Programmazione B
Ingegneria e Scienze Informatiche - Cesena
A.A. 2021-2022

Tipi di dati avanzati: ENUM, STRUCT, UNION

Catia Prandi - catia.prandi2@unibo.it

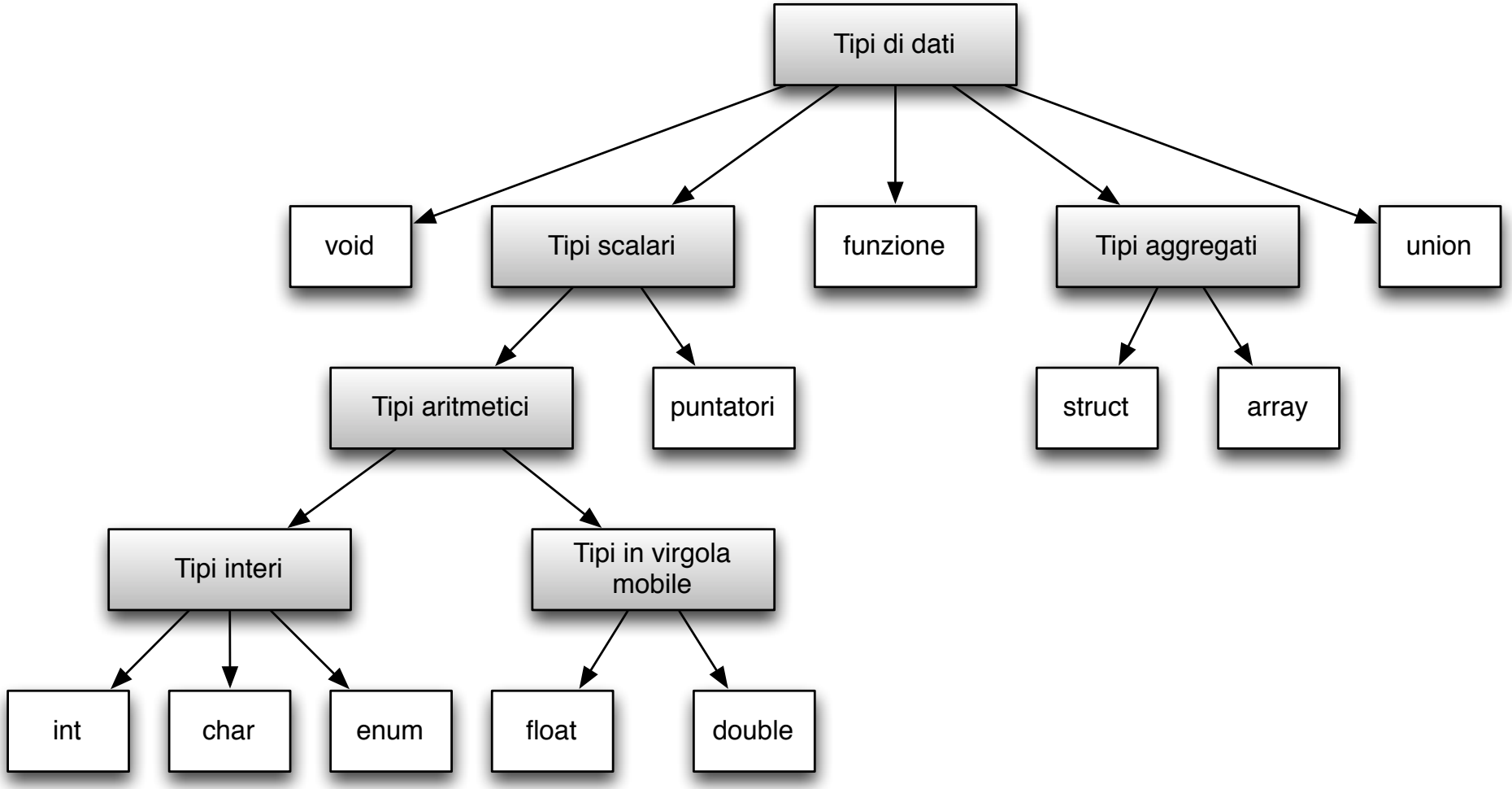
Credit: Pietro Di Lena

```c
// The real mutually exclusive bool type
union bool {
  unsigned int false : 1;
  unsigned int true  : 1;
};
```

Introduzione

- Abbiamo visto finora solo i tipi di dati base (o primitivi) del linguaggio C.
- Sappiamo dichiarare variabili e funzioni che facciano uso di tali tipi di dato.
- Vediamo le regole sintattiche del linguaggio C che ci permettono di *costruire* tipi di dati *complessi* (o avanzati) a partire dai tipi di dati base.
- Parleremo in particolare di **enumerazioni**, **strutture**, **unioni**.
- Affronteremo anche i seguenti argomenti:
  - Come possiamo dichiarare un *nuovo* nome per un tipo di dato?
  - Come e quando possiamo riutilizzare gli identificatori in un programma C?

Classificazione dei tipi di dati in C



Il tipo enum

- Il **tipo enumerativo** è un tipo di dato che consiste di un insieme ristretto di valori.
- La caratteristica principale dei tipi enumerativi è che ogni elemento appartenente al tipo ha un proprio nome (identificatore).
- I nomi associati agli elementi del tipo sono trattati come valori costanti, chiamati **costanti enumerative**.
- Un esempio di tipo enumerativo potrebbe essere il set di *semi* (`seed`) di un mazzo di carte. Abbiamo a disposizione solo quattro semi distinti, che possiamo *identificare* con quattro nomi differenti (`clubs`, `diamonds`, `hearts`, `spades`). Una variabile di tipo enumerativo `seed` potrà assumere uno di questi quattro valori.
- I tipi enumerativi sono stati aggiunti al linguaggio C dallo standard ANSI.
- Il tipo `enum` deve essere sufficientemente grande da poter contenere tutti gli `int`.
  - La dimensione specifica dipende dall'implementazione.
- In C le enumerazioni hanno come unica funzione quella di rendere più leggibile il codice (*syntactic sugar*): possiamo *simulare* le costanti enumerative con le macro.

Dichiarazione di tipi enum

- Per poter dichiarare un tipo di dato enumerativo facciamo uso della parola chiave `enum` seguita da una lista di **costanti enumerative** (identificatori).

```
enum [Tag] {
    <Nome1> [ = ConstExpr1],
    <Nome2> [ = ConstExpr2],
    ...
    <NomeN> [ = ConstExprN]
};
```

- Per convenzione, gli identificatori delle costanti enumerative sono in maiuscolo.
- Il nome del tipo enumerativo (**tag**) è opzionale.

```c
#include<stdio.h>
// Tipo booleano
enum bool {FALSE, TRUE};

int main() {
 printf("%d %d\n",FALSE,TRUE);
 return 0;
}
```

```c
#include<stdio.h>
// Tipo booleano anonimo
enum {FALSE, TRUE};

int main() {
 printf("%d %d\n",FALSE,TRUE);
 return 0;
}
```

In entrambe le definizioni, abbiamo i due tipi enumerativi `FALSE`, `TRUE` (tipi di dato intero) che possiamo utilizzare nel loro ambito di visibilità. Nel primo caso, definiamo anche il nome del tipo di dato enumerativo `bool`. Nel secondo caso, il nome del tipo di dato è anonimo ma possiamo comunque utilizzare gli identificatori dichiarati.

