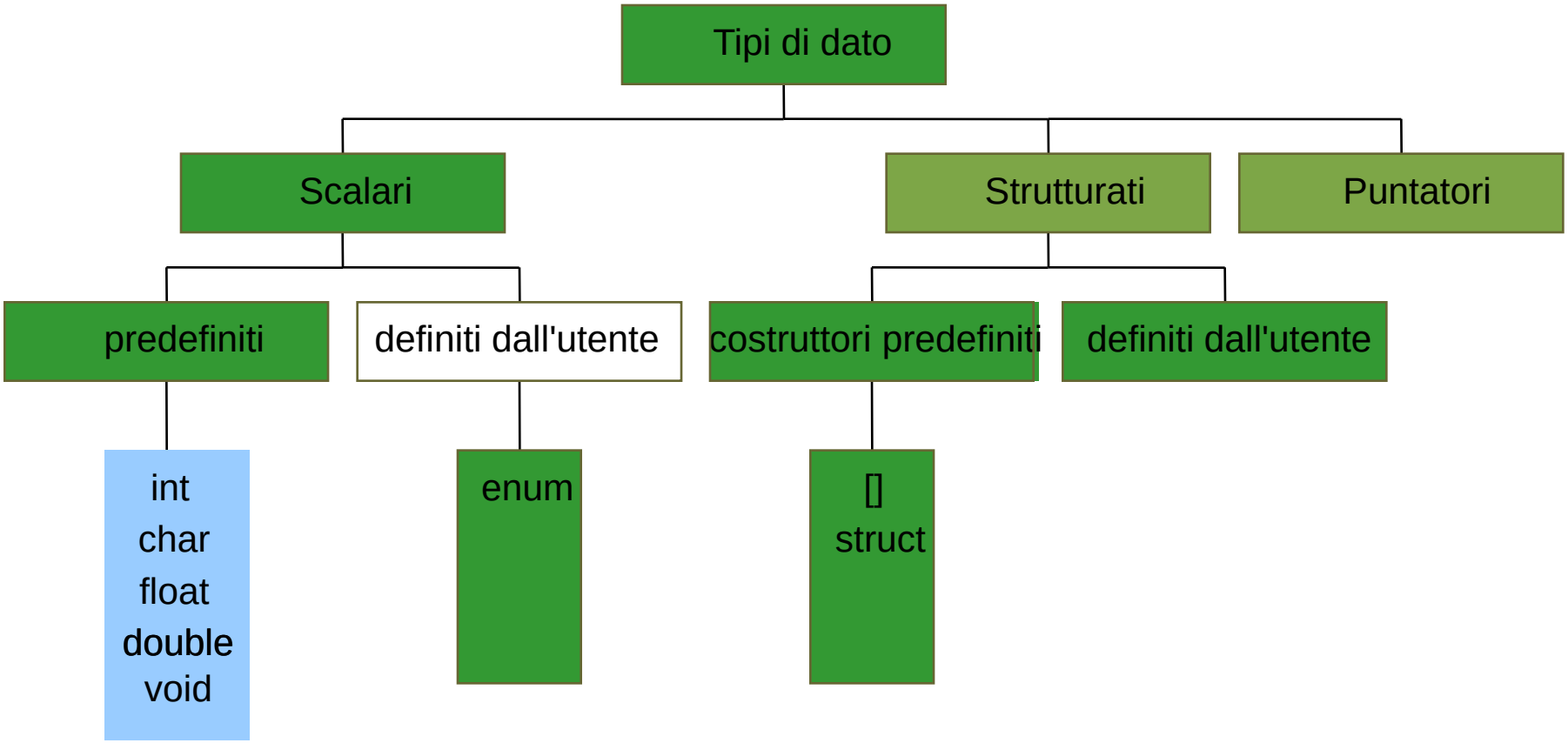
E INFATTI...

```
main(){
  struct string20 {
     char s[20];
  } s1 = {"Paolino Paperino" },
    s2 = {"Gastone Fortunato" };

  s1 = s2;    /* FUNZIONA!! */

}
```

Tipi di dato


Tipi di dato
Scalari
Strutturati
Puntatori
predefiniti
definiti dall'utente
costruttori predefiniti
definiti dall'utente
int
char
float
double
void
enum
[]
struct

TIPI ENUMERATIVI

- **Un tipo enumerativo viene specificato tramite *l'elenco dei valori* che i dati di quel tipo possono assumere.**
- **Schema generale:**

```
typedef enum {
   a1, a2, a3, ... , aN } EnumType;
```

- **Il compilatore associa a ciascun "identificativo di valore" a1,..aN un *numero naturale* (0,1,...)**, che viene usato nella valutazione di espressioni che coinvolgono il nuovo tipo.

