PROGRAMMAZIONE PROCEDURALE

A.A. 2023/2024

STRUCTURES

STRUCTS

- Arrays allow to define type of variables that can hold several data items of the same kind. Similarly **structure** is another user defined data type available in C that allows to combine data items of different kinds.
- Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book −
  - ✓Title
  - ✓Author
  - ✓Subject
  - ✓Book ID
- To access the fields of a structure, use “.”

struct [*tag_name*] { *member_declaration_list* };

DEFINITION OF STRUCTURES

- To define a structure, you must use the **struct** statement. The struct statement defines a new data type.

```
struct [tag] {

   member definition;
   member definition;

   member definition;
};
```

```
struct Song {
        char title[64];
        char artist[32];
        char composer[32];
        short duration;
        struct Date published;
};
```

NAME SPACES

- The tags of structure types are a distinct name space: the compiler distinguishes them from variables or functions whose names are the same as a structure tag.
- Likewise, the names of structure members form a separate name space for each structure type.
- Uppercase helps you to understand when it is a struct.

MEMBERS

- The members of a structure may have any desired complete type, including previously defined structure types. They must not be variable-length arrays.

```
struct Song {
        char title[64];
        char artist[32];
        char composer[32];
        short duration;
        struct Date published;
};
```

```
struct Date {
        short int day;
        short int month;
        short int year;
};
```

- A structure type cannot contain itself as a member, as its definition is not complete until the closing brace (}).

EXAMPLE

```
struct Song {
        char title[64];
        char artist[32];
        char composer[32];
        short duration;
        struct Song similar;
};
```

EXAMPLE

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

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( ) {

   struct Books Book1;        /* Define Book1 of type Book */
   struct Books Book2;        /* Define Book2 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* print Book1 info */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   return 0;
}
```

SELF REFERENTIAL STRUCTURES

- However, structure types can and often do contain pointers to their own type.
- Such *self-referential structures* are used in implementing linked lists, for example.

```
struct Cell { struct Song song;        // This record's data.
              struct Cell *pNext;      // A pointer to the next record.
};
```

STRUCTURE OBJECTS AND TYPEDEF NAMES

- Within the scope of a structure type definition, you can declare objects of that type:
  - ✓struct Song song1, song2, *pSong = &song1;
- The keyword **struct** must be included whenever you use the structure type.
- You can also use *typedef* to define a one-word name for a structure type:

```
typedef struct Song Song_t;            // Song_t is now a synonym for
                                       // struct Song.
Song_t song1, song2, *pSong = &song1;  // Two struct Song objects and a
                                       // struct Song pointer.
```

EXAMPLE

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

OR

```
struct S {
    int x;
};

typedef struct S T;
```

```
struct S var1;
T var2;
```

FUNCTIONS AND STRUCTS

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

struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

/* function declaration */
void printBook( struct Books book );

int main( ) {

   struct Books Book1;        /* Declare Book1 of type Book */

   /* book 1 specification */
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "Nuha Ali");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;

   /* print Book1 info */
   printBook( Book1 );

   return 0;
}
```

```c
void printBook( struct Books book ) {

   printf( "Book title : %s\n", book.title);
   printf( "Book author : %s\n", book.author);
   printf( "Book subject : %s\n", book.subject);
   printf( "Book book_id : %d\n", book.book_id);
}
```

scanf(“%s”, &book1.title)
scanf(“%d”, &book1.book_id)

POINTERS AND STRUCTS

- You can define pointers to structures in the same way as you define pointer to any other variable

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main() {
  struct Books *struct_pointer;
  struct Books Book1;
  struct_pointer = &Book1;

  // commands
}
```

ACCESSING STRUCTURE MEMBERS

- Two operators allow you to access the members of a structure object: the dot operator (**.**) and the arrow operator (**->**). Both of them are binary operators whose right operand is the name of a member.
- -> is a shortcut instead of
  - ✓(*pointer_to_structure).field

Precedence	Operators	Associativity

1.	Postfix operators:
	`[]  ()  .  ->  ++  --`
	`(type name){list}`
		Left to right

2.	Unary operators:
	`++  --`
	`!  ~  +  -  *  &`
		Right to left

EXAMPLE

```
struct Books {
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main() {
  struct Books *struct_pointer;
  struct Books Book1;
  struct_pointer = &Book1;

  (*struct_pointer).book_id= 1    // Primo esempio per accedere
  struct_pointer->book_id = 1     // Secondo esempio
}
```

COPY STRUCTURES

- You can use an assignment to copy the entire contents of a structure object to another object of the same type:
  - ✓Books book1, book2; book2 = book1;
- After this assignment, each member of book2 has the same value as the corresponding member of book1.

INITIALIZATION LIST

- To initialize a structure object explicitly when you define it, you must use an *initialization list*: this is a comma-separated list of *initializers*, or initial values for the individual structure members, enclosed in braces.
  - ✓The initializers are associated with the members in the order of their declarations
  - ✓Each initializer must have a type that matches (or can be implicitly converted into) the type of the corresponding member

EXAMPLE