Dichiarazione di tipi enum: valore delle costanti enumerative 1/2

- Ogni costante enumerativa può assumere **solo** valori di tipo `int`.
- Se non specificato, di *default* il primo elemento nella lista ha valore 0 e gli elementi successivi sono associati con il valore dell'elemento precedente incrementato di 1.

```
1 enum days {
2   MON, // 0
3   TUE, // 1
4   WED, // 2
5   THU, // 3
6   FRI, // 4
7   SAT, // 5
8   SUN  // 6
9 };
```

- Possiamo modificare i valori di default esplicitando un assegnamento con un'**espressione costante di tipo intero**. Le costanti enumerative che seguono l'assegnamento saranno inizializzate incrementando di uno il valore della costante precedente.

```
1 enum days {
2   MON = -6, // -6
3   TUE,      // -5
4   WED,      // -4
5   THU,      // -3
6   FRI,      // -2
7   SAT,      // -1
8   SUN       //  0
9 };
```

Dichiarazione di tipi enum: valore delle costanti enumerative 2/2

- E' possibile avere costanti enumerative con lo stesso valore.

```
1 // Tipo bool inutile
2 enum bool {
3   FALSE = 1, // 1
4   TRUE  = 1  // 1
5 };
```

- Ogni assegnamento esplicito viene utilizzato come punto di partenza per gli assegnamenti successivi.

```
1 enum days {
2   MON = 3, // 3
3   TUE,     // 4
4   WED,     // 5
5   THU,     // 6
6   FRI = 0, // 0
7   SAT,     // 1
8   SUN      // 2
9 };
```

Dichiarazione di tipi enum: ulteriori regole sintattiche

- Gli identificatori in una lista di enumerazioni devono essere distinti da altri identificatori di enumerazioni nello stesso scope, anche se assumono lo stesso valore.

```
1 enum bool { FALSE, TRUE };
2
3 enum extended_bool {
4   UNKNOWN = -1,
5   FALSE, // Errore: duplicazione
6   TRUE   // Errore: duplicazione
7 };
```

- Gli elementi enumerativi devono essere separati da una virgola. Non è un errore di sintassi specificare una virgola dopo l'ultimo elemento.

```
1 enum bool {
2   FALSE // Errore
3   TRUE
4 };
```

```
1 enum bool {
2   FALSE,
3   TRUE, // OK
4 };
```

- Gli assegnamenti possono essere solo **espressioni costanti intere**.

```
1 const int n=1; // Variabile read-only
2 enum extended_bool {
3   UNKNOWN = -1.0,    // Errore
4   FALSE   = 'A'-65, // OK
5   TRUE    =  n       // Errore
6 };
```

Dichiarazione di variabili di tipo enum

- La dichiarazione di un tipo enum *dichiara* un nuovo tipo di dato (non riserva memoria).
- Una dichiarazione di variabile di tipo enum (riserva memoria) deve essere preceduta, come di consueto, dal nome tipo enumerativo.

```
1 enum bool { FALSE, TRUE }; /*definizione */
2
3 enum bool flag; /* dichiarazione */
```

Il tipo della variabile `flag` è `enum bool`.

- In forma contratta, possiamo far precedere il nome di una variabile dalla dichiarazione del tipo enumerativo.

```
1 enum bool { FALSE, TRUE } flag1, flag2;
```

Il tipo delle variabili `flag1, flag2` è `enum bool`.

- Se il tag del tipo enumerativo è omesso (tipo anonimo) non possiamo dichiarare successivamente variabili di tale tipo enumerativo.

```
1 enum { FALSE, TRUE } flag1, flag2;
2
3 enum flag3; // Definizione di tipo enum "vuoto"
```

L'identificatore `flag3` non è il nome di una variabile ma di un tipo enumerativo *vuoto*. Tale dichiarazione è sintatticamente valida ma inutile.

Operazioni con variabili di tipo enum

- In C, le variabili di tipo enum sono effettivamente variabili di tipo *intero*.
- Possiamo quindi assegnare ad una variabile di tipo enum qualsiasi valore di tipo intero, non solo quelli specificati nella enumerazione.

```
1 enum bool { FALSE, TRUE } flag1 = 10, flag2 = -1;
```

- Possiamo utilizzare le variabili di tipo enum con tutti gli operatori applicabili con i tipi di dato interi.

```
1 enum bool { FALSE, TRUE } flag1, flag2, flag3;
2
3 flag1 = FALSE+1;           // 1
4 flag2 = flag1*TRUE + 10;   // 11
5 flag3 = flag1/flag2;       // 0 (divisione tra int)
```

- Possiamo verificare la dimensione di un tipo enumerativo con l'operatore sizeof.

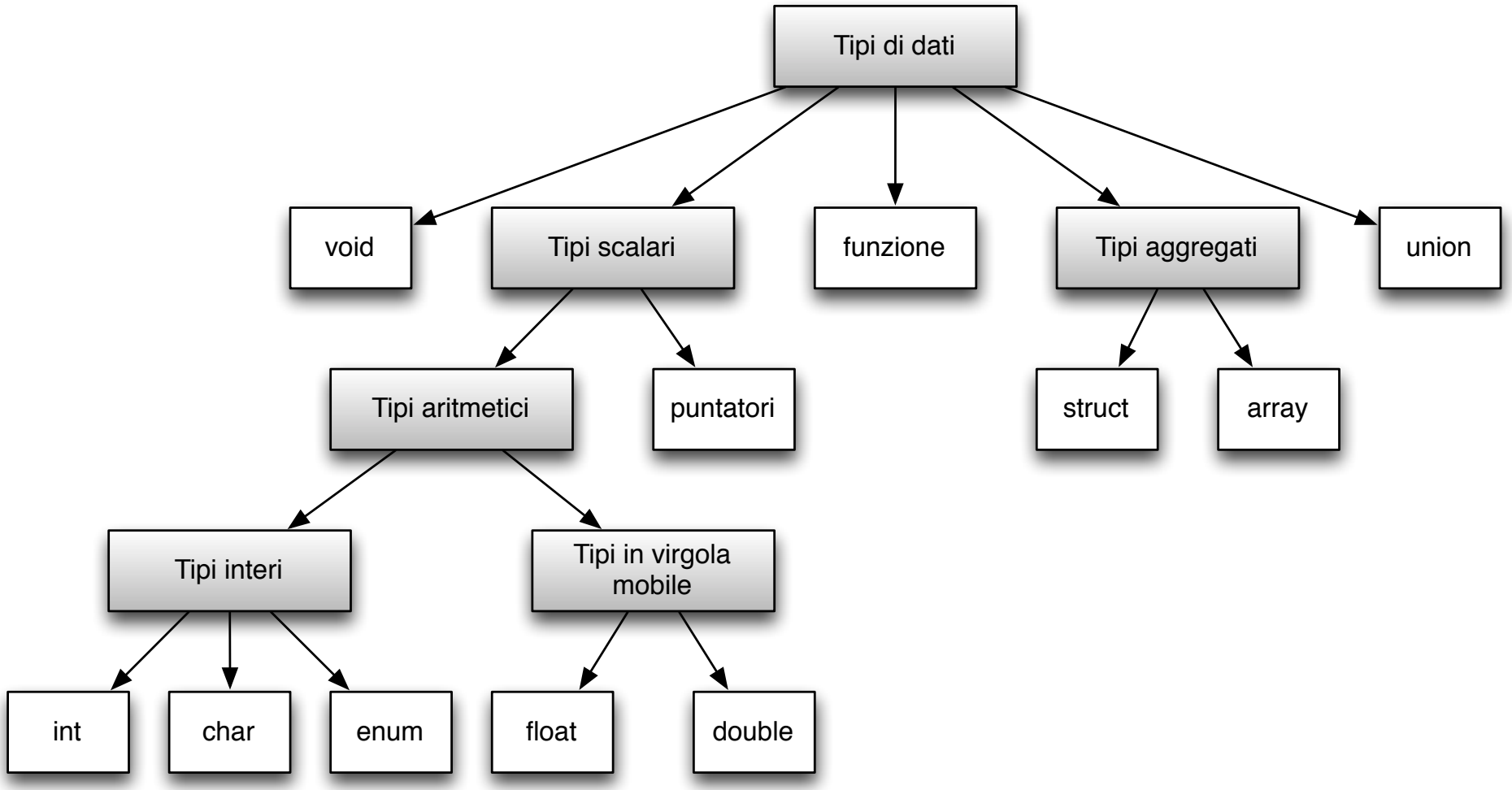
```
1 #include <stdio.h>
2 enum bool {FALSE, TRUE};
3 enum seed {CLUB,DIAMOND,HEART,SPADE};
4
5 int main() {
6  printf("%lu %lu\n",sizeof(enum bool),sizeof(enum seed));
7  return 0;
8 }
```

