МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«ХАРКІВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторних робіт
з дисципліни «Програмування»
для студентів денної та заочної форм навчання
за спеціальністю 123 «Комп’ютерна інженерія»

Частина 2

Затверджено
редакційно-видавничою
радою університету,
протокол №3 від 06.10.2021 р.

Харків
НТУ «ХПІ»
2021

Методичні вказівки до виконання лабораторних робіт з дисципліни «Програмування» для студентів денної та заочної форм навчання за спеціальністю 123 «Комп'ютерна інженерія». Ч. 2 / уклад. : Порошин С. М., Статкус А. В., Корольова Я. Ю., Онищенко В. В. – Харків: НТУ «ХПІ», 2021. – 183 с.

Укладачі: С. М. Порошин,
А. В. Статкус
Я. Ю. Корольова
В. В. Онищенко

Рецензент В. В. Усик

Кафедра мультимедійних інформаційних технологій і систем

ЗМІСТ

ВСТУП

Метою вивчення навчальної дисципліни «Програмування» є отримання навичок побудови основних обчислювальних алгоритмів, вивчення основ мови програмування високого рівня *C* та розробки комп'ютерних програм, ознайомлення з методами розв'язання обчислювальних задач та формування вмінь з їх практичного використання.

Під час проєктування програми одним з найбільш важливих кроків є обрання відповідного способу представлення даних. У багатьох випадках вибір звичайної змінної або навіть масиву може виявитися недостатнім. Мова *C* дозволяє розширити цей вибір за допомогою інших форм представлення даних, якими вона буде маніпулювати. Правильне обрання форми представлення даних може перетворити написання програми на дуже просту задачу.

Друга частина методичних вказівок до виконання лабораторних робіт з дисципліни «Програмування» призначена для надання студентам необхідної інформації при дослідженні можливостей використання структур, об'єднань та перелічених типів у програмах на *C*, основних функцій для роботи зі списком і деком а також при дослідженні принципів сортування списків.

Дане видання стане в нагоді при дослідженні особливостей роботи з функціями динамічного виділення пам'яті, керуванні окремими бітами при виконанні логічних операцій та операцій зсуву, розгляді засобів препроцесорної обробки, а також основних можливостей роботи з графічною бібліотекою WinBGIm та її функціями в інтегрованому середовищі *Code::Blocks*.

Лабораторна робота №1

ДОСЛІДЖЕННЯ ОСНОВНИХ МОЖЛИВОСТЕЙ ВИКОРИСТАННЯ СТРУКТУР В ПРОГРАМАХ НА *C*

Мета роботи: ознайомитися з основними можливостями мови *C* при роботі зі структурами; навчитися звертатися до полів структур та застосовувати масиви структур.

Теоретична частина

Структура – це сукупність різнотипних елементів, яким присвоюється одне ім’я, що займає одну ділянку пам’яті (може бути відсутнім). Елементи, що складають структуру, називаються полями [1–5].

Змінна типу «структура» (як і будь-яка інша змінна) повинна бути описана. Опис складається з опису шаблону (тобто складу полів) або типу структури й опису змінних структурного типу.

Синтаксис опису структури має такий вигляд:

```
struct [<ім'я_структури>]      // [ ] – означає, що це
{                              // необов’язковий елемент
    <тип_1> ім'я_поля_1;
    <тип_2> ім'я_поля_2;
    ...
} st1, st2, ...;
```

де **struct** – ключове слово; `<ім'я_структури>` – ім’я типу «структура» (може бути відсутнім); **<тип_1>, <тип_2>, ...** – імена стандартних або визначених типів; `ім’я_поля_1`, `ім’я_поля_2`, ... – імена полів структури; `st1`, `st2`, ... – імена змінних типу «структура».

