

# Об'єднання, перелічувані типи та бітові поля



*Лекція №3*

*Дисципліна «Програмування»*

*2-й семестр*



# Об'єднання

**Об'єднання** – це тип, який дозволяє зберігати дані різних типів в одному і тому ж самому місці пам'яті (але не одночасно).

Застосовуючи масив об'єднань, можна створити масив одиниць однакових розмірів, кожна з яких може містити дані різних типів.

Шаблон об'єднання:

```
union Hold
{
    int    digit;
    double bigfl;
    char   letter;
};
```



# Ініціалізація об'єднань

Приклад визначення трьох змінних об'єднання типу **Hold**:

```
union Hold fit;           // змінна об'єднання типу Hold
union Hold save[10];     // масив з 10 змінних об'єднання
union Hold *pu;          // вказівник на змінну типу Hold
```

Варіанти ініціалізації об'єднання:

- ініціалізувати об'єднання іншим, що має такий самий тип;
- ініціалізувати перший елемент об'єднання;
- у випадку **c99** застосувати призначений ініціалізатор.



# Ініціалізація об'єднань

```
union Hold val_A;  
val_A.letter = 'R';  
  
// ініціалізація одного об'єднання іншим  
union Hold val_B = val_A;  
  
// ініціалізація члену digit об'єднання  
union Hold val_C = { 88 };  
  
// призначений ініціалізатор  
union Hold val_D = { .bigfl = 118.2 };
```



# Використання об'єднань

```
// у змінній fit зберігається 23 (використовується 2 байти)
fit.digit = 23;
// 23 очищено, 2.0 збережено (використовується 8 байтів)
fit.bigfl = 2.0;
// 2.0 очищено, 'h' збережено (використовується 1 байт)
fit.letter = 'h';
```

Операція «крапка» показує, який тип даних застосовується в даний момент. Можна використовувати операцію "->" з вказівниками на об'єднання в тому ж стилі, як це робилося з вказівниками на структури:

```
pu = &fit;
x = pu->digit;           // те ж саме, що й x = fit.digit
```

Нижче показано, як не слід робити:

```
fit.letter = 'A';
flnum = 3.02 * fit.bigfl;    // ПОМИЛКА!
```



# Використання об'єднань

```
struct Owner {
    char socsecurity[12];
    ...
};

struct Leasecompany {
    char name[40];
    char headquarters[40];
    ...
};

union Data {
    struct Owner owncar;
    struct Leasecompany leasecar;
};

struct Car_data {
    char make[15];
    int status;           // 0 - власник, 1 - взятий напрокат
    union Data ownerinfo;
    ...
}
```



# Анонімні об'єднання (C11)

```
struct Owner {
    char socsecurity[12];
    ...
};
struct Leasecompany {
    char name[40];
    char headquarters[40];
    ...
};
struct Car_data {
    char make[15];
    int status;           // 0 - власник, 1 - взятий напрокат
    union {
        struct Owner owncar;
        struct Leasecompany leasecar;
    };
    ...
};
```



# Перелічувані типи

Перелічуваний тип можна використовувати для оголошення символічних імен, що являють собою цілочислові константи.

Ключове слово **enum** дозволяє створити новий «тип» і вказати значення, які для нього допускаються.

Насправді константи **enum** мають тип **int**, тому їх можна застосовувати усюди, де дозволено використовувати тип **int**.

Метою перелічуваних типів є покращення читабельності програми.

```
enum Spectrum { RED, ORANGE, YELLOW, GREEN, BLUE, VIOLET };
enum Spectrum color;
int c;
color = BLUE;
if(color == YELLOW)
    ...;
for(color = RED; color <= VIOLET; color++)
    ...;
```





# Константи enum

## Стандартні значення

За замовчуванням константам у списку перелічувань присвоюється цілочислові значення 0, 1, 2 і т. д. Оголошення

```
enum Kids { NIPPY, SLATS, SKIPPY, NINA, LIZ };
```

призводить до того, що **NINA** має значення 3.

