Introdução à Programação 2020/2021

António J. R. Neves
Daniel Corujo

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

an@ua.pt / dcorujo@ua.pt

http://elearning.ua.pt/

Topics

deti universidade de aveiro
departamento de eletrónica, telecomunicações e informática

- Enumerated Types
- Structs
- Bit fields
- Unions

- An enumeration is a user-defined data type that consists of integral constants
- The definition of an enumeration begins with the keyword **`enum`**, possibly followed by an identifier for the enumeration, and contains a list of the type's possible values, with a name for each value:

```
enum [identifier] { enumerator-list };
```

- The following example defines the enumerated type **`enum color`**:

```
enum color { black, red, green, yellow, blue, white, gray };
```

- The identifier **`color`** is the tag of this enumeration.
- The identifiers in the list are the enumeration constants, and have the type **`int`**.
- We can use these constants anywhere within their scope (ex. case constants in a switch statement).

- Each enumeration constant of a given enumerated type represents a certain value, which is determined either implicitly by its position in the list, or explicitly by initialization with a constant expression.
- A constant without an initialization has the value 0 if it is the first constant in the list, or the value of the preceding constant plus one.
- An enumerated type always corresponds to one of the standard integer types and it is possible to perform ordinary arithmetic operations with variables of enumerated types.
- Different constants in an enumeration may have the same value:

```
enum { OFF, ON, STOP = 0, GO = 1, CLOSED = 0, OPEN = 1 };
```

- In the original C and C++ enum types, the unqualified enumerators are visible throughout the scope in which the enum is declared
- In the second semestre (UC PpO) we will discuss scoped enums, where the enumerator name must be qualified by the enum type name
- Unscoped enum constants can be implicitly converted to int, but an int is never implicitly convertible to an enum value
- A cast is required to convert an int to a scoped or unscoped enumerator
- Scoped enumerators must be qualified by the enum type name (identifier) and cannot be implicitly converted

```
enum Colors1 {black, white};

enum class Colors2 {red, blue};

enum Colors1 c;

c = white;

cout << c << endl;

enum class Colors2 c2;

c2 = Colors2::blue;

cout << static_cast<int>(c2) << endl;
```

- The information manipulated by a program can sometimes be grouped together in objects (for example, a date may have three data items: month, day and year).
- In the C/C++ programming languages it is possible to group together zero or more data items in a so-called **`struct`**. The data items are the members (or fields) of the **`struct`**.
- A structure type is a type defined within the program that specifies the names and types of its members, and the order in which they are stored.
- Once you have defined a structure type, you can use it like any other type in declaring objects, pointers to those objects, and arrays of such structure elements.
  The members of a structure may have any desired complete type, including previously defined structure types.

- The definition of a structure type begins with the keyword **`struct`**, and contains a list of declarations of the structure's members, in braces:

```
struct tag_name { member_declaration_list };
```

- To be useful, a structure must contain at least one member. The following example defines the type **`struct Date`**, which has three members, all of type **`int`**:

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

- The identifier **`Date`** is this structure type's tag. The identifiers **`year`**, **`month`**, and **`day`** are the names of its members.
- A structure type cannot contain itself as a member, as its definition is not complete until the closing brace (**`}`**). However, structure types can, and often do, contain pointers to their own type.

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

- The **`struct Song`** type has five members, used to store five pieces of information about a music recording.
- Within the scope of a structure type definition, we can declare objects of that type.

```
struct Song song1, song2; // objects of type struct Song
```

- 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 song1, song2; // objects of type song_t (struct Song)
```

- 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.
- The left operand of the dot operator is an expression that yields a structure object.

```
Song_t song1, song2;      // the objects
Song_t *pSong1 = &song1; // a pointer to the first object
strcpy( song1.title, "Gruas" );
strcpy( song1.composer, "Rodrigo Leão" );
song1.published.year = 2015;

song1.duration = 4 * 60 + 26;
pSong1->duration = 4 * 60 + 26 // alternative
(*pSong1).duration = 4 * 60 + 26; // another syntax
pSong1[0].duration = 4 * 60 + 26; // another syntax

song2 = song1; // copy entire content of song1 to song2
```

- When we define structure objects without explicitly initializing them, the usual initialization rules apply:
  - if the structure object has automatic storage class, then its members have indeterminate initial values
  - if, on the other hand, the structure object has static storage duration, then the initial value of its members is zero
- To initialize a structure object explicitly when you define it, we must use an initialization list:

```
Song_t mySong = { "Diagrama",
                  "Rodrigo Leão",
                   "Rodrigo Leão",
                   297,
                  {9,26,2000} };
```

- We may also specify fewer initializers than the number of members. In this case, any remaining members are initialized to zero.

