Computer Science

User-defined data types

Structures, Unions and Bit-fields in C

- Structures
- Unions
- Bit-fields
- `typedef`

Structures (I)

- A **structure** is a collection of variables of *different type* grouped together under a name for convenient handling
- Typically used to work with data bases
- Variables in the structure are called ***members***
- A **structure declaration** creates a *type* of structure without creating any concrete structure or variable

```
struct structuretypename
{
        datatype1 member1;
        …
        datatypeN memberN;
};
```

Structures (II)

- To instantiate a structure of a previously declared type:

```
struct structuretypename structurename
```

- It can be made as well in the initial declaration:

```
struct structuretypename
{
        datatype1 member1;
        …
        datatypeN memberN;
} structurenames;
```

- Structure declaration is placed before `main()` in the headers files `.h`
- The amount of memory that a structure occupies can be obtained with `sizeof`

Structures (III)

- Example

```
struct military   /* type of structure*/
{
        char name[40];
        char surname[80];
        unsigned age;
        unsigned long telephone;
} private, sergeant, lieutenant;

struct military captain;

/* private, sergeant, lieutenant and captain are
   struct variables of military type */
```

Structures (IV)

- The **operations** with structures are:
  - **To copy** `struct1 = struct2`
  - **To access** to a member `structurename.member`
  - **To take address** of a member `&structurename.member`
- Examples:

```
/* Initialization of some members of struct
sergeant of military type  */
gets(sergeant.name)
sergeant.age = 25;
scanf("%d", &sergeant.telephone);
```

Unions (I)

- A **union** is a variable that holds objects of different type and size, at different times (the programmer must know what type at what time)
- They provide a way to manipulate different kinds of data in the **same memory area**
- **Use:** Analogous to structures
  - **Declaration**:

```
union uniontypename
{
        datatype1 member1;
        …
};
```

  - **Instantiation**

```
union uniontypename unionname
```

  - **Access to a member**

```
unionname.membername
```

Unions (II)

- Example

```
union Size
{
      int number;           /* 38, 40, 42 */
      char letter;          /* P, M, G */
      char letters[4];      /* L, XL, XXL */
} tshirt, shirt, jersey;

tshirt.number = 44;
scanf("%c",&tshirt.letter);
gets(tshirt.letters);

/* First the integer 44 is stored, later the letter
read with scanf, and finally a string with at least
4 characters (null included) */
```

Bit-fields (I)

- A bit-field is a set of adjacent bits stored in a *word*
- They are defined as an structure and each bit is a *field* that can be accessed individually
- Definition `datatype fieldname:length;`
  - `datatype` can just be integer
  - `fieldname` is the bit-field name
  - `lenght` indicates the length of the bit-field
- Features:
  - Facilitate bit-level operations
  - Facilitate Boolean variable storage
  - They increase number of CPU operations (parallelism)
  - Save memory

Bit-fields (II)

- Restrictions/caveats
  - Their memory storage is compiler and machine -dependent
  - Their memory address cannot be obtained.
  - Their size cannot be larger than an integer
- Example

```
struct campobit
                {
                int number;
                unsigned sevenbits:7;
                char letter
                } threeobjects;
threeobjects.sevenbits = data7b;
```

typedef

- `typedef` allows new datatype names:

```
typedef validdatatype newname;
```

- Examples:
  - `typedef short int age`
  - Particularly useful for short notation with structures

```
typedef struct military{
                        ...
                       } mranks;
mranks private, sergeant, lieutentant;
```