TIPI ENUMERATIVI

- **Gli "identificativi di valore" `a1,... , aN` sono a tutti gli effetti *delle nuove costanti*.**
- **Esempi**

```
typedef enum {
  lu, ma, me, gi, ve, sa, do } Giorni;
typedef enum {
  cuori, picche, quadri, fiori } Carte;

Carte   C1, C2, C3, C4, C5;
Giorni  Giorno;
if (Giorno == do) /* giorno festivo */
else /* giorno feriale */
```

TIPI ENUMERATIVI - NOTE

- **Un tipo enumerativo è *totalmente ordinato:* vale l'ordine con cui gli identificativi divalore sono stati elencati nella definizione.**

- **Esempio**

```
typedef enum {
   lu, ma, me, gi, ve, sa, do } Giorni;
```

Data la definizione sopra,

`lu < ma` *è vera*

`lu >= sa` *è falsa*

in quanto `lu` ↔ 0, `ma` ↔ 1, `me` ↔ 2, etc.

TIPI ENUMERATIVI - NOTE

- **Poiché un tipo enumerativo è, *per la macchina C,* indistinguibile da un intero, è possibile in linea di principio *mischiare interi e tipi enumerativi***

- **Esempio**

```
typedef enum {
   lu, ma, me, gi, ve, sa, do } Giorni;
Giorni g;
g = 5;          /* equivale a g = sa */
```

- ***È una pratica da evitare ovunque possibile!***

TIPI ENUMERATIVI - NOTE

- **Poiché un tipo enumerativo è, *per la macchina C,* indistinguibile da un intero, è possibile in linea di principio *mischiare interi e tipi enumerativi***

- **Esempio**

```
typedef enum {lu, ma, me, gi, ve, sa, do} Giorni;
Giorni g;
g = 5;          /* equivale a g = sa */
```

Non c'è alcun controllo sugli estremi:
g = 24 verrebbe accettato...
Ma con quali conseguenze??

- ***È una pratica <u>da evitare</u> ovunque possibile!***

TIPI ENUMERATIVI - NOTE

- **È anche possibile specificare i valorinaturali cui associare i simboli** `a1, . . . , aN`
- qui, `lu` ↔ 0, `ma` ↔ 1, `me` ↔ 2, etc.:

```
typedef enum {
   lu, ma, me, gi, ve, sa, do } Giorni;
```

- qui, invece, `lu` ↔ 1, `ma` ↔ 2, `me` ↔ 3, etc.:

```
typedef enum {
   lu=1, ma, me, gi, ve, sa, do } Giorni;
```

- qui, infine, l'associazione è data caso per caso

```
typedef enum { lu=1, ma, me=7, gi, ve,
   sa, do} Giorni;
```

TIPI ENUMERATIVI - NOTE

- **È anche possibile specificare i valorinaturali cui associare i simboli** `a1, . . . , aN`
- qui, `lu` ↔ 0, `ma` ↔ 1, `me` ↔ 2, etc.:

```
typedef enum {
   lu, ma, me, gi, ve, sa, do } Giorni;
```