Наприклад, для знаходження середнього балу, отриманого студентами в період зимової сесії з дисциплін «Вища математика», «Фізика» та «Програмування», визначимо таку структуру:

```
struct Student
{
    char   priz[20];               // прізвище та ініціали
```

```
    int    vish_mat, fiz, prog;     // дисципліни
    float  sb;                      // середній бал
} stud1, stud2;
```

Змінні **stud1** і **stud2** можна оголосити окремим оператором, наприклад:

```
struct Student stud1, stud2;
```

Ініціалізацію полів структури слід здійснювати під час її опису наступним чином:

```
struct Student
{
    char   priz[20];
    int    vysh_mat, fiz, prog;
    float  sb;
}
stud1 = { "Потапенко І.С.", 4, 5, 4 },
stud2 = { "Миколенко М.О.", 3, 4, 5 };
```

Якщо ініціалізація виконується у тілі програми, то для звернення до імені поля треба спочатку записати ім’я структурної змінної, а потім ім’я поля. Ці обидва записи відокремлюються крапкою та являють собою складене ім’я.

Отже, у випадку оголошення змінної **stud1** у програмі, для її ініціалізації можна записати

```
struct Student stud1 = { "Потапенко І.С.", 4, 5, 4 };
```

або присвоєння значень виконується за допомогою складених полів.

Розглянемо програму, в якій застосовується структура. Текст програми буде мати такий вигляд:

```
#include <stdio.h>
#include <string.h>
#include <windows.h>

int main(void)
{
    struct Student
    {
        char   priz[20];
        int    vysh_mat, fiz, prog;
        float  sb;
    } stud1, stud2;

    SetConsoleOutputCP(1251);
    strcpy(stud1.priz, "Потапенко І.С.");
```

```
    stud1.vysh_mat   = 4;
    stud1.fiz        = 5;
    stud1.prog       = 4;
    stud1.sb = (float)(stud1.vysh_mat + stud1.fiz +
                       stud1.prog)/3.0;
    stud2 = stud1;   //  Присвоювання структур
    printf("%-20s %2d %2d %2d %5.2f\n", stud2.priz,
           stud2.vysh_mat, stud2.fiz, stud2.prog, stud2.sb);
    return 0;
}
```

Результат виконання програми наведено на рис. 1.1.

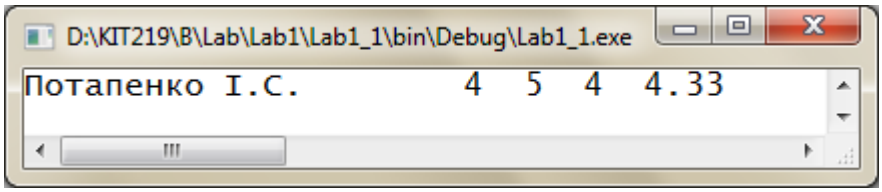


Рисунок 1.1 – Результат виконання програми, яка використовує структури

У наведеній програмі відбувається присвоювання всім полям структури **stud1** відповідних значень. Слід зауважити, що поле **stud1.priz** отримує значення шляхом застосування функції

```
strcpy(stud1.priz,"Потапенко I.C.");
```

Структурна змінна **stud2** має той самий тип, що і **stud1**, тому справедлива операція присвоювання

```
stud2 = stud1;
```

Якщо функція використовує тільки один структурний тип, то цей тип можна оголосити без імені. Тоді структуру, яка вже була розглянута раніше, можна оголосити таким чином:

```
struct
{
    char   priz[20];
    int    vysh_mat, fiz, prog;
    float  sb;
} stud1, stud2;
```

Структуру можна застосовувати при описі інших структур. Тобто поля структури можуть самі мати тип **struct**. Такі структури називаються вкладе-

ними. Їх можна використовувати, наприклад, якщо треба обробляти списки студентів та викладачів університету. Списки студентів містять такі дані: прізвище та ініціали, дата (день, місяць, рік) народження, група та середній бал успішності, а в списках викладачів присутні такі дані: прізвище та ініціали, дата народження, кафедра, посада. У процесі обробки списків студентів і викладачів можна оголосити відповідно такі структури:

```
struct Student
{
    char   priz[20];
    int    den;
    int    rik;
    char   mis[10];
    char   grup[10];
    float  sb;
};
struct Vykladach
{
    char   priz[20];
    int    den;
    int    rik;
    char   mis[10];
    char   kafedra[30];
    char   posada[20];
};
```

В оголошених типах однакові поля можна об’єднати в окрему структуру і застосовувати її при описі інших типів. Поетапно це виглядає таким чином:

– загальна структура –

```
struct Osoba
{
    char  priz[20];
    int   den;
    int   rik;
    char  mis[10];
};
```