I tipi enum bool ed enum seed hanno la stessa dimensione, che dipende dall'implementazione.

Approfondimento `sizeof()`

- `sizeof` è un operatore che restituisce come int il numero di byte occupato dal tipo di dato o dalla variabile indicati tra le parentesi.
- Si noti che `sizeof()` non è una funzione, ma un operatore: esso è dunque intrinseco al compilatore e non fa parte di alcuna libreria. Inoltre esso restituisce sempre un valore di tipo int, indipendentemente dal tipo di dato o di variabile specificato tra le parentesi.

Classificazione dei tipi di dati in C



Il tipo struct

- Le *strutture* sono tipi di dato che permettono di *raggruppare* una o più variabili.
- Ci permettono di raggruppare variabili di uno o più tipi differenti.
- Ci permettono di avere un nome unico per referenziare gruppi di variabili differenti.
- Le strutture aiutano ad organizzare dati complessi poiché permettono di trattare come un'entità unica insiemi di variabili logicamente correlate tra di loro.
- Tutti i linguaggi ad alto livello mettono a disposizione costrutti sintattici per poter organizzare gruppi di variabili in strutture.
- In alcuni linguaggi le strutture sono chiamate *record*.
- Nel linguaggio C, possiamo definire strutture utilizzando la parola chiave `struct`.

Dichiarazione di tipi struct

- In C una dichiarazione di struttura è definita com una collezione di dichiarazioni di variabili (chiamate *campi* o *membri*), racchiusa tra parentesi graffe e preceduta dalla parola chiave `struct`.
- Ogni campo deve avere un **tipo** e un **nome** (identificatore):

```
struct [Tag] {
  <Tipo1> <Nome1>;
  <Tipo2> <Nome2>;
  ...
  <TipoN> <NomeN>;
};
```

- Il nome della struttura (**tag**) è opzionale.

```c
// Struttura con nome
struct point {
  int x; // x-axis coord
  int y; // y-axis coord
};
```

```c
// Struttura anonima
struct {
  int x; // x-axis coord
  int y; // y-axis coord
};
```

- Come per i tipi `enum`, la dichiarazione non riserva memoria, ma dichiara semplicemente l'esistenza di un nuovo tipo di dato.
- Se dichiariamo il `nome` (tag) della struttura, successivamente potremo fare riferimento a tale tipo di dato indicando `struct tag`.

Dichiarazione di variabili di tipo struct

- Possiamo dichiarare variabili di tipo `struct`, utilizzando la sintassi consueta: il nome della variabile deve essere preceduto dal tipo della variabile.

```c
// Struttura con nome
struct point {
  int x; // x-axis coord
  int y; // y-axis coord
} coord1, coord2;
```

```c
// Struttura anonima
struct {
  int x; // x-axis coord
  int y; // y-axis coord
} coord1, coord2;
```

In entrambi gli esempi, il tipo delle variabili `coord1, coord2` è una struttura contenente due interi.

- Se abbiamo specificato un nome per la struttura, possiamo *riutilizzare* il tipo di dato per dichiarazioni successive.

```c
// Struttura con nome
struct point {
  int x; // x-axis coord
  int y; // y-axis coord
};

// Variabili di tipo struct point
struct point coord1, coord2;
```

```c
// Struttura anonima
struct {
  int x; // x-axis coord
  int y; // y-axis coord
} coord1; // Variabile di tipo struct

// Struttura vuota con nome coord2.
struct coord2;
```

- La dichiarazione di una variabile di tipo `struct` riserva memoria per la variabile.

Strutture annidate

- E' possibile annidare strutture all'interno di strutture.

```c
// Strutture con nome
struct point {
  int x; // x-axis coord
  int y; // y-axis coord
};


struct line {
  struct point coord1;
  struct point coord2;
} line1, line2;
```

```c
// Strutture anonime
struct {
  struct {
    int x; // x-axis coord
    int y; // y-axis coord
  } coord1;
  struct {
    int x; // x-axis coord
    int y; // y-axis coord
  } coord2;
} line1, line2;
```

In entrambi gli esempi, il tipo delle variabili `line1, line2` è una struttura contenete due campi `coord1, coord2` il cui tipo è, a loro volta, una struttura contenente due interi.

- Limiti dello standard ISO C89:
  - massimo 15 i livelli di annidamento in una singola dichiarazione di struttura;
  - massimo 127 membri in una singola dichiarazione di struttura.

Spazio di allineamento dei tipi di dati struct

- In memoria i campi di una struttura sono allocati in modo contiguo e nello stesso ordine definito dalla dichiarazione.
- A seconda del calcolatore, tra un campo e il successivo possono essere inseriti degli spazi di memoria di **allineamento**. Tali **bit di padding** non possono essere utilizzati.
- Lo spazio di allineamento serve per rendere la dimensione di memoria della struttura un multiplo di byte, che dipende dalle caratteristiche del processore.
- Possiamo verificare la dimensione in byte di una struttura con l'operatore `sizeof`.

```c
#include <stdio.h>

struct s {
 char x;
 int  y;
};

int main() {
 printf("sizeof(char) = %lu\n",sizeof(char));
 printf("sizeof(int)  = %lu\n",sizeof(int));
 printf("sizeof(s)    = %lu\n",sizeof(struct s));
 return 0;
}
```

Tipicamente, `sizeof(struct s) > sizeof(char)+sizeof(int)`.