## Присвоєні значення

За бажанням можна обрати цілочислові значення, які повинні мати константи:

```
enum Levels { LOW = 100, MEDIUM = 500, HIGH = 2000 };
```

Подальші константи отримають значення, які послідовно зростають на 1:

```
enum Feline { CAT, LYNX = 10, PUMA, TIGER };
```

В цьому випадку **CAT** отримає стандартне значення с, а **LYNX**, **PUMA** і **TIGER** – відповідно, 10, 11 і 12.



# Використання enum

```
#include <stdio.h>
#include <windows.h>
#include <string.h>          // для strcmpn(), strchr()
#include <stdbool.h>         // заціб C99
#define LEN 30

char *s_gets(char *st, int n);

enum Spectrum { RED, ORANGE, YELLOW, GREEN, BLUE, VIOLET };

const char *colors[] = { "red",    "orange", "yellow",
                        "green", "blue",    "violet" };

int main(void)
{
    char choice[LEN];
    enum Spectrum color;
    bool color_is_found = false;

    SetConsoleOutputCP(1251);
```



# Використання enum

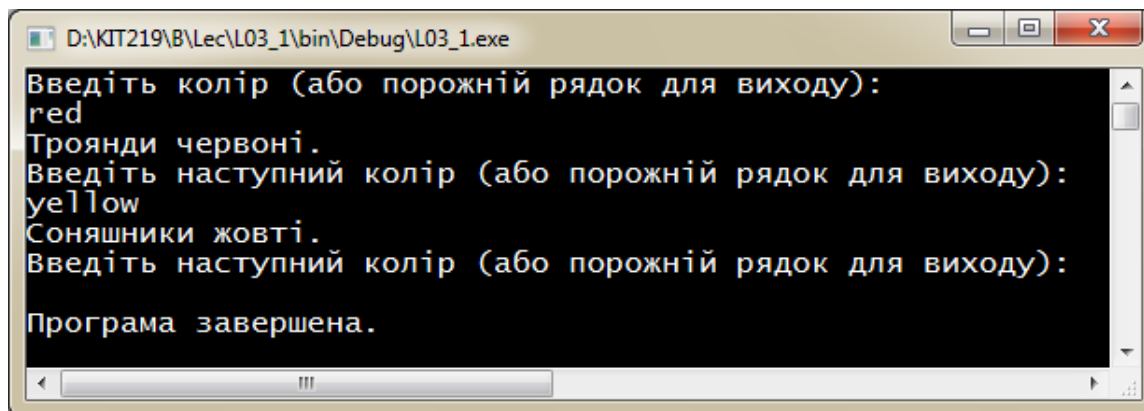
```
puts("Введіть колір (або порожній рядок для виходу):");
while(s_gets(choice, LEN) != NULL && choice[0] != '\0')
{
    for(color = RED; color <= VIOLET; color++)
    {
        if(strcmp(choice, colors[color]) == 0)
        {
            color_is_found = true;
            break;
        }
    }
    if(color_is_found)
    {
        switch(color)
        {
            case RED: puts("Троянди червоні.");
                     break;
            case ORANGE: puts("Маки оранжеві.");
                     break;
```



# Використання enum

```
    case YELLOW: puts("Соняшники жовті."); break;
    case GREEN: puts("Трава зелена."); break;
    case BLUE: puts("Дзвоники сині."); break;
    case VIOLET: puts("Фіалки фіолетові."); break;
                default: break;
        }
    }
    else
        printf("Колір %s не визначений.\n", choice);
    color_is_found = false;
    puts("Введіть наступний колір"
        " (або порожній рядок для виходу):");
}
puts("Програма завершена.");
return 0;
}
```

# Використання enum



```
D:\KIT219\B\Lec\L03_1\bin\Debug\L03_1.exe
Введіть колір (або порожній рядок для виходу):
red
Троянди червоні.
Введіть наступний колір (або порожній рядок для виходу):
yellow
Соняшники жовті.
Введіть наступний колір (або порожній рядок для виходу):

Програма завершена.
```



# Засіб typedef

Ключове слово **typedef** являє собою вдосконалений засіб маніпулювання даними, який дозволяє створювати власне ім'я для типу.