```
struct Date {
        short int day;
        short int month;
        short int year;
};

struct Song { char title[64];
        char artist[32];
        char composer[32];
        short duration;
        struct Date published;
};

int main () {

struct Song mySong = { "What It Is",
                "Aubrey Haynie”,
                "Mark Knopfler",
                297,
                { 26, 9, 2000 }
};

// commands
}
```

INITIALIZING SPECIFIC MEMBERS

- You can explicitly associate an initializer with a certain member.
- To do so, you must prefix a *member designator* with an equal sign to the initializer. The general form of a designator for the structure member *member* is:
  - ✓ .*member* // Member designator

```
Song_t aSong = { .title = "I've Just Seen a Face”,
                 .composer = "John Lennon; Paul McCartney",
                 127 };
```

- 127 is the initialization of the first field after “composer”, i.e., “duration”.

ARRAYS OF STRUCT

```
struct Song { char title[64];
              char artist[32];
              char composer[32];
              short duration;
              struct Date published;
};

int main () {

  struct Song array_of_songs[100];

  // commands
}
```

SIZEOF OF A STRUCT

```
#include <stdio.h>
struct Song {
        char title[64];
        char artist[32];
        char composer[32];
        short duration;
};


int main () {

  struct Song songVar;

  printf("La dimensione di una struttura Song in bytes e': %ld", sizeof(songVar));

}
```

```
MacBook-Francesco:ProgrammI francescosantini$
./esempio
La dimensione di una struttura Song in bytes e': 130
```

UNIONS

WHAT UNIONS ARE

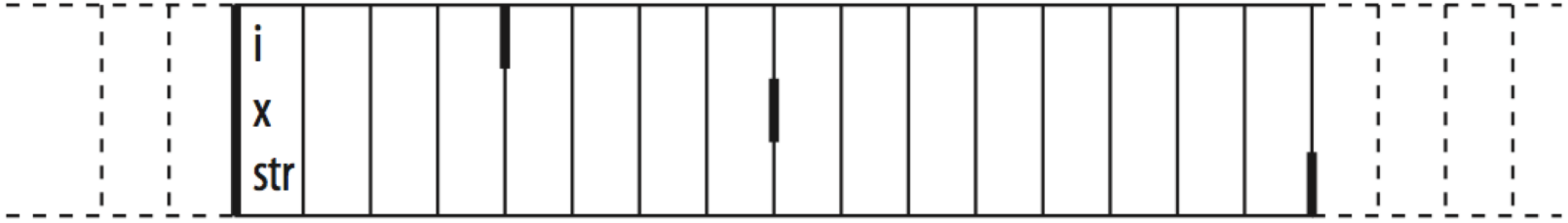
- Unlike structure members, which all have distinct locations in the structure, **the members of a *union* all share the same location in memory**:
- All members of a union start at the same address.
- Thus you can define a union with many members, **but only one member can contain a value at any given time**.
- Unions are an easy way for programmers to use a location in memory in different ways.

DEFINITION

- The definition of a union is formally the same as that of a structure, except for the keyword union in place of struct:
  - ✓*union [tag_name] { member_declaration_list };*
- An object of this type can store an integer, a floating-point number, or a short string.
  - ✓union Data { int i; double x; char str[16]; };
- A union is big as its largest member.
  - ✓Using our example, sizeof(union Data) yields the value 16.

DIFFERENCE IN MEMORY WRT STRUCTS

```
union Data { int i; double x; char str[16]; };
```




```
struct Data { int i; double x; char str[16]; };
```

EXAMPLE

```
union Data { int i; double x; char str[16]; };
```

```
var.x = 3.21;
var.x += 0.5;
strcpy( var.str, "Jim" );
myData[0].i = 50;
```

INITIALIZING UNIONS

- Like structures, union objects are initialized by an *initialization list*. For a union, though, the list can only contain one *initializer*.
- If the initializer has no member designator, then it is associated with the first member of the union.

```
union Data var1 = { 77 },
                var2 = { .str = "Mary" },
                var3 = var1,
                myData[100] = { {.x= 0.5}, { 1 }, var2 };
```

EXAMPLE

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

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   printf( "Memory size occupied by data : %d\n", sizeof(data));

   return 0;
}
```

```
Memory size occupied by data :
20
```

EXAMPLE

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

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   printf( "data.i : %d\n", data.i);

   data.f = 220.5;
   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

EXAMPLE

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

union Data {
   int i;
   float f;
   char str[20];
};

int main( ) {

   union Data data;

   data.i = 10;
   data.f = 220.5;
   strcpy( data.str, "C Programming");

   printf( "data.i : %d\n", data.i);
   printf( "data.f : %f\n", data.f);
   printf( "data.str : %s\n", data.str);

   return 0;
}
```

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

SU LIBRO

- Sezione 10.1-10.8