Inizializzazione dei tipi di dati struct

- E' possibile inizializzare una variabile di tipo `struct` facendo seguire alla propria dichiarazione un'istruzione di assegnamento con:
  - una lista di valori (**espressioni costanti**) tra parentesi graffe;
  - una variabile dello stesso tipo.
- E' possibile specificare anche solo alcuni valori di inizializzazione. In tal caso, i restanti campi sono inizializzati automaticamente a zero.

```
1  struct point {
2    int x;
3    int y;
4  };
5
6  struct line {
7    struct point coord1;
8    struct point coord2;
9  };
10
11 struct point p1 = {1,1};           // Contiene coord (1,1)
12 struct point p2 = {1};             // Contiene coord (1,0)
13 struct point p3 = p1;              // Contiene coord (1,1)
14 struct line  l1 = {{0,0}, {1,1}}; // Contiene coord (0,0) e (1,1)
15 struct line  l2 = {1}; // Contiene coord (1,0) e (0,0) - solo coord1
16 struct line  l2 = {1,0}; // Contiene coord (1,0) e (0,0) - solo coord1
17 struct line  l3 = l2;              // Contiene coord (1,0) e (0,0)
```

printf("(%d,%d) e (%d,%d)", l3.coord1.x, l3.coord1.y, l3.coord2.x, l3.coord2.x);
/* (1,0) (0,0) */

Inizializzazione dei tipi di dati struct: esempi con errori

- Non possiamo inizializzare i membri di una struttura nella sua dichiarazione.

```
1 struct point {
2   int x = 0; // Errore di sintassi
3   int y = 0; // Errore di sintassi
4 };
```

- Possiamo inizializzare con lista di valori una variabile di tipo struct solo al momento della sua definizione.

```
1 struct point {int x; int y;};
2
3 struct point p1 = {0,0}; // OK
4 struct point p2;
5 p2 = {0,0};              // Errore di sintassi
```

- Strutture (anche anonime) dichiarate con gli stessi campi, sono tipi di dato differente.

```
1 struct    {int x; int y;} p1;
2 struct    {int x; int y;} p2 = p1; // Errore
3 struct S1 {int x; int y;} p3 = p1; // Errore
4 struct S2 {int x; int y;} p4 = p3; // Errore
```

Operazioni su tipi di dati struct

- Per **accedere ai campi** di un tipo di dato `struct`, usiamo l'operatore `.` (punto).

```
1 struct point {int x; int y;} p;
2 struct line  {struct point coord1; struct point coord2;} l;
3
4 p.x = p.y = 0;
5 l.coord1 = p;
6 l.coord2 = p;
7 printf("(%d,%d)\n",l.coord1.x,l.coord1.y);
8 printf("(%d,%d)\n",l.coord2.x,l.coord2.y);
```

- Possiamo **assegnare** ad un variabile di tipo `struct` il contenuto di una variabile dello stesso tipo.
- Non è possibile utilizzare gli altri operatori con un tipo di dato `struct`, ma possiamo farlo con i campi della struttura.

```
1  struct point { int x; int y; } p1 = {0,1}, p2 = {1,0};
2
3  p1    = p2;                    //  OK
4  p1    = p1 + p2;               // Errore di sintassi;
5  p1.x = p1.x + p2.x;            // OK
6  p1.y = p1.y + p2.y;            // OK
7  if(p1==p2)                     // Errore di sintassi
8    printf("Same coord\n");
9  if(p1.x==p2.x && p1.y==p2.y) // OK
10   printf("Same coord\n");
```

Ulteriori caratteristiche del tipo `struct`

- A differenza del tipo enumerativo, le dichiarazioni *vuote* di un tipo `struct` possono essere utilizzate per dichiarare la struttura prima della sua definizione (**forward declaration**). Parliamo di **dichiarazione opaca** di struttura.

```
1 // Dichiarazione di esistenza della struttura point
2 struct point p1;  // p1 variabile di tipo struct point
3
4 // Dichiarazione  della struttura point
5 struct point { int x; int y; };
```

- L'uso di *dichiarazioni opache* di strutture è per certi aspetti vicino all'uso che facciamo dei prototipi per le funzioni: in programmi complessi possiamo *esporre* il nome della struttura a file esterni, senza mostrarne il suo effettivo contenuto.
  - Il tipo di dato struttura diventa noto ma non i suoi campi, che non potranno essere referenziati direttamente da chi vede solo la dichiarazione.
- La forward declaration (in forma implicita) ci permette anche di definire **strutture autoreferenziali**: strutture che contengono campi il cui tipo è la struttura stessa.
- La limitazione che ci impone il linguaggio C è che tali campi debbano essere di tipo *puntatore alla struttura*.
- Vedremo questo aspetto della forward declaration quando parleremo di puntatori.

I bit-field

- I **bit-field** (campi di bit) sono locazioni di memoria contigue di **bit**.
- Possono rappresentare valori di tipo `int` (`signed` o `unsigned`).
- Possono essere utilizzati per salvare spazio di memoria quando l'insieme di valori da rappresentare per una data variabile è di molto inferiore al numero di valori rappresentabili con i tipi di dato intero.
- Possiamo definire bit-field in strutture, specificando dopo il nome del campo il simbolo `:` seguito dai numero di bit da utilizzare.
- I bit sono accorpati in gruppi di byte delle dimensioni di un `int` (`signed` o `unsigned`).

```c
// Carta da gioco con bit-field
struct card {
  unsigned int value : 4; // 13 carte
  unsigned int seed  : 2; // 4 semi
  unsigned int color : 1; // 2 colori
};
```

```c
// Carta da gioco
struct card {
  unsigned int value; // carta
  unsigned int seed;  // seme
  unsigned int color; // colore
};
```

La prima struttura avrà tipicamente dimensione `sizeof(int)` (7 bit sono meno di un byte) mentre la seconda `3*sizeof(int)`.

- I bit-field possono essere creati usando i tipi `signed int` o `unsigned int`. Alcuni compilatori supportano altri tipi (estensioni non standard e non portabili).

