Elementi Di Informatica E Programmazione

Prof. Andrea Loreggia

UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

typedef

- **typedef** si usa per creare/rinominare un tipo di dato
  - Convenienza dei nomi
  - Chiarire l’uso di un tipo
  - Leggibilità del codice
  - Portabilità
- Esempio

```
typedef int size_t;
typedef long int32;
typedef long long int64;
```

Tipo di enumerazione

UNIVERSITÀ DEGLI STUDI DI BRESCIA

- Insieme di costanti intere rappresentate da nomi (= identificatori)
  - Gli identificatori sono associati agli interi a partire da 0
- Definito con la keyword enum

```
/* CUORI=0; QUADRI=1; FIORI=2; PICCHE=3 */
enum seme {CUORI, QUADRI, FIORI, PICCHE};
enum seme miaCarta;
int carteEstratte[4] = {0, 0, 0, 0};

miaCarta = Estrai();/* estrazione casuale di una carta */

/* conta le carte estratte per ogni seme */
if( miaCarta == CUORI )carteEstratte[cuori]++;
```

Tipo di enumerazione

- Anche per i tipi enum si può compattare la dichiarazione con l'uso di typedef
- L'associazione degli identificatori agli interi può essere specificata dall'utente

```
/* LUN=1; MAR=2; MER=3;… */
typedef enum {LUN=1, MAR, MER, GIO,VEN, SAB, DOM} giorno;
giorno day = MAR;
```

```
/* ALFA=1; GAMMA=3; DELTA=4; */
enum lettera {ALFA=1, GAMMA=3, DELTA};
```

Tipo di enumerazione

```
/* Stampa i nomi dei giorni della settimana */
#include <stdio.h>

main() {
       /* LUN=0; MAR=1; MER=2;… */
       typedef enum {LUN, MAR, MER, GIO, VEN, SAB, DOM } giorno;
       giorno g;
       char * nomeGiorno[] = {“Lunedi”, “Martedi”,
             “Mercoledi“, “Giovedi”, “Venerdi”, “Sabato”,
             “Domenica”};

       for(g=LUN; g<=DOM; g++)
             printf(“%s \n”, nomeGiorno[ g ]);
}
```

Structure Variables

- The properties of a ***structure*** are different from those of an array.
  - The elements of a structure (its ***members***) aren't required to have the same type.
  - The members of a structure have names; to select a particular member, we specify its name, not its position.
- In some languages, structures are called ***records,*** and members are known as ***fields.***

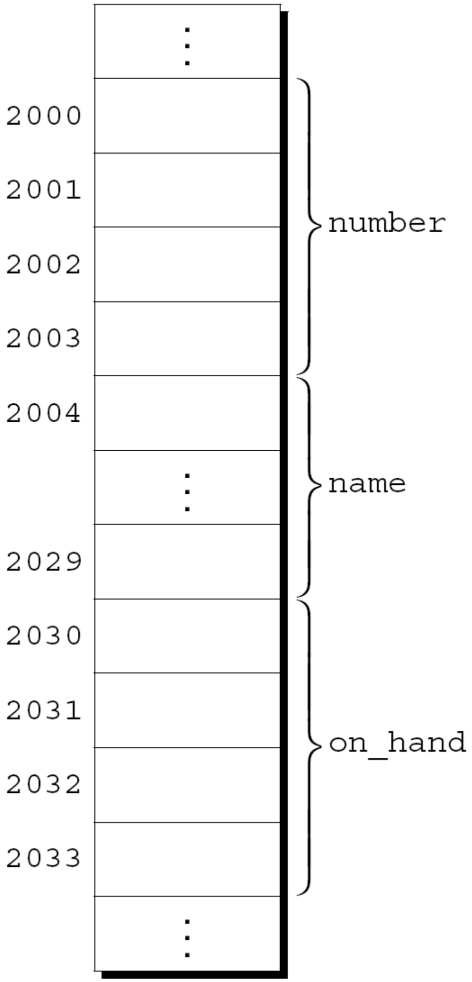
Declaring Structure Variables

- A structure is a logical choice for storing a collection of related data items.
- A declaration of two structure variables that store information about parts in a warehouse:

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

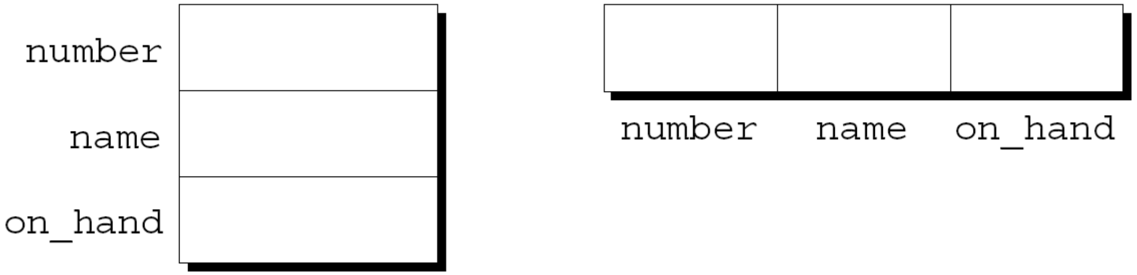
Declaring Structure Variables

- The members of a structure are stored in memory in the order in which they're declared.
- Appearance of `part1` ⟶
- Assumptions:
  - `part1` is located at address 2000.
  - Integers occupy four bytes.
  - `NAME_LEN` has the value 25.
  - There are no gaps between the members.



Declaring Structure Variables

- Abstract representations of a structure:



- Member values will go in the boxes later.

Declaring Structure Variables

- Each structure represents a new scope.
- Any names declared in that scope won't conflict with other names in a program.
- In C terminology, each structure has a separate ***name space*** for its members.

Declaring Structure Variables

- For example, the following declarations can appear in the same program:

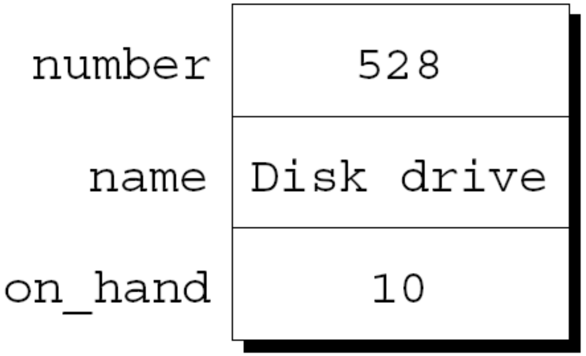
```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;

struct {
  char name[NAME_LEN+1];
  int number;
  char sex;
} employee1, employee2;
```

Initializing Structure Variables

- A structure declaration may include an initializer:

```
struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1 = {528, "Disk drive", 10},
  part2 = {914, "Printer cable", 5};
```

- Appearance of `part1` after initialization:



number 528

name Disk drive

on_hand 10

Initializing Structure Variables

- Structure initializers follow rules similar to those for array initializers.
- Expressions used in a structure initializer must be constant.
- An initializer can have fewer members than the structure it's initializing.
- Any "leftover" members are given 0 as their initial value.

Operations on Structures

- To access a member within a structure, we write the name of the structure first, then a period, then the name of the member.
- Statements that display the values of `part1`’s members:

```
printf("Part number: %d\n", part1.number);
printf("Part name: %s\n", part1.name);
printf("Quantity on hand: %d\n", part1.on_hand);
```

Operations on Structures

- The members of a structure are lvalues.
- They can appear on the left side of an assignment or as the operand in an increment or decrement expression:

```
part1.number = 258;
  /* changes part1's part number */
part1.on_hand++;
  /* increments part1's quantity on hand */
```

Operations on Structures

- The period used to access a structure member is actually a C operator.
- It takes precedence over nearly all other operators.
- Example:

```
scanf("%d", &part1.on_hand);
```

The `.` operator takes precedence over the `&` operator, so `&` computes the address of `part1.on_hand`.

Operations on Structures

- The other major structure operation is assignment:

```
part2 = part1;
```

- The effect of this statement is to copy `part1.number` into `part2.number`, `part1.name` into `part2.name`, and so on.