– структура для опису інформації про студентів –

```
struct Student
{
    struct  Osoba dr;
    char    grup[10];
    float   sb;
} stud1, stud2;
```

– структура для опису інформації про викладачів –

```
struct Vykladach
{
    struct  Osoba dr;
    char    kaf[30];
    char    posada[20];
} vykl1, vykl2;
```

У структурах **Student** і **Vykladach** для оголошення поля, що містить дані про прізвище і дату народження, використовується раніше описаний тип **struct** Osoba. Тепер до полів **priz, den, rik, mis** можна звернутися, використовуючи запис stud1.dr.priz або vykl1.dr.priz, наприклад, при зверненні до функції вводу

```
gets(stud1.dr.priz);  або  gets(vykl1.dr.priz);
```

Після оголошення змінних типу «структура», для роботи з їх полями можна також застосовувати вказівники.

Тоді опис структури матиме такий вигляд:

```
struct Student
{
    char   priz[20];
    int    vysh_mat, fiz, prog;
    float  sb;
} stud1, *pst;
```

Доступ до полів може здійснюватися двома способами:

– з використанням операції розіменування **'*'**:

```
gets((*pst).priz);
(*pst).fiz = 5;
```

– з використанням вказівника **'->'**:

```
gets(pst->fam);
pst->fiz = 5;
```

Крім того, до полів змінної **stud1** можна звертатися, вказуючи поля за допомогою операції **'.'**, як це робилося раніше.

Дані типу «структура» можна об’єднувати в масиви. Тоді для попереднього прикладу (з урахуванням кількості студентів, яка дорівнює 15) структуру можна записати наступним чином:

```
struct Student
{
    char   priz[20];
    int    vysh_mat, fiz, prog;
    float  sb;
} spis[15], *sp = &spis[0];
```

У випадку, коли масив описується десь у тексті програми (не після опису структури), його можна оголосити наступним чином:

```
struct Student spis[15];
```

Тобто визначаємо масив типу «структура» з ім'ям **spis**, що містить відповідну інформацію про групу з 15 студентів.

Доступ до елементів масиву типу «структура» здійснюється із застосуванням індексу або вказівника-константи, яким є ім'я масиву, тобто одним з таких способів:

```
strcpy(spis[i].priz, " ");          // індекс елементу масиву
spis[i].fiz = 5;
```

або

```
strcpy((spis+i)->priz, " ");        // вказівник
(spis+i)->fiz = 5;
```

або

```
strcpy((*(spis+i)).priz, " ");      // ім'я масиву структур
(*(spis+i)).fiz = 5;
```

У останньому виразі `(*(spis+i)).fiz = 5;` необхідно застосовувати зовнішню пару дужок, тому що операція `'.'` має вищий пріоритет, ніж операція розіменування `'*'`.

Поля структури можуть також бути масивами. Наприклад, у розглянутій структурі **Student** можна оцінки з різних предметів об'єднати в масив. Тоді структуру слід описати наступним чином:

```
struct Student
{
    char   priz[20];
    int    dyst[3];
    float  sb;
} spis[15], *pst = &spis[0];
```

Звернення до полів здійснюватиметься одним з таких способів:

```
((*pst).priz)
(pst->dyst[0]),
```

наприклад,

```
gets((*pst).priz);
scanf("%d", pst->dyst[0]);
scanf("%d", pst->dyst[1]);
scanf("%d", pst->dyst[2]);
```

або

```
scanf("%d", pst->*(dyst+0));
scanf("%d", pst->*(dyst+1));
scanf("%d", pst->*(dyst+2));
```

Доцільно також зазначити, що заголовний файл **stdlib.h** містить спеціальні функції для пошуку та сортування структурних змінних.

Розглянемо приклади використання типу «структура».

Приклад №1. Напишемо *C*-програму, яка вводить відомість успішності групи студентів, які здали зимову сесію з дисциплін «Вища математика», «Фізика» та «Програмування», і обчислимо:

– середній бал кожного студента;

– середній бал групи з кожної дисципліни.

Виведемо на екран також прізвища відмінників з програмування.

