Operating Systems Lab (C+Unix)

Enrico Bini

University of Turin

Outline

Outline

Structures: declaration

- primitive data types: `int`, `char`, `double`, etc
- collection of homogeneous data: arrays
- collection of heterogeneous data: *structures*
- How to declare a structure? Example:

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

- Each piece of data is called *field* of the `struct`
- In the example, the `struct point` has 2 `double` fields with names `x` and `y`
- The name of the type is "`struct point`". Hence, variables of that type are declare by

```
struct point p1, p2;
```

Structures: initialization

- Initialization by listing values within curly braces `{...}` separated by commas

```
struct info {
    int id;
    char *name;
    int age;
};

struct info el1 = {3, "Aldo", 45};
```

- the initialization of each field must follow the order of declaration.

Structures: usage

- Each field of a `struct` is referred by the "dot" notation

```
struct info {
    int id;
    char *name;
    int age;
};
struct info v1;

v1.id = 10;
```

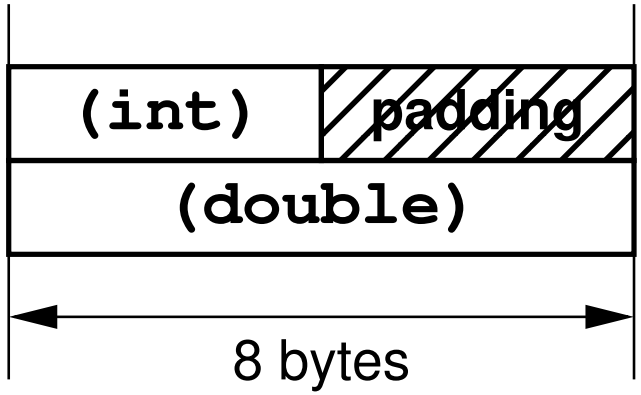
- When structures are accessed by pointers, each field of the pointed `struct` is referred by the notation "->"

```
struct info * p;

p = malloc(sizeof(*p));
p->age = 35; /*same as (*p).age = 35 */
```

Structures: byte alignment, padding

- How much memory is allocated to a struct? Where?

```
struct myrecord {
    int field1;
    double field2;
    /* more fields */
};
```



- Normally, fields are allocated in memory in the order they are declared
- Amount of memory of a `struct` **may be more** than sum of memory of each field

$$\texttt{sizeof(struct myrecord)} = \texttt{sizeof(field1)} + \\ + \texttt{sizeof(field2)} + \cdots + \text{"}\texttt{padding}\text{"}$$

- "padding" may be added to align the fields to "good" memory boundaries (multiples of 4, 8, or 16)
- *test-struct.c*

Structures: assignment

- `struct` may be assigned

```
struct info a, b;

a = b;
```

- however, they cannot by tested with the equal sign. The following code is incorrect

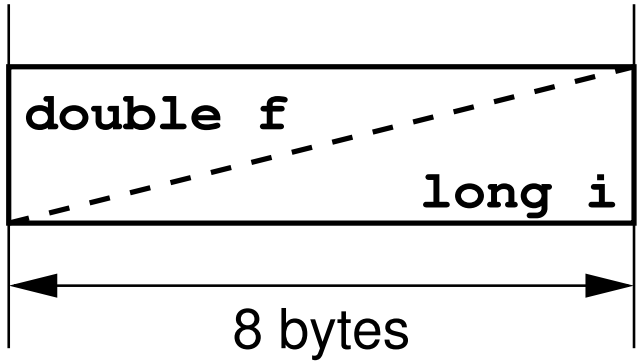
```
struct info a, b;

if (a == b) {
    ...
}
```

Outline

Unions

- The `union` data type is declared similarly to `struct`

```
union my_union_t {
    double f;
    long i;
};
```



- however all fields **overlaps** in memory, starting from the **same address**!! (the term "field" may sound a bit inappropriate for unions)
- if you modify one field, the others are modified too!!
- *test-union.c*
- hence, `sizeof(<union>)` is the size of the largest field
- unions are used to store alternatives
- `union` used to save memory (especially in embedded systems)

Outline

Enumerations

- Enumerations are used to define "labelled constants"
- A labelled constant is an integer constant with a name
- Example of declaration

```
enum month {Gen, Feb, Mar, Apr, May, Jun,
            Jul, Aug, Sep, Oct, Nov, Dec};
```

- *test-enum.c*
  - The value of the first constant is set to zero unless explicitly specified by the programmer (for example, with "`Gen = 1`")
  - From the second constant, the value is incremented unless the programmer specifies explicitly another value (for example, with "`May = 2`")
- The purpose of `enum` is to improve readability of code
- variables of `enum` type are replaced by their value in the assembly code

Outline

Defining new types

- `typedef` allows defining "new" types (to rename an old type)

```
typedef <old-type> <new-type>;
```

- Used to hide the real type used
  - good: when you do not trust who will read your code
  - bad: when you trust who will read your code (it may be complicated to go through many include files to understand the type of a variable)
- for example, `/usr/include/stdint.h` has many integer types defined which specifies the exact size of the integer

```
gedit /usr/include/stdint.h
```

- often types are also defined by pre-processor macros with `#define`.

```
#define MY_TYPE double

MY_TYPE my_var;
```

- Differences: macro-defined type is just a replacement by the pre-processor

Outline

Dynamic lists by `struct`, `typedef`, `malloc`, ...

- In C, dynamic lists are created by
  - defining the element of the list by a `struct`

```
typedef struct node {
  int value; /* or any data */
  struct node * next;
} node;

typedef node* list;
```

  - the `struct` has a pointer to the **next element**
  - setting a pointer `head` to the **head** of the list
  - the `.next` field of last element has value `NULL`
  - node insertion:
    1. new node allocated by `malloc(...)`
    2. the new node is properly linked
  - node removal:
    1. node is unlinked
    2. node memory deallocated by `free(...)`
- *test-list.c*