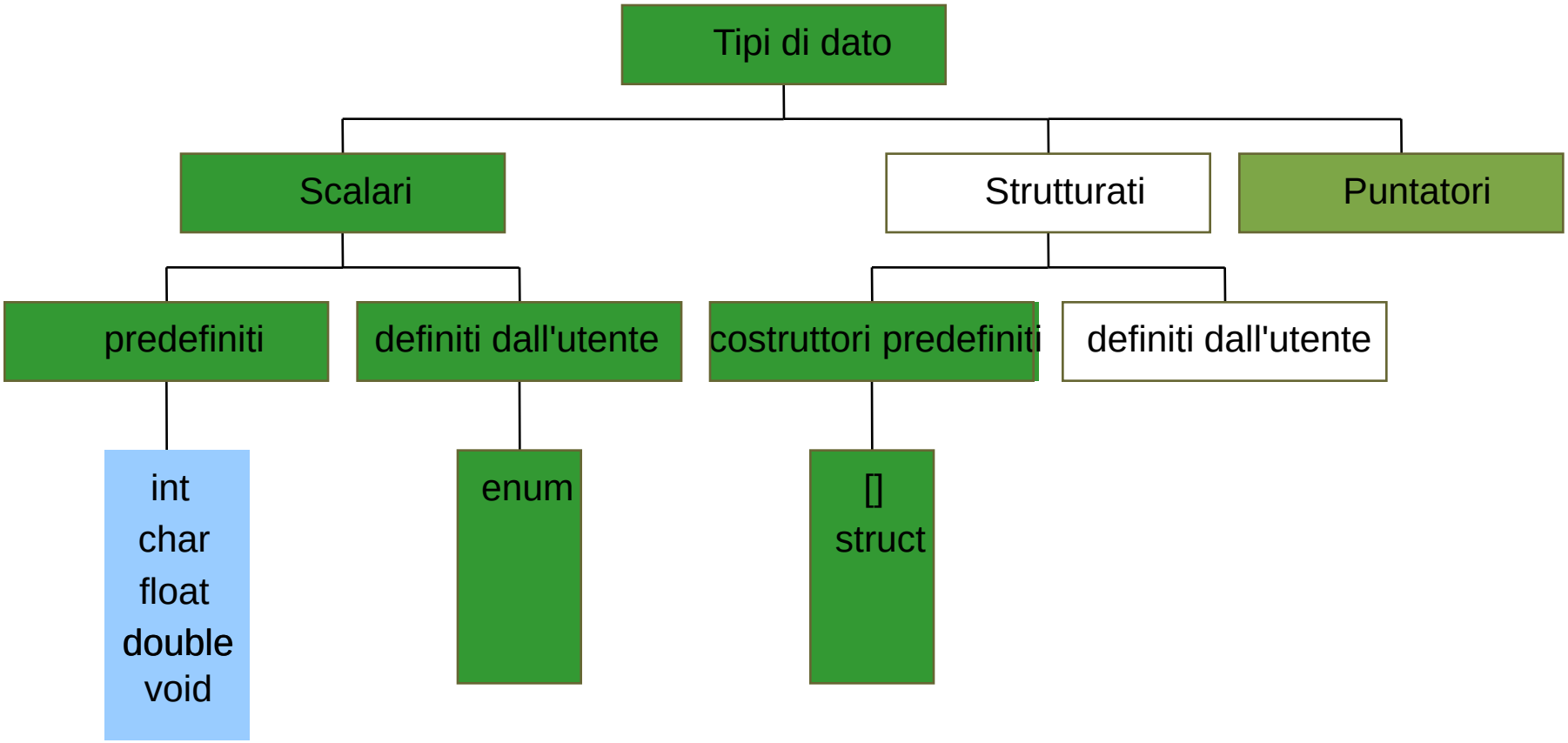
- qui, invece, `lu` ↔ 1, `ma` ↔ 2, `me` ↔ 3, etc.:

```
typedef enum {
   lu=1, ma, me, gi, ve, sa, do } Giorni;
```

- qui, infine, l'associazione è data caso per caso

```
typedef enum { lu=1, ma, me=7, gi, ve,
   sa, do} Giorni;
```

Tipi di dato


Tipi di dato
Scalari
Strutturati
Puntatori
predefiniti
definiti dall'utente
costruttori predefiniti
definiti dall'utente
int
char
float
double
void
enum
[]
struct

DEFINIZIONE DI NUOVI TIPI

Perché definire nuovi tipi è importante?

- **Permette di progettare *al nostro livello di astrazione*, *non a quello della macchina C***
  - Il mondo reale è fatto di *giorni, temperature, colori, stringhe...* **non di int, char, etc!!**
  - operare su "stringhe", "matrici", "vettori" è ben diverso che operare su char x[20], etc
- **Consente di *prescindere dalla rappresenta-azione concreta delle cose***
  - temperature, colori, interi... saranno anche tutti int, alla fine... **ma concettualmente sono diversi!**

TIPI DI DATO ASTRATTO IN C

In C, un *ADT* si costruisce definendo:

- ***il nuovo tipo con `typedef`***
- ***una funzione per ogni operazione***

Esempio: il contatore

- una entità caratterizzata da un valore intero

```
typedef int counter;
```

- con operazioni per
  - resettare il contatore a zero `reset(counter*);`
  - incrementare il contatore `inc(counter*);`