I bit-field: ulteriori caratteristiche

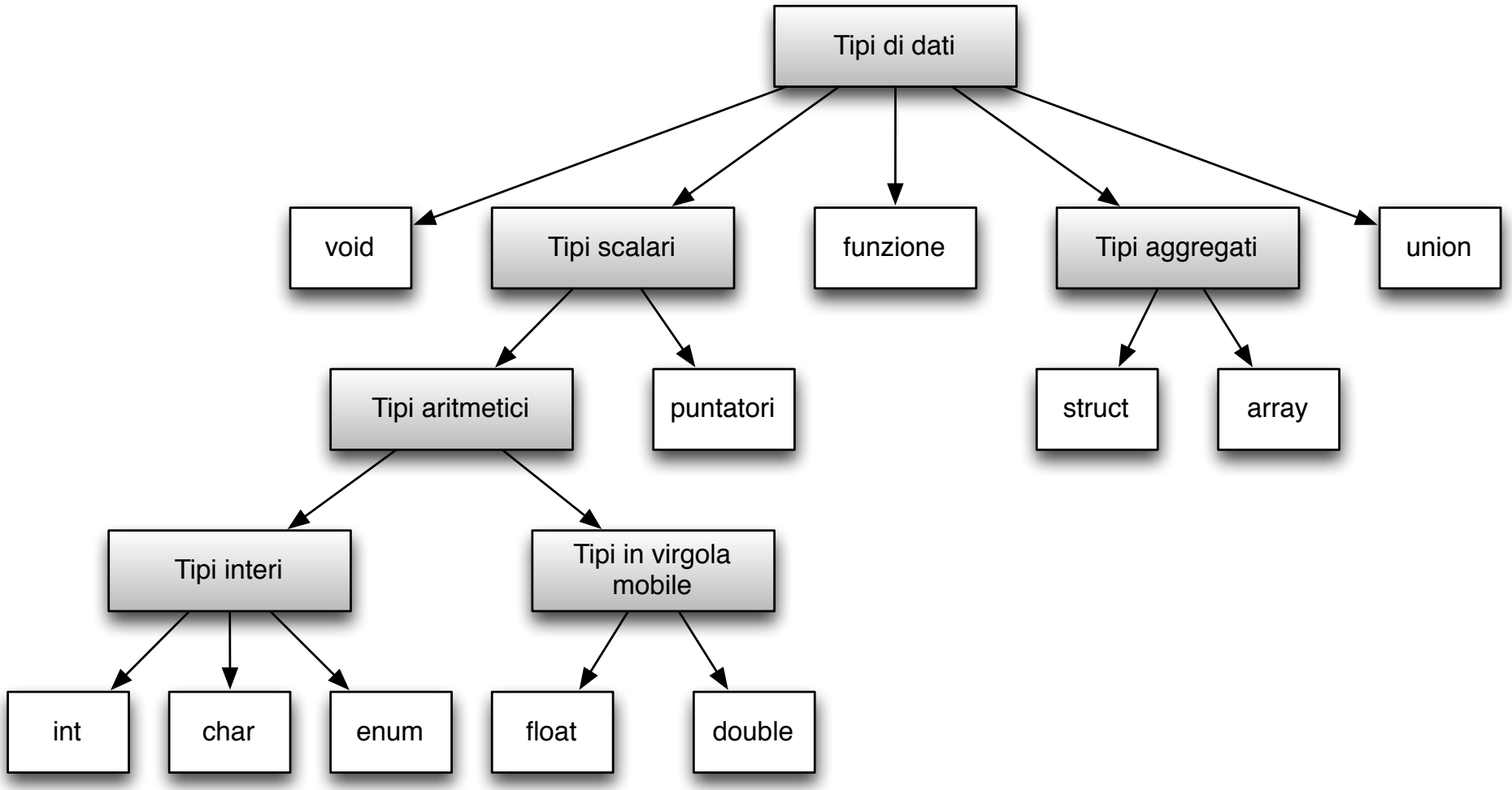
- E' possibile definire bit-field *anonimi*, utilizzati unicamente come riempitivo.
- Se il valore di un campo anonimo è zero, il membro successivo viene *allocato* in un nuovo `signed` o `unsigned int`.

```
1  struct card {
2    unsigned int value : 4; // 13 carte
3    unsigned int       : 1; // Riempitivo
4    unsigned int seed  : 2; // 4 semi
5    unsigned int       : 0; // Il bit color    allocato in nuovo int
6    unsigned int color : 1; // 2 colori
7  };
```

  Questa struttura avrà dimensione almeno `2*sizeof(int)`.
- Valgono le stesse regole di conversione viste per i tipi di dato interi con e senza segno.

```
1  enum COLOR {R,B};
2  enum SEED  {C,D,H,S};
3  enum VALUE {C1=1,C2,C3,C4,C5,C6,C7,C8,C9,C10,CJ,CQ,CK};
4
5  struct card {
6    unsigned int value : 4; // valori 0..16
7    unsigned int seed  : 2; // valori 0..3
8    unsigned int color : 1; // valori 0,1
9  } x = {CJ,H,B};
10 x.value =  17;  // Overflow
11 x.seed  = -1;   // Convertito in unsigned int (S)
12 x.color =  1.2; // Troncamento (B)
```

Classificazione dei tipi di dati in C



Il tipo union

- Nei linguaggi di programmazione, una *unione* è un tipo di dato che permette di contenere tipi di dato differenti nella stessa locazione di memoria.
- Un tipo di dato unione può essere dichiarato con molti campi differenti, ma soltanto uno di questi campi può contenere un valore in un dato momento.
- I vari tipi di dato specificati in una unione condividono la stessa locazione di memoria e sono mutualmente esclusivi.
- Le unioni sono tipicamente utilizzate per ottimizzare l'uso della memoria.
- Nel linguaggio C possiamo dichiarare unioni utilizzando la parola chiave `union`.
- La sintassi del costruttore `union` è praticamente la stessa del costruttore `struct`.
- A livello semantico, i due costruttori dichiarano tipi di dati molto differenti:
  - se dichiariamo una struttura `S` che contenga un `int` e un `double`, in una variabile di tipo `struct S` potremo memorizzare **un** `int` **e un** `double`;
  - se dichiariamo una unione `U` che contenga un `int` e un `double`, in una variabile di tipo `union U` potremo memorizzare **un** `int` **o un** `double`.

Dichiarazione di tipi union

- La sintassi per dichiarare un tipo union è la stessa vista per il tipo struct.

```
union [Tag] {
  <Tipo1> <Nome1>;
  <Tipo2> <Nome2>;
  ...
  <TipoN> <NomeN>;
};
```

- Come per i tipi enum e struct, il tag del tipo union è opzionale e la dichiarazione del tipo non riserva memoria.
- La dimensione di memoria del tipo di dato union è sufficientemente ampia da poter contenere il più grande dei tipi specificati nella dichiarazione.

```
1 union xtype {
2   char   ctype; // tipo carattere
3   int    itype; // tipo intero
4   float  ftype; // tipo float
5   double dtype; // tipo double
6 };
```

  Tipicamente, in questo esempio sizeof(union xtype) sarà uguale a sizeof(double).
- Gli operatori utilizzabili con le unioni sono gli stessi visti per le strutture.
- Valgono gli stessi limiti di annidamento e numero di campi imposti alle strutture.

Dichiarazione e inizializzazione di variabili di tipo union

- La dichiarazione di variabili di tipo union segue le regole viste per i tipi enum e struct.
- E' possibile inizializzare solo il primo membro nella dichiarazione di variabile.
- Il programmatore deve avere cura di ricordare quale campo sta utilizzando.

```c
#include <stdio.h>

union xtype {
  char   ctype; // tipo carattere
  int    itype; // tipo intero
  float  ftype; // tipo float
  double dtype; // tipo double
};

int main() {
  union xtype x = {'A'}; // Inizializza .ctype

  printf("x = %c\n",x.ctype); // Stampa 'A'
  printf("x = %d\n",x.itype); // Garbage o 65
  printf("x = %g\n",x.ftype); // Garbage
  printf("x = %g\n",x.dtype); // Garbage

  x.ftype = 10.15;
  printf("x = %c\n",x.ctype); // Garbage
  printf("x = %d\n",x.itype); // Garbage
  printf("x = %g\n",x.ftype); // Stampa 10.15
  printf("x = %g\n",x.dtype); // Garbage
  return 0;
}
```