Спочатку розглянемо перший варіант реалізації поставленої задачі, коли доступ до елементів масиву структур здійснюється за допомогою індексів. Текст програми має такий вигляд:

```
#include <stdio.h>
#include <string.h>
#include <windows.h>
int main(void)
{
    int smat;              // Сума оцінок з математики
    int sfiz;              // Сума оцінок з фізики
    int sprog;             // Сума оцінок з програмування
    int N;
    int pr = 0;            // Якщо pr == 1, то це ознака того, що є
                           // відмінники з програмування
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    printf("Введіть кількість студентів у групі: ");
```

```
scanf("%d", &N);
rewind(stdin);        // Очищуємо буфер обміну
struct Student
{
    char   priz[20];
    int    mat, fiz, prog;
    float  sb;
} vid[N];             // Масив студентів (відомість)
smat = sfiz = sprog = 0;
// Вводимо інформацію про успішність студентів
for(int i = 0; i < N; i++)
{
    printf("\nВведіть інформацію про %d-го студента\n", i+1);
    printf("Прізвище та ініціали:          ");
    gets(vid[i].priz);
    printf("Оцінки (матем., фіз., прогр.): ");
    scanf("%d %d %d", &vid[i].mat, &vid[i].fiz, &vid[i].prog);
    rewind(stdin);         // Очищення буферу обміну
    // Розрахунок середнього балу студента за сесію
    vid[i].sb = (float)(vid[i].mat + vid[i].fiz +
                        vid[i].prog)/3.0;
    if(vid[i].prog == 5) pr++;
    // Суми оцінок у групі з кожної дисципліни
    smat  += vid[i].mat;
    sfiz  += vid[i].fiz;
    sprog += vid[i].prog;
}
printf("\n\n            Результати зимової сесії           \n");
printf("===============================================\n");
printf("Студент             Математика Фізика Програмування\n");
printf("===============================================\n");
for(int i = 0; i < N; i++)
{
    printf("%-20s%6d%9d%11d\n", vid[i].priz,
           vid[i].mat, vid[i].fiz, vid[i].prog);
}
printf("===============================================\n");
printf("\nСередній бал з математики:     %4.2f\n",
       (float)smat/N);
printf("Середній бал з фізики:         %4.2f\n", (float)sfiz/N);
printf("Середній бал з програмування:  %4.2f\n", (float)sprog/N);
if(pr == 0)
    printf("\nВідмінників з програмування в групі немає.\n");
else
{
    printf("\nВідмінники з програмування:\n");
    for(int i = 0; i < N; i++)
    {
        if(vid[i].prog == 5)
            printf("%-20s\n", vid[i].priz);
    }
}
```

```
    printf("\n\n");
    return 0;
}
```

Результат виконання програми наведено на рис. 1.2.