В цьому сенсі він подібний до директиви **#define**, але з трьома відмінностями:

- 1) На відміну від **#define**, засіб **typedef** обмежений призначенням символічних імен тільки типам, а не значенням.
- 2) Інтерпретація **typedef** виконується компілятором, а не препроцесором.
- 3) В рамках своїх обмежень засіб **typedef** є більш гнучким, ніж **#define**.



# Засіб typedef

```
typedef unsigned char BYTE;  
BYTE x, y[10], *z;
```

Деякі можливості **typedef** можна продублювати за допомогою **#define**. Наприклад, вказівка

```
#define BYTE unsigned char
```

змушує препроцесор замінювати **BYTE** типом **unsigned char**.

Наступний приклад **typedef** неможливо відтворити за допомогою **#define**:

```
typedef char * STRING;
```

Засіб **typedef** можна також використовувати зі структурами:

```
typedef struct Complex  
{  
    float real;  
    float imag;  
} COMPLEX;
```



# Зачіб typedef

```
typedef struct { double x; double y; } rect;
```

Припустимо, що визначений за допомогою **typedef** ідентифікатор застосовується так, як показано нижче:

```
rect r1 = { 3.0, 6.0 };  
rect r2;
```

Цей код транслюється на наступні оператори:

```
struct { double x; double y; } r1 = { 3.0, 6.0 };  
struct { double x; double y; } r2;  
r2 = r1;
```

Друга причина використання **typedef** пов'язана з тим, що імена **typedef** часто застосовуються для складених типів. Наприклад, оголошення

```
typedef char (* FRPTC() ) [5];
```

робить **FRPTC** ідентифікатором типу, який є функцією, що повертає вказівник на масив з 5 елементів **char**.





# Бітові поля

Одним з методів маніпулювання бітами є застосування бітових полів, які являють собою набір сусідніх бітів всередині значення типу **signed int** або **unsigned int**.

Стандарти C99 і C11 додатково дозволяють мати бітові поля типу **\_Bool**.

Бітове поле створюється шляхом оголошення структури, в якій зазначено кожне поле та визначено його розмір.

Наступне оголошення встановлює чотири однобітових поля:

```
struct
{
    unsigned int a1 : 1;           prnt.a1 = 0;
    unsigned int b1 : 1;           prnt.b1 = 1;
    unsigned int c1 : 1;
    unsigned int d1 : 1;
} prnt;
```



# Бітові поля

Часом налаштування передбачає більш ніж дві опції, тому для представлення всіх варіантів одного біта виявляється недостатньо.

```
struct
{
    unsigned int code1 : 2;
    unsigned int code2 : 2;
    unsigned int code3 : 8;
} prcode;
```

Цей код створить два двобітових поля та одне восьмибітове. Тепер можливі наступні присвоювання:

```
prcode.code1 = 0;
prcode.code2 = 3;
prcode.code3 = 102;
```



# Бітові поля

Структуру полів можна заповнити неіменованими проміжками з застосуванням ширини неіменованих полів. Використання неіменованого поля шириною 0 призводить до того, що наступне поле вирівнюється по наступній області цілочислового значення:

```
struct
{
    unsigned int field1 : 1;
    unsigned int          : 2;
    unsigned int field2 : 1;
    unsigned int          : 0;
    unsigned int field3 : 1;
} stuff;
```

В даному фрагменті між полями `stuff.field1` і `stuff.field2` є двобітовий проміжок, а поле `stuff.field3` зберігається в наступній області `int`.

