
# C Programming

## User Defined Data Types & Type Conversion

# User Defined Data Types - Structs & Unions

- Structures are a way to aggregate data together in order to
  - Define new data types based on existing data types
  - Make abstracted objects. E.g. *date, person etc.*
- A structure can be defined using the struct keyword

  struct [tag_name] { member_declaration_list };
  - The tag name is an identifier
    - In C, tag names have a different namespace from variables and functions. The compiler can distinguish tag names from the other identifiers. Therefore it is possible to have for example a variable with the same tag name of a struct.
  - The member variables are declared just like local variables
    - A struct shall have at least one member. A struct also makes a new namespace and its possible two have two members with the same name in two different structs

```c
struct Date
{
    unsigned char day;
    unsigned char month;
    unsigned short int year;
};

struct Person
{
    char name[32];
    unsigned char age;
    float height, weight;
    struct Date birthdate;
};
```

# User Defined Data Types - Structs & Unions

- The last member of a struct with more than one member may be a variable sized object.
  - But we should avoid using flexible members
- A member of a struct cannot be of type of the struct itself
- A struct variable can be declared like: struct *[tag_name]* variable_name;
- A struct variable can be initialized in different ways
  - When we declare the variable
    - struct Date date = {0}; // All the members are set to zero
    - struct Date date = {15, 2, 2021}; // Members shall be initialized in order
    - struct Date date = {.day = 15,.month = 2, .year=2021};// Initializing members in order using member designators
    - ***A member shall not be initialized more than once***
  - After declaration (using the **dot** operator)
    - struct Date date; date.day = 15; date.month = 2; date.year = 2021;

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

# User Defined Data Types - Structs & Unions

- A struct variable can be initialized partially, but in order
  - The other members are automatically set to zero
  - E.g. struct Date date = {15, .month = 2 }; // day is 15, month is 2 and year is 0
- To get access to the members of a struct variable the dot(.) operator is used
  - E.g. date.year = 2021; printf("Date D: %04d-%02d-%02d\n", date.day, date.month, date.year);
- It is possible to copy a struct variable to another one. They shall have the same type.
  - E.g. struct Date date1 = {15, 2, 2021}; struct Date date2 = date1;
- To get size of a type defined by a struct we can use the sizeof operator
  - The sizeof operator is evaluated in compile time.
  - E.g. printf("Size of Date in byte: %d\n", sizeof(struct Date));
- Getting size of a member of a struct during compilation is tricky: ***sizeof(((type *)0)->member)***
- It is possible to use typedef to make a new type of a struct. E.g. typedef struct Date date_t;

# User Defined Data Types - Structs & Unions

- A **struct variable** can have any storage class. E.g. static date_t date;
- A **struct member** can have any type qualifier. E.g. struct temp { const int a; float b; }
- **Bit-field**s are special type of structs which can be used to store data in a compact way
  - The members are declared like: ***type bitfield_name: width;***
  - **type** can only be signed int, unsigned int or _Bool.
    - Single-bit named bit fields shall not be of a signed type
  - **width** is the number of bits in the bit-field
  - Impossible to get address of a bit-field using **&** operator and order of bitfields is not guaranteed

```c
typedef struct
{
    uint32_t day : 5;
    uint32_t month : 4;
    uint32_t year : 23;
} date_t;
```

```c
    date_t bf_date = {.day = 15U, .month = 2U, .year = 2021U};

    printf("Bit-Field Date: %02d-%02d-%04d\n", bf_date.day, bf_date.month, bf_date.year);

#if 0
    printf("%p\n", &bf_date.month); // Impossible to get the address of a bit-field
#endif
```