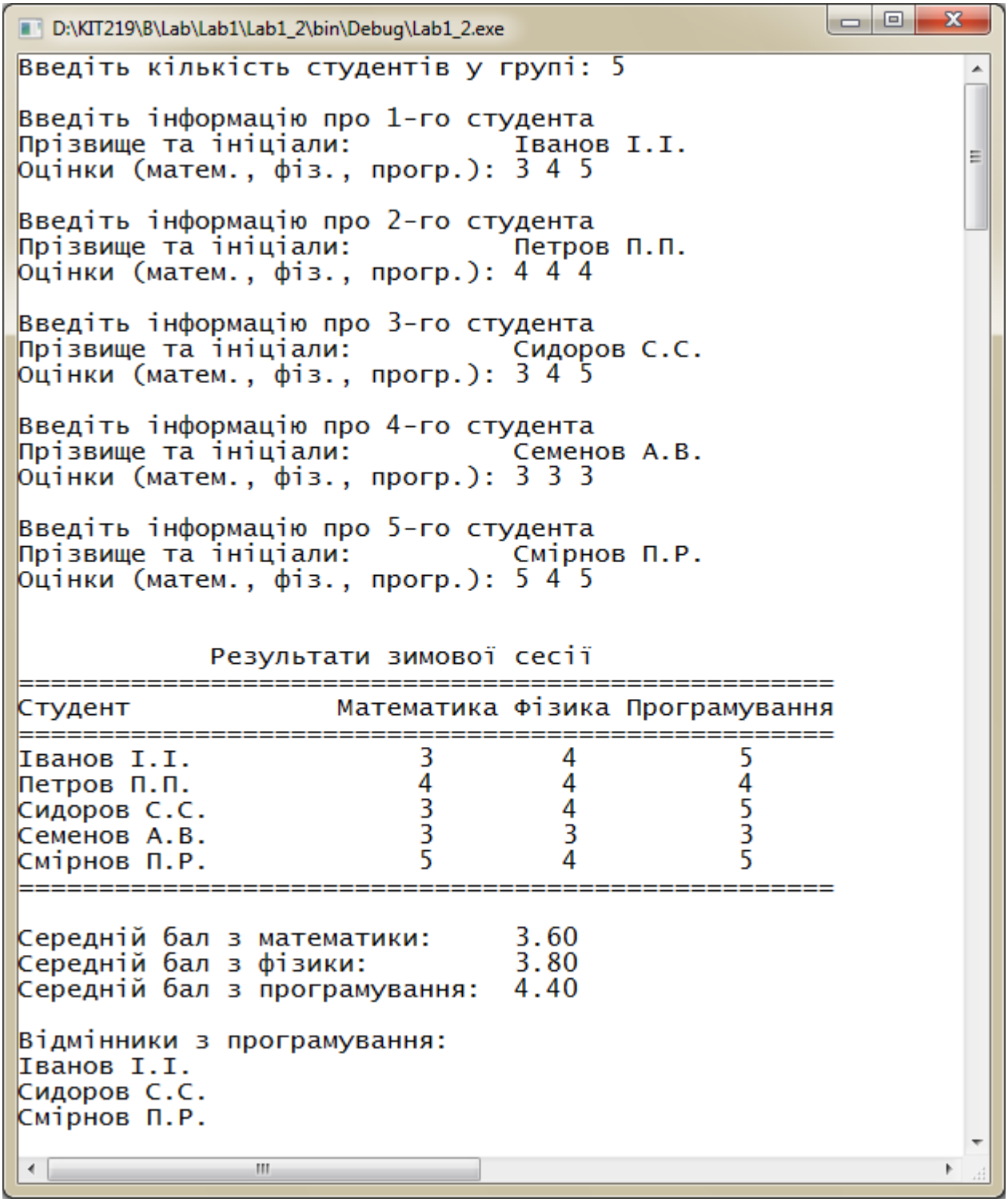


Рисунок 1.2 – Результат виконання програми, коли доступ до елементів масиву структур здійснюється за допомогою індексів

Приклад №2. Другий варіант передбачає використанням вказівників на структуру, яка містить поля оцінок з кожної дисципліни у вигляді масиву.

```
#include <stdio.h>
#include <string.h>
#include <windows.h>
#define KOL 3              // Кількість дисциплін

int main(void)
{
    int smat  = 0;         // Сума оцінок з математики
    int sfiz  = 0;         // Сума оцінок з фізики
    int sprog = 0;         // Сума оцінок з програмування
    int N;
    int pr = 0;            // Якщо pr == 1, то це ознака того, що є
                           // відмінники з програмування

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    printf("Введіть кількість студентів у групі: ");
    scanf("%d", &N);
    rewind(stdin);         // Очищуємо буфер обміну

    struct Student
    {
        char   priz[20];
        int    ocin[3];
        float  sb;
    } vid[N], *pst = &vid[0]; // pst — вказівник на масив оцінок

    // Вводимо інформацію про успішність студентів
    for(int i = 0; i < N; i++)
    {
        printf("\nВведіть інформацію про %d-го студента\n", i+1);
        printf("Прізвище та ініціали:          ");
        gets((*(pst+i)).priz);
        printf("Оцінки (матем., фіз., прогр.): ");
        (*(pst+i)).sb = 0.0;
        for(int j = 0; j < KOL; j++)
        {
            scanf("%d", &(*(pst+i)).ocin[j]);
            if( ((*(pst+i)).ocin[j] == 5) && (j == KOL-1) )