Operations on Structures

- The = operator can be used only with structures of ***compatible*** types.
- Two structures declared at the same time (as `part1` and `part2` were) are compatible.
- Structures declared using the same “structure tag” or the same type name are also compatible.
- Other than assignment, C provides no operations on entire structures.
- In particular, the == and != operators can’t be used with structures.

Structure Types

- Suppose that a program needs to declare several structure variables with identical members.
- We need a name that represents a *type* of structure, not a particular structure *variable*.
- Ways to name a structure:
  - Declare a “structure tag”
  - Use `typedef` to define a type name

Declaring a Structure Tag

UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

- A ***structure tag*** is a name used to identify a particular kind of structure.
- The declaration of a structure tag named `part`:

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
};
```

- Note that a semicolon must follow the right brace.

Declaring a Structure Tag

- The `part` tag can be used to declare variables:

```
struct part part1, part2;
```

- We can't drop the word `struct`:

```
part part1, part2;    /*** WRONG ***/
```

  `part` isn't a type name; without the word `struct`, it is meaningless.

- Since structure tags aren't recognized unless preceded by the word `struct`, they don't conflict with other names used in a program.

Declaring a Structure Tag

- The declaration of a structure *tag* can be combined with the declaration of structure *variables:*

```
struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} part1, part2;
```

Declaring a Structure Tag

- All structures declared to have type `struct part` are compatible with one another:

```
struct part part1 = {528, "Disk drive", 10};
struct part part2;

part2 = part1;
  /* legal; both parts have the same type */
```

Defining a Structure Type

- As an alternative to declaring a structure tag, we can use `typedef` to define a genuine type name.
- A definition of a type named `Part`:

```
typedef struct {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} Part;
```

- `Part` can be used in the same way as the built-in types:

```
Part part1, part2;
```

Structures as Arguments and Return Values

- Functions may have structures as arguments and return values.
- A function with a structure argument:

```
void print_part(struct part p)
{
  printf("Part number: %d\n", p.number);
  printf("Part name: %s\n", p.name);
  printf("Quantity on hand: %d\n", p.on_hand);
}
```

- A call of `print_part`:

```
print_part(part1);
```

Structures as Arguments and Return Values

- Within a function, the initializer for a structure variable can be another structure:

```
void f(struct part part1)
{
  struct part part2 = part1;
  …
}
```

- The structure being initialized must have automatic storage duration.

Arrays of Structures

- One of the most common combinations of arrays and structures is an array whose elements are structures.
- This kind of array can serve as a simple database.
- An array of `part` structures capable of storing information about 100 parts:

```
struct part inventory[100];
```

Arrays of Structures

- Accessing a part in the array is done by using subscripting:

```
print_part(inventory[i]);
```

- Accessing a member within a `part` structure requires a combination of subscripting and member selection:

```
inventory[i].number = 883;
```

- Accessing a single character in a part name requires subscripting, followed by selection, followed by subscripting:

```
inventory[i].name[0] = '\0';
```

Initializing an Array of Structures

```
const struct dialing_code country_codes[] =
  {{"Argentina",            54}, {"Bangladesh",      880},
   {"Brazil",               55}, {"Burma (Myanmar)",  95},
   {"China",                86}, {"Colombia",         57},
   {"Congo, Dem. Rep. of", 243}, {"Egypt",            20},
   {"Ethiopia",            251}, {"France",           33},
   {"Germany",              49}, {"India",            91},
   {"Indonesia",            62}, {"Iran",             98},
   {"Italy",                39}, {"Japan",            81},
   {"Mexico",               52}, {"Nigeria",         234},
   {"Pakistan",             92}, {"Philippines",      63},
   {"Poland",               48}, {"Russia",            7},
   {"South Africa",         27}, {"South Korea",      82},
   {"Spain",                34}, {"Sudan",           249},
   {"Thailand",             66}, {"Turkey",           90},
   {"Ukraine",             380}, {"United Kingdom",   44},
   {"United States",         1}, {"Vietnam",          84}};
```

- The inner braces around each structure value are optional.

Program: Maintaining a Parts Database

- The `inventory.c` program illustrates how nested arrays and structures are used in practice.
- The program tracks parts stored in a warehouse.
- Information about the parts is stored in an array of structures.
- Contents of each structure:
  - Part number
  - Name
  - Quantity

Program: Maintaining a Parts Database

- Operations supported by the program:
  - Add a new part number, part name, and initial quantity on hand
  - Given a part number, print the name of the part and the current quantity on hand
  - Given a part number, change the quantity on hand
  - Print a table showing all information in the database
  - Terminate program execution

Program: Maintaining a Parts Database

- The codes `i` (insert), `s` (search), `u` (update), `p` (print), and `q` (quit) will be used to represent these operations.
- A session with the program:

```
Enter operation code: i
Enter part number: 528
Enter part name: Disk drive
Enter quantity on hand: 10

Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 10
```

Program: Maintaining a Parts Database

```
Enter operation code: s
Enter part number: 914
Part not found.

Enter operation code: i
Enter part number: 914
Enter part name: Printer cable
Enter quantity on hand: 5

Enter operation code: u
Enter part number: 528
Enter change in quantity on hand: -2
```

Program: Maintaining a Parts Database

```
Enter operation code: s
Enter part number: 528
Part name: Disk drive
Quantity on hand: 8

Enter operation code: p
Part Number   Part Name             Quantity on Hand
    528       Disk drive                    8
    914       Printer cable                 5

Enter operation code: q
```

Program: Maintaining a Parts Database

- The program will store information about each part in a structure.
- The structures will be stored in an array named `inventory`.
- A variable named `num_parts` will keep track of the number of parts currently stored in the array.

Program: Maintaining a Parts Database

- An outline of the program’s main loop:

```
for (;;) {
  prompt user to enter operation code;
  read code;
  switch (code) {
    case 'i': perform insert operation; break;
    case 's': perform search operation; break;
    case 'u': perform update operation; break;
    case 'p': perform print operation; break;
    case 'q': terminate program;
    default: print error message;
  }
}
```

Program: Maintaining a Parts Database

- Separate functions will perform the insert, search, update, and print operations.
- Since the functions will all need access to `inventory` and `num_parts`, these variables will be external.
- The program is split into three files:
  - `inventory.c` (the bulk of the program)
  - `readline.h` (contains the prototype for the `read_line` function)
  - `readline.c` (contains the definition of `read_line`)

inventory.c

```
/* Maintains a parts database (array version) */

#include <stdio.h>
#include "readline.h"

#define NAME_LEN 25
#define MAX_PARTS 100

struct part {
  int number;
  char name[NAME_LEN+1];
  int on_hand;
} inventory[MAX_PARTS];

int num_parts = 0;   /* number of parts currently stored */

int find_part(int number);
void insert(void);
void search(void);
void update(void);
void print(void);
```

```
/**********************************************************
 * main: Prompts the user to enter an operation code,     *
 *       then calls a function to perform the requested   *
 *       action. Repeats until the user enters the        *
 *       command 'q'. Prints an error message if the user *
 *       enters an illegal code.                          *
 **********************************************************/
int main(void)
{
  char code;
  for (;;) {
    printf("Enter operation code: ");
    scanf(" %c", &code);
    while (getchar() != '\n')   /* skips to end of line */
      ;
    switch (code) {
      case 'i': insert();
                break;
      case 's': search();
                break;
      case 'u': update();
                break;
      case 'p': print();
                break;
      case 'q': return 0;
      default:  printf("Illegal code\n");
    }
    printf("\n");
  }
}
```

```c
/**********************************************************
 * find_part: Looks up a part number in the inventory     *
 *            array. Returns the array index if the part  *
 *            number is found; otherwise, returns -1.     *
 **********************************************************/
int find_part(int number)
{
  int i;

  for (i = 0; i < num_parts; i++)
    if (inventory[i].number == number)
      return i;
  return -1;
}
```