```
Song_t yourSong = { "Módulos" };
Song_t aSong = { .title = "Suspensos",
                 .composer = "Rodrigo Leão"};
```

- If a function parameter has a structure type, then the contents of the corresponding argument are copied to the parameter when you call the function

```
void printDate( struct Date d )
{
  std::cout << d.month << ", " << d.day << ", " <<  d.year;
}
```

- This approach can be rather inefficient unless the structure is small.
- Larger structures are generally passed by reference
- The function call copies only the address of a Song object, not the structure's contents
- We can also use pointers here...

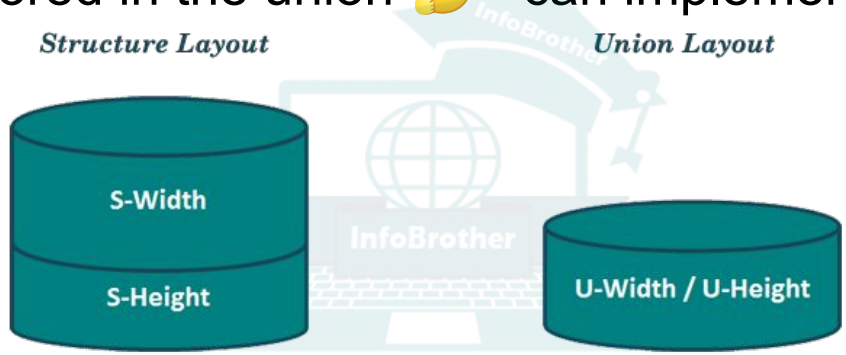
```
void printSong( const Song_t &pSong ) {
  std::cout << pSong->artist << ", " ... ;
}
```

- In order to be able to map the contents of, for example, hardware registers, or to pack more efficiently information, it is possible to specify, in a **struct**, an integer data type with a given number of bits.
- For example, in the declaration

```
struct rgb_color {
  unsigned int red   : 5; // 5 bits
  unsigned int green : 6; // 6 bits
  unsigned int blue  : 5; // 5 bits
};
```

the data type **struct rgb_color** has three integer fields, which can be packed in only 16 bits (5+6+5). The actual layout used (which field comes first), is compiler dependent.

- A union is a user-defined type in which all members share the same memory location
- This definition means that at any given time, a union can contain no more than one object from its list of members
- It also means that no matter how many members a union has, it always uses only enough memory to store the largest member
- A union can be useful for conserving memory when you have lots of objects and limited memory
- However, a union requires extra care to use correctly: you're responsible for ensuring that you always access the same member you assigned
- Take a look at union-like classes: it encloses the union in a struct, and includes an enum member that indicates the member type currently stored in the union 🤔 - can implement tagged unions



https://www.infobrother.com/Tutorial/C++/C++-Structure-and-Union.php

```
union RecordType{
    char   c;
    int    i;
    double d;
};

int main() {
    RecordType r;
    r.c = 'a';
    cout << r.c << endl;
    cout << r.d << endl; // undefined
    r.i = 5000;
    cout << r.i << endl;
    cout << r.d << endl; // undefined
    r.d = 5.5;
    cout << r.d << endl;
    cout << r.c << endl; // undefined
    return 0;
}
```

- An anonymous union is one declared without a class-name
- Names declared in an anonymous union are used directly, like nonmember variables - it implies that the names declared in an anonymous union must be unique in the surrounding scope

```
enum class WeatherDataType{
    Temperature, Wind
};

struct Input
{
    WeatherDataType type;
    union
    {
        int temp;
        double wind;
    };
};

Input first;
first.type = WeatherDataType::Temperature;
first.temp = 25;

Input second;
second.type = WeatherDataType::Wind;
second.wind = 5.5;
```