                pr++;
            (*pst).sb += (*(pst+i)).ocin[j];
        }
        rewind(stdin);         // Очищення буферу обміну
        // Розрахунок середнього балу студента за сесію
        (*pst).sb = (float)(*pst).sb / 3.0;
        // Суми оцінок у групі з кожної дисципліни
        smat  += (*(pst+i)).ocin[0];
        sfiz  += (*(pst+i)).ocin[1];
```

```
        sprog += (*(pst+i)).ocin[2];
    }
    printf("\n\n         Результати зимової сесії               \n");
    printf("================================================\n");
    printf("Студент          Математика   Фізика  Програмування \n");
    printf("================================================\n");
    for(int i = 0; i < N; i++)
    {
        printf("%-20s", (*(pst+i)).priz);
        for(int j = 0; j < KOL; j++)
        {
            printf("%11d", (*(pst+i)).ocin[j]);
        }
        printf("\n");
    }
    printf("================================================\n");
    printf("\nСередній бал з математики:     %4.2f\n",
           (float)smat/N);
    printf("Середній бал з фізики:         %4.2f\n", (float)sfiz/N);
    printf("Середній бал з програмування:  %4.2f\n", (float)sprog/N);
    if(pr == 0)
        printf("\nВідмінників з програмування в групі немає.\n");
    else
    {
        printf("\nВідмінники з програмування:\n");
        for(int i = 0; i < N; i++)
        {
            if((*(pst+i)).ocin[KOL-1] == 5)
                printf("%-20s\n", (*(pst+i)).priz);
        }
    }
    printf("\n\n");
    return 0;
}
```