Tecniche tipiche per l'uso di tipi union

- I tipi di dato union sono generalmente utilizzati facendo uso di un *discriminante*, che permette di mantenere traccia del campo valido in particolare momento.
- E' necessario annidare la union in una struct.

```c
#include <stdio.h>

struct xtype {
  // Discriminante (0=char, 1=int, 2=float, 3=double)
  unsigned int type;
  // Variabile di tipo union
  union {char ctype; int itype; float ftype; double dtype;} val;
};

int main() {
  struct xtype x;

  x.val.ftype = 10.15; x.type = 2;

  switch(x.type) { // Il discriminante marca il campo attivo
    case 0: printf("x = %c\n",x.val.ctype); break;
    case 1: printf("x = %d\n",x.val.itype); break;
    case 2: printf("x = %g\n",x.val.ftype); break;
    case 3: printf("x = %g\n",x.val.dtype); break;
  }

  return 0;
}
```

I bit-field con il tipo union

- Possiamo utilizzare i bit-field anche con i tipi di dato `union`.
- Esempio. Tipo di dato che implementa un dato generico a 4,6,8 o 10 facce:

```
1
2  // Struttura generica di dado a 4,6,8 o 10 facce
3  struct dice {
4    unsigned int type  : 2; // Discriminante (0=d4, 1=d6, 2=d8, 3=d10)
5    union {
6      unsigned int d4  : 2; // 4 facce
7      unsigned int d6  : 3; // 6 facce
8      unsigned int d8  : 3; // 8 facce
9      unsigned int d10 : 4; // 10 facce
10   } side; // Tipi di dado, mutualmente esclusivi
11 };
```

- Da notare in questo esempio i campi `d6`, `d8` (riga 7 e 8, rispettivamente), dichiarati con lo stesso numero di bit.
- Se stiamo utilizzando un dado a 6 o 8 facce, potremo stampare correttamente il contenuto della `union` utilizzando indifferentemente il campo `d6` o `d8`: condividono la stessa porzione di memoria di 3 bit.

L'operatore typedef

- L'operatore **typedef** (keyword) permette di creare nuovi nomi per i tipi di dato.

  `typedef <Tipo> <Nome>;`

- La dichiarazione del nuovo tipo è identica alla dichiarazione di una variabile, tranne per il fatto che la dichiarazione è preceduta dalla parola chiave `typedef`.
- Il nome dichiarato diventa un *sinonimo* per il tipo di dato dichiarato.

```c
typedef double length; // il tipo length     sinonimo di double
typedef double weight; // il tipo weight     sinonimo di double
```

- I nuovi tipi sono trattati dal compilatore come equivalenti ai tipi originari.

```c
typedef double heigth;
typedef double weight;

heigth h;    // Altezza in metri
weight w;    // Peso in chilogrammi
double IMC; // Indice di massa corporea

scanf("%lf %lf",&h,&w); // Leggiamo h e w come tipi double
IMC = w/(h*h);          // Utilizziamo h e w come tipi double
```

L'operatore typedef con i tipi di dato enum, struct, union

- Possiamo utilizzare l'operatore `typedef` per dichiarare sinonimi per i tipi di dati `enum`, `struct`, `union`.

```
typedef enum   {FALSE, TRUE} bool;
typedef struct {int x; int y;} point;
typedef union  {int itype; double ftype;} xtype;

bool flag   = TRUE;
point coord = {0,0};
xtype var   = {0};
```

Tranne quando il nome del tipo è molto esplicativo (come per il tipo `bool`) è sempre preferibile non nascondere la *natura* del tipo di dato: osservando gli identificatori `point` e `xtype` nell'esempio, non riusciamo a capire che si tratti rispettivamente di un tipo struttura o di un tipo unione.

- L'operatore `typedef` cambia completamente la semantica delle dichiarazioni:

```
struct {
  int x;
  int y;
} point1, point2; // Variabili
```

```
typedef struct {
  int x;
  int y;
} point1, point2; // Tipi
```

Nei due esempi, gli identificatori `point1, point2` sono rispettivamente: variabili (primo esempio), tipi di dati (secondo esempio).

L'operatore typedef con i tipi di dato funzione

- Possiamo utilizzare l'operatore `typedef` anche per definire un *tipo di dato funzione*.

  `typedef <Tipo> <Nome>(<Tipo1> [<Arg1>], .., <TipoN> [<ArgN>]);`

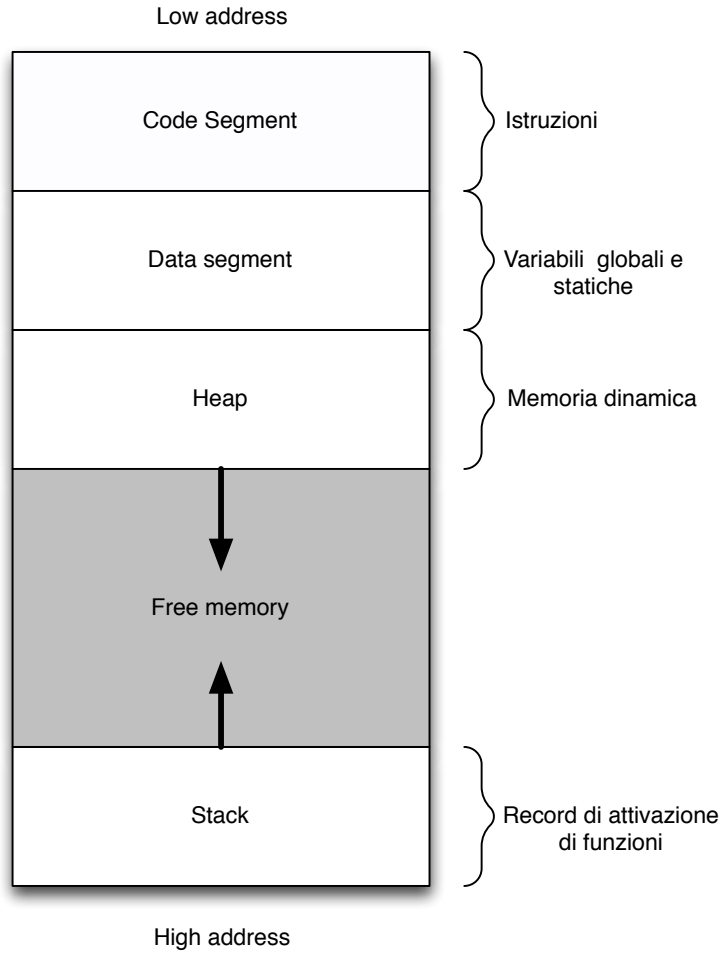
- La dichiarazione del nuovo tipo è identica alla dichiarazione di un prototipo, tranne per il fatto che la dichiarazione è preceduta dalla parola chiave `typedef`.
- Il nome (identificatore) dichiarato viene adesso interpretato non come nome di funzione ma come nome di un *tipo di dato funzione*, il cui tipo di ritorno e tipo dei parametri formali sono quelli specificati nella dichiarazione.

```
1 typedef int func_type(int x, int y);
```

  Il tipo `func_type` rappresenta un tipo di dato funzione che ritorna un intero e prende in input due argomenti di tipo intero.

- In C non possiamo dichiarare una variabile di tipo funzione ma possiamo dichiarare un puntatore ad un tipo funzione.
- Allo stesso modo, le funzioni in C non possono ritornare o avere argomenti di tipo funzione ma possono ritornare e avere argomenti di tipo puntatore a funzione.
- In particolare, in un programma C il nome di funzione è sinonimo dell'indirizzo di memoria della prima istruzione della funzione (che risiede nel code segment).