# User Defined Data Types - Structs & Unions

- A **union** is a data type like struct in which all members share the same memory location
- A union can be defined using union keyword; E.g. union *[tag_name]* { member_declaration_list };
  - You can get size of a union and its members like a struct using sizeof
  - A union variable can be declared like *union tag_name variable_name;*
  - A union variable can be initialized like a struct using an initialization list
    - ***But you can have only one initializer and if you don't use a designator, the first member is initialized***
  - We can get access the same data in different ways; union Data data = {0x01020304U};
    - printf("Byte: 0x%02X, Word: 0x%04X, Double Word: 0x%08X\n", data.byte, data.word, data.dword);
  - It is possible to assign a union variable to another, but with the same type
    - union Data data1 = {.byte = 0x12u}; union Data data2 = data1;
  - It is possible to make a new name for a union type. E.g. typedef union Data data_t;
  - A **union variable** can have any storage class. E.g. static date_t date;
  - A **union member** can have any type qualifier. E.g. union temp { const int a; float b; }

```
union Data
{
    uint32_t dword;
    uint16_t words;
    uint8_t bytes;
};
```

# User Defined Data Types - Type Conversions

- Type casting means converting one data type into another one (type conversion)
- Type casting is required when operands of different types are combined in an operation

```
char a = 'A'; // The ascii code of A is 65
float b = 12.0f;
double c = a + b; // Implicit type casting
```

- In C there are two types of type casting
  - **Implicit** type casting which is done automatically by the compiler
  - **Explicit** type cast which is done using the *type casting operator* **()**
- Implicit type casting
  - Conversion of data types without changing the significance of the values stored inside the variable
  - Is done automatically by the compiler when type of operands mismatch.
  - The compiler promotes the lower types to higher compatible types according to some rules
    - _Bool, signed/unsigned char and short int ➤ int (done automatically)
    - int ➤ unsigned int ➤ long int ➤ unsigned long int ➤ long long int ➤ unsigned long long int (done if needed)
    - unsigned long long int ➤ float ➤ double ➤ long double (done if needed)

# User Defined Data Types - Type Conversions

- **Implicit** type conversion is also called as standard type conversion
- Converting smaller data types into larger data types is also called as **type promotion**.
- **Implicit** type casting also occurs also in
  - Assignments and initializations. The right side operand is converted to the type of the left side operand
  - Function calls. The passed arguments are converted to the types of the corresponding parameters
  - In return statements. The value of a return expression is converted to the function's return type.
- Implicit type casting of a data type to lower compatible data types in assignments
  - Can cause problems and values lose their meaning. For example:
    - High order bits may be lost when long int is converted to int or int to short int or int to char.
    - Fractional part will be truncated during conversion from floating point type (like double, float) to int type.
    - When the double type is converted to float type digits are rounded off and maybe we lose some accuracy.
    - When a signed type is changed to unsigned type, the sign may be dropped.
- ***Generally we should avoid implicit type conversion and we shall use explicit type casting***

# User Defined Data Types - Type Conversions

- Genelly implicit type conversions is one of the sources of bugs.
- Enable warnings during compilation
  - **gcc -Wall** … reports all the warnings. Even it is possible to convert warnings to errors using **-Werror**
  - **gcc -Wconversion** reports all the warnings regarding implicit conversions in the program
- **Explicit** type casting using the type cast operator is used to force type conversion

  ***(type_name) expression***

  - The type name is a standard data type.
  - The expression can be a constant, a variable or an actual expression.
  - E.g. int a = 'A'; char c = (char)a;
- We can discard an expression by casting it to void. i.e. (void) expression
- Sometimes we have to use explicit type casting. E.g. int a = 10; int b = 3; float c = (float)a / b;
- We can not convert type of a struct or union to a different type.

# User Defined Data Types - Type Conversions

- The value of an expression shall not be cast to an inappropriate essential type
- Casting from **void** to any other types is not permitted as it results in **undefined behaviour.**

| **Essential type category** | **from** | | | | | |
|---|---|---|---|---|---|---|
| **to** | Boolean | character | enum | signed integer | unsigned integer | floating-point |
| Boolean | | × | × | × | × | × |
| character | × | | | | | × |
| enum | × | × | × | × | × | × |
| signed integer | × | | | | | |
| unsigned integer | × | | | | | |
| floating-point | × | × | | | | |