Результат виконання програми наведено на рис. 1.3.

Слід нагадати, що запис виразу з вказівником, наприклад, `(*pst).priz`, рівнозначний запису `pst->priz`.

Наведемо приклад, в якому ініціалізація масиву типу «структура» здійснюється під час опису, а в процесі роботи з масивом використовується ім’я масиву як вказівник на масив.

Приклад №3. Припустимо, що необхідно ініціалізувати масив структур, який містить інформацію про монітори, а саме: назву монітору, термін гарантії та ціну. Потім слід відсортувати монітори за спаданням ціни та вивести відповідні повідомлення.

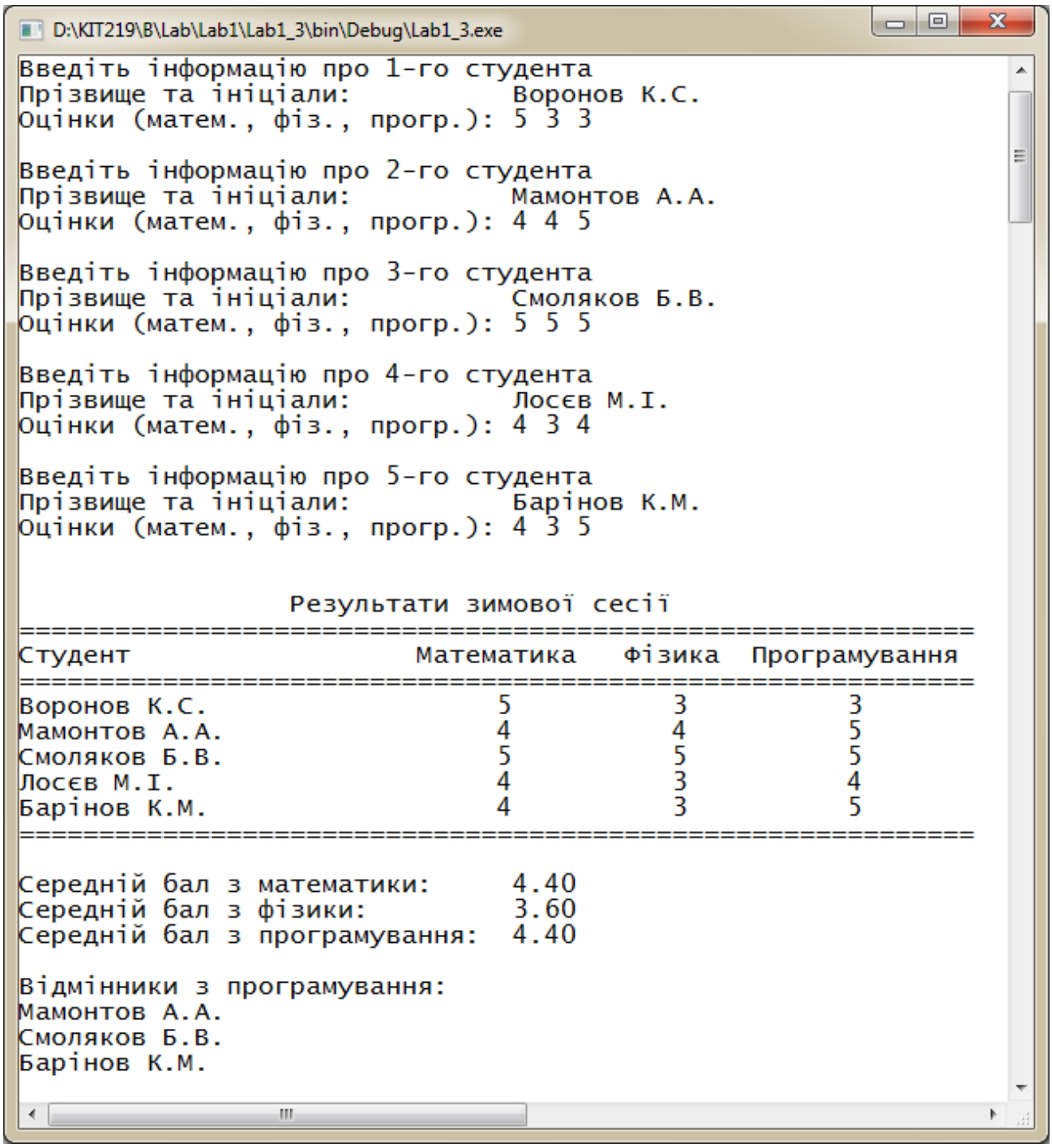


Рисунок 1.3 – Результат виконання програми, коли доступ до елементів масиву структур здійснюється за допомогою вказівників на структуру

Результат виконання програми наведено на рис. 1.4.

Текст програми має такий вигляд:

```
#include <stdio.h>
#include <string.h>
#include <windows.h>
#define N    6
```

```
#define KOL 20
int main(void)
{
    struct Monitor
    {
        char   name[KOL];  // марка монітору
        int    garant;     // гарантійний термін (місяців)
        double tsina;      // ціна
    } mon[N] = { { "Samsung 757 NF",    36,  1100.1 },
                 { "Sony CPD-C520K",    36, 11753.4 },
                 { "Hansol H95S TFT",   24,  4260.2 },
                 { "Samsung 757 DFX",   12,  1054.5 },
                 { "LG T710H Flatron",  36,   855.7 },
                 { "Samsung 192V TFT",  36,  4252.2 } };
    // pst – вказівник на масив моніторів
    struct Monitor *pst = &mon[0];
    int i, j, k;

    SetConsoleOutputCP(1251);
    printf("             Список моніторів                \n");
    printf("=============================================\n");
    printf("Назва монітору   Гарантія      Ціна          \n");
    printf("=============================================\n");
    for(i = 0; i < N; i++)
    {
        printf("%-20s", (*(pst+i)).name);
        printf("%4d", (*(pst+i)).garant);
        printf("%15.1f", (*(pst+i)).tsina);
        printf("\n");
    }
    printf("=============================================\n");
    //=============================================
    //  Сортування за зменшенням вартості
    //=============================================
    // rab – для перестановки елементів структури
    struct Monitor rab;
    for(i = N-1; i > 0; i--)
    {
        for(j = 0; j < i; j++)
        {
            if(mon[j].tsina < mon[j + 1].tsina)
            {
                for(k = 0; k < KOL; k++)
                    rab.name[k] = mon[j].name[k];
                rab.garant = mon[j].garant;
                rab.tsina = mon[j].tsina;
                for(k = 0; k < KOL; k++)
                    mon[j].name[k] = mon[j + 1].name[k];
                mon[j].garant = mon[j + 1].garant;
                mon[j].tsina = mon[j + 1].tsina;
                for(k = 0; k < KOL; k++)
                    mon[j + 1].name[k] = rab.name[k];
```

```
                mon[j + 1].garant = rab.garant;
                mon[j + 1].tsina = rab.tsina;
            }
        }
    }
    printf("\n\n");
    printf("   Сортування моніторів за спаданням ціни