Ripasso: Spazio di indirizzamento virtuale



Struttura semplificata dello spazio di indirizzamento virtuale.

La portabilità con typedef

- L'operatore `typedef` viene spesso utilizzato per nascondere come il compilatore gestisce alcuni tipi di dato.
- La motivazione è quella di fornire al programmatore un comportamento *standard* in modo da aumentare la portabilità del codice.
- Esempio:
  - Il tipo `size_t` definito nell'header `stddef.h` dichiara un *tipo di dato senza segno*, utilizzato per rappresentare la dimensione in byte di un oggetto (è il tipo restituito dall'operatore `sizeof()`).
  - Il risultato dell'applicazione dell'operatore `sizeof` è una costante intera di tipo `size_t`.
  - La effettiva dimensione del tipo `size_t` dipende dall'implementazione: potrebbe, ad esempio, essere `unsigned int` o `unsigned long int`.
  - Specificare il tipo `size_t` per variabili che sono utilizzate unicamente per memorizzare dimensioni in byte rende il codice maggiormente portabile.
- Per convenzione, i nomi dei tipi di dato che terminano con `_t` sono nomi riservati e non dovrebbero essere utilizzati dal programmatore per definire nuovi tipi.

Riutilizzo degli identificatori (1/2)

- Abbiamo visto tutti i costrutti del linguaggio che fanno uso di identificatori definiti dal programmatore:
  - nomi e parametri di macro,
  - nomi di variabili e funzioni,
  - nomi di strutture, unioni e enumerazioni (tag),
  - nomi di campi in strutture e unioni,
  - nomi di costanti enumerative,
  - nomi di tipi definiti dal programmatore (`typedef`)
  - nomi di label (per l'uso del costrutto `goto`).
- Vediamo le regole generali che ci permetto di riutilizzare un identificatore all'interno di un programma:
  - namespace degli identificatori,
  - visibilità (scope) degli identificatori.
- Le regole di visibilità e il namespace definiscono interamente le modalità per poter riutilizzare gli identificatori in un programma C.

Namespace degli identificatori

- Il **namespace** (spazio dei nomi) è un insieme di classi in cui sono partizionati i tipi di identificatori in un programma C.
- In C abbiamo quattro classi di namespace:
  1. I **tag** (nomi) di `struct`, `union` e `enum`.
  2. I **campi** di `struct` e `union`. Ogni `struct` o `union` ha un namespace differente.
  3. Le **label** (utilizzate con il costrutto `goto`).
  4. Tutti gli altri **identificatori** (nomi di variabili, funzioni, costanti enumerative, tipi).
- Abbiamo inoltre un namespace a parte per le macro e argomenti delle macro.
  - Le sostituzioni di macro sono effettuate prima della compilazione.
- E' possibile riutilizzare gli stessi identificatori se appartengono a namespace differenti.
- Possiamo, ad esempio, utilizzare lo stesso nome per una struttura (classe 1) ed una funzione (classe 4) nello stesso scope.

Visibilità degli identificatori (1/2)

- Le regole di visibilità degli identificatori estendono quelle viste per i nomi di variabili.

Visibilità di variabili e funzioni: riassunto

- Variabili:
  - Una variabile globale è visibile a livello di programma.
  - Una variabile globale `static` è visibile solo a livello di file.
  - Una variabile locale è visibile solo nel blocco in cui è stata dichiarata.
- Funzioni
  - Una funzione è visibile alle funzioni che seguono la sua dichiarazione/definizione.
  - Una funzione nello stesso programma può essere invocata anche se non visibile, a patto che il suo tipo di ritorno sia `int` (sintassi K&R).
  - Una funzione `static` è visibile solo a livello di file.

```
1  #include<stdio.h>
2  int x = 0;              // Variabile visibile a livello di programma
3  static int y = 0;       // Variabile visibile a livello di file
4
5  static int f(void) { // Funzione visibile a livello di file
6    return 10;
7  }
8
9  int g(void) {           // Funzione invocabile a livello di programma
10   int i;                // Variabile visibile a livello di blocco (funzione)
11   for(i=x; i<=f(); i++) {
12     int j = y-i;        // Variabile visibile a livello di blocco
13     printf("%d\n",j);
14   }
15   return i;
16 }
```

Visibilità degli identificatori (2/2)

- Le regole di visibilità degli identificatori estendono quelle viste per i nomi di variabili.
- Un identificatore è visibile nella regione di programma in cui è dichiarato.
- Abbiamo quattro classi di visibilità:
  1. Visibilità a **livello di file**.
     - Ogni identificatore dichiarato al di fuori di una funzione ha visibilità a livello del file in cui è dichiarato.
     - Ricordiamo che affinché una variabile globale sia visibile in altri file è necessario dichiararla come `extern` nei file in cui si vuole utilizzarla.
  2. Visibilità a **livello di funzione**.
     - Le label (`goto`) sono gli unici identificatori con visibilità a livello di funzione.
     - E' possibile referenziare una label all'interno di una funzione, indipendentemente dal punto in cui è dichiarata nella stessa funzione.
     - Non possiamo dichiarare due label con lo stesso nome nella stessa funzione.
  3. Visibilità a **livello di prototipo di funzione**.
     - I nomi dei parametri formali in un prototipo (non definizione di funzione) hanno uno scope limitato alla dichiarazione.
     - Non possiamo dichiarare due parametri con lo stesso nome.
  4. Visibilità a **livello di blocco**.
     - Gli identificatori dichiarati in un blocco sono visibili unicamente nel blocco.
     - Un identificatore in un blocco *maschera* un uguale identificatore nello stesso namespace, esterno al blocco.

Riutilizzo degli identificatori (2/2)

- Le regole di visibilità e il namespace definiscono interamente le modalità per poter riutilizzare gli identificatori in un programma C.
  1. Non possiamo riutilizzare lo stesso identificatore nello stesso namespace e scope.
  2. Nello stesso scope, possiamo riutilizzare uno stesso identificatore se questo è utilizzato in namespace differenti.
  3. In scope differenti, possiamo riutilizzare uno stesso identificatore dichiarato in qualche scope più esterno. In questo caso, l'identificatore nello scope più interno *maschera* localmente la visibilità dell'identificatore nello scope più esterno.