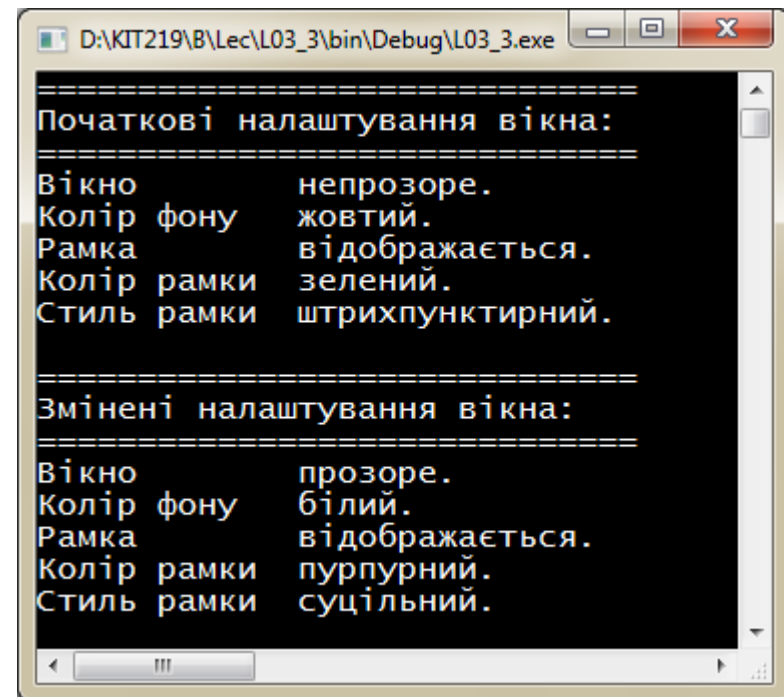
# Використання бітових полів

```
#include <stdio.h>
#include <windows.h>
#include <stdbool.h>    // C99, визначення bool, true, false

// стилі лінії
#define SOLID      0
#define DOTTED    1
#define DASHED     2

// основні кольори
#define BLUE       4
#define GREEN      2
#define RED        1

// змішані кольори
#define BLACK      0
#define YELLOW    ( RED   | GREEN )
#define MAGENTA   ( RED   | BLUE  )
#define CYAN      ( GREEN  | BLUE  )
#define WHITE     ( RED   | GREEN | BLUE )
```



```
D:\KIT219\B\Lec\L03_3\bin\Debug\L03_3.exe

=====
Початкові налаштування вікна:
=====
Вікно      непрозоре.
Колір фону  жовтий.
Рамка      відображається.
Колір рамки зелений.
Стиль рамки штрихпунктирний.

=====
Змінені налаштування вікна:
=====
Вікно      прозоре.
Колір фону  білий.
Рамка      відображається.
Колір рамки пурпурний.
Стиль рамки суцільний.
```



# Використання бітових полів

```
const char *colors[8] = {
    "чорний", "червоний", "зелений", "жовтий",
    "синій", "пурпурний", "блакитний", "білий" };

struct Box_props
{
    bool opaque                : 1; // або unsigned int (до C99)
    unsigned int fill_color    : 3;
    unsigned int               : 4;
    bool show_border          : 1; // або unsigned int (до C99)
    unsigned int border_color  : 3;
    unsigned int border_style  : 2;
    unsigned int               : 2;
};

void show_settings(const struct Box_props *pb);
```



# Використання бітових полів

```
int main(void)
{
    struct Box_props box =
        { true, YELLOW, true, GREEN, DASHED };
    SetConsoleOutputCP(1251);
    printf("=====\n");
    printf("Початкові налаштування вікна: \n");
    printf("=====\n");
    show_settings(&box);
    box.opaque          = false;
    box.fill_color      = WHITE;
    box.border_color    = MAGENTA;
    box.border_style    = SOLID;
    printf("\n=====\n");
    printf("Змінені налаштування вікна: \n");
    printf("=====\n");
    show_settings(&box);
    return 0;
}
```



# Використання бітових полів

```
void show_settings(const struct Box_props *pb)
{
    printf("Вікно          %s.\n",
           pb->opaque == true ? "непрозоре" : "прозоре");
    printf("Колір фону      %s.\n", colors[pb->fill_color]);
    printf("Рамка          %s.\n",
           pb->show_border == true ? "відображається"
                                   : "не відображається");
    printf("Колір рамки    %s.\n", colors[pb->border_color]);
    printf("Стиль рамки    ");
    switch (pb->border_style)
    {
        case SOLID: printf("суцільний.\n"); break;
        case DOTTED: printf("пунктирний.\n"); break;
        case DASHED: printf("штрихпунктирний.\n"); break;
        default: printf("невідомого типу\n"); break;
    }
}
```