```
/**********************************************************
 * insert: Prompts the user for information about a new   *
 *         part and then inserts the part into the        *
 *         database. Prints an error message and returns  *
 *         prematurely if the part already exists or the  *
 *         database is full.                              *
 **********************************************************/
void insert(void)
{
  int part_number;

  if (num_parts == MAX_PARTS) {
    printf("Database is full; can't add more parts.\n");
    return;
  }
  printf("Enter part number: ");
  scanf("%d", &part_number);
  if (find_part(part_number) >= 0) {
    printf("Part already exists.\n");
    return;
  }

  inventory[num_parts].number = part_number;
  printf("Enter part name: ");
  read_line(inventory[num_parts].name, NAME_LEN);
  printf("Enter quantity on hand: ");
  scanf("%d", &inventory[num_parts].on_hand);
  num_parts++;
}
```

```
/**********************************************************
 * search: Prompts the user to enter a part number, then  *
 *         looks up the part in the database. If the part *
 *         exists, prints the name and quantity on hand;  *
 *         if not, prints an error message.               *
 **********************************************************/
void search(void)
{
  int i, number;

  printf("Enter part number: ");
  scanf("%d", &number);
  i = find_part(number);
  if (i >= 0) {
    printf("Part name: %s\n", inventory[i].name);
    printf("Quantity on hand: %d\n", inventory[i].on_hand);
  } else
    printf("Part not found.\n");
}
```

```c
/**********************************************************
 * update: Prompts the user to enter a part number.       *
 *         Prints an error message if the part doesn't    *
 *         exist; otherwise, prompts the user to enter    *
 *         change in quantity on hand and updates the     *
 *         database.                                      *
 **********************************************************/
void update(void)
{
  int i, number, change;

  printf("Enter part number: ");
  scanf("%d", &number);
  i = find_part(number);
  if (i >= 0) {
    printf("Enter change in quantity on hand: ");
    scanf("%d", &change);
    inventory[i].on_hand += change;
  } else
    printf("Part not found.\n");
}
```

```
/**********************************************************
 * print: Prints a listing of all parts in the database,  *
 *        showing the part number, part name, and         *
 *        quantity on hand. Parts are printed in the      *
 *        order in which they were entered into the       *
 *        database.                                       *
 **********************************************************/
void print(void)
{
  int i;

  printf("Part Number   Part Name                  "
         "Quantity on Hand\n");
  for (i = 0; i < num_parts; i++)
    printf("%7d       %-25s%11d\n", inventory[i].number,
           inventory[i].name, inventory[i].on_hand);
}
```

Program: Maintaining a Parts Database

- The version of `read_line` in Chapter 13 won’t work properly in the current program.
- Consider what happens when the user inserts a part:

```
Enter part number: 528
Enter part name: Disk drive
```

- The user presses the Enter key after entering the part number, leaving an invisible new-line character that the program must read.
- When `scanf` reads the part number, it consumes the 5, 2, and 8, but leaves the new-line character unread.

Program: Maintaining a Parts Database

- If we try to read the part name using the original `read_line` function, it will encounter the new-line character immediately and stop reading.
- This problem is common when numerical input is followed by character input.
- One solution is to write a version of `read_line` that skips white-space characters before it begins storing characters.
- This solves the new-line problem and also allows us to avoid storing blanks that precede the part name.

readline.h

```
#ifndef READLINE_H
#define READLINE_H

/***********************************************************
 * read_line: Skips leading white-space characters, then   *
 *            reads the remainder of the input line and    *
 *            stores it in str. Truncates the line if its *
 *            length exceeds n. Returns the number of      *
 *            characters stored.                           *
 ***********************************************************/
int read_line(char str[], int n);

#endif
```

readline.c

```
#include <ctype.h>
#include <stdio.h>
#include "readline.h"

int read_line(char str[], int n)
{
  int ch, i = 0;

  while (isspace(ch = getchar()))
    ;
  while (ch != '\n' && ch != EOF) {
    if (i < n)
      str[i++] = ch;
    ch = getchar();
  }
  str[i] = '\0';
  return i;
}
```