Riutilizzo degli identificatori: esempio 1/3

- Nel seguente esempio (sintatticamente corretto) tutti gli identificatori uguali appartengono a namespace differenti e sono dichiarati nello stesso scope (corpo del `main()`).

```
1  int main() {
2    typedef int D; // nome di tipo (classe 4)
3
4    struct A {     // tag (classe 1)
5     int A;        // campo di struct (classe 2)
6     int B;        // campo di struct (classe 2)
7    };
8
9    union B {      // tag (classe 1)
10    int A;        // campo di union (classe 2)
11    int B;        // campo di union (classe 2)
12   };
13
14   enum C {       // tag (classe 1)
15    B,            // identificatore (classe 4)
16    C             // identificatore (classe 4)
17   };
18
19   int A;         // identificatore (classe 4)
20   A:             // label (classe 3)
21   B:             // label (classe 3)
22   C:             // label (classe 3)
23   D:             // label (classe 3)
24   return A;
25 }
```

Riutilizzo degli identificatori: esempio 2/3

- Non possiamo riutilizzare un identificatore nello stesso namespace e contemporaneamente nello stesso scope. In scope differenti, l'identificatore più interno *maschera* l'identificatore più esterno.

```
1 void func() {
2   struct A {int x;};
3   union  A {int x;}; // Errore
4 }
```

```
1 void func() {
2     struct A {int x;};
3   { union  A {int x;}; } // OK
4 }
```

- Esempio con costanti enumerative e variabili.

```
1 void func() {
2   enum E {A,B};
3
4   int  A; // Errore
5 }
```

```
1 enum E {A,B};
2
3 void func() {
4  int A; // OK
5 }
```

- Esempio con nomi di variabili e funzioni.

```
1 int func;
2
3 void func() { // Errore
4 }
```

```
1 void func() {
2   int func; // OK
3
4 }
```

Riutilizzo degli identificatori: esempio 3/3

- I nomi e parametri delle macro appartengono ad un namespace distinto da quello degli altri identificatori in un programma C.

```c
#define B X

int A;
int B; // OK
```

```c
#define B A

int A;
int B; // Errore: diventa A
```

- Possiamo utilizzare lo stesso nome per tag e tipo di dato dichiarato con la `typedef`.

```c
typedef struct S {
  int x;
} S;

struct S x; /*S    la struttura*/
S y; /* S e' il tipo */
x = y; // OK: stesso tipo
```

```c
typedef union S {
  int x;
} S;

union S x;
S y;
x = y; // OK: stesso tipo
```

- I nomi di tipi appartengono allo stesso namespace dei nomi di variabili e funzioni.

```c
typedef int T;

void T() { // Errore
}
```

```c
typedef int T;

int T;    // Errore
void F() {
  int T; // OK
}
```

Funzioni e tipi di dati struct

- Vediamo quali sono le principali caratteristiche di funzioni che abbiano strutture come parametri formali e come tipo di ritorno.
- Ci concentriamo unicamente sul tipo di dato `struct` poiché:
  - i tipi enumerativi sono trattati come `int`,
  - le modalità di utilizzo del tipo `union` con funzioni sono (essenzialmente) le stesse che vedremo per il tipo `struct`,
  - le modalità di utilizzo dei tipi definiti dall'utente dipendono dal tipo originario.
- Possiamo definire/dichiarare funzioni che abbiano parametri formali e/o valore di ritorno di tipo `struct`.

```
struct point {int x; int y;};

double distance(struct point x, struct point y);

struct point origin(void);
```

- E' necessario prestare attenzione ai seguenti aspetti:
  1. come avviene il passaggio dei parametri di tipo `struct`,
  2. come viene passato al chiamante un valore di ritorno di tipo `struct`,
  3. visibilità del tipo di dato `struct` per tutte le funzioni che ne fanno uso.

Funzioni e tipi di dati `struct`

1. Passaggio di parametri di tipo `struct`.
   - In C il passaggio dei parametri avviene per valore, quindi il contenuto di una struttura verrà interamente ricopiato nella locazione di memoria riservata per il parametro formale della funzione.
   - Se la struttura è *ingombrante* paghiamo un costo in termini di *tempo per la gestione del record di attivazione* (difficilmente verranno usati i registri).
2. Valore di ritorno di tipo `struct`.
   - Se la `struct` è *ingombrante*, difficilmente sarà possibile utilizzare i registri del processore per il passaggio del valore alla funzione chiamante.
3. Visibilità del tipo di dato `struct`.
   - E' necessario che sia il chiamante che la funzione invocata *conoscano* la stessa definizione della struttura.
   - Ricordiamo che sono considerate tipi di dato differenti `struct` che abbiano:
     - lo stesso tag, ma dichiarate in scope differenti;
     - tag differenti, anche se dichiarate nello stesso scope;
     - nessun tag, in qualsiasi scope.
   - Queste regole valgono anche per dichiarazioni di `struct` perfettamente identiche (stessi campi).
   - E' necessario quindi dichiarare le strutture in modo tale che siano visibili globalmente a tutte le funzioni nel programma che ne vogliano fare uso.

Funzioni e tipi di dati struct: esempio

```
1  #ifndef TRIANGLE_H
2  #define TRIANGLE_H
3
4  // Definisce una coordinata 2D
5  struct point {
6   double x;
7   double y;
8  };
9
10 // Definisce un set di 3 coordinate 2D
11 struct triangle {
12  struct point A;
13  struct point B;
14  struct point C;
15 };
16
17 // Ritorna 1 se le coordinate sono collineari, 0 altrimenti.
18 int is_segment(struct triangle t);
19
20 // Ritorna un triangolo random con coordinate intere in [min,max].
21 //
22 // Nota: non setta il seed del generatore random.
23 struct triangle random_triangle(int min, int max);
24
25 #endif
```

Ripasso: #ifndef

- Le direttive `#if`, `#ifdef` e `#ifndef` permettono di escludere parti di codice dalla compilazione in base al verificarsi o meno di particolari condizioni.
- Ogni direttiva di questo tipo deve essere *chiusa* da una direttiva `#endif`.
- Nel linguaggio C la direttiva ifndef controlla se un simbolo è stato dichiarato con la direttiva define in fase di preprocessione del codice. La direttiva è la contrazione di IF Not DEFine.
- Se il simbolo non è definito, il preprocessore aggiunge al codice il blocco di istruzioni tra ifndef e endif. Viceversa, lo esclude.

Funzioni e tipi di dati struct: esempio

```c
#ifndef TRIANGLE_H
#define TRIANGLE_H

// Definisce una coordinata 2D
struct point {
 double x;
 double y;
};

// Definisce un set di 3 coordinate 2D
struct triangle {
 struct point A;
 struct point B;
 struct point C;
};

// Ritorna 1 se le coordinate sono collineari, 0 altrimenti.
int is_segment(struct triangle t);

// Ritorna un triangolo random con coordinate intere in [min,max].
//
// Nota: non setta il seed del generatore random.
struct triangle random_triangle(int min, int max);

#endif
```

Funzioni e tipi di dati struct: esempio

```c
#include <stdlib.h>
#include "triangle.h"

int is_segment(struct triangle t) {
 /*  Verifica la condizione di allineamento
  *  per tre punti 2D (x1,y1) (x2,y2) (x3,y3):
  *
  *      (x2-x1)*(y3-y2) = (x3-x1)*(y2-y1)
  */
  double a = (t.B.x-t.A.x)*(t.C.y-t.A.y);
  double b = (t.C.x-t.A.x)*(t.B.y-t.A.y);
  return a==b;
}

struct triangle random_triangle(int min, int max) {
  struct triangle t;
  int M = min<max ? max : min; // Per sicurezza
  int m = min<max ? min : max; // Per sicurezza
  do { // Ricomincia se le coordinate random sono allineate
    t.A.x = ((int)random()%(M-m+1))+m;
    t.A.y = ((int)random()%(M-m+1))+m;
    t.B.x = ((int)random()%(M-m+1))+m;
    t.B.y = ((int)random()%(M-m+1))+m;
    t.C.x = ((int)random()%(M-m+1))+m;
    t.C.y = ((int)random()%(M-m+1))+m;
  } while(is_segment(t));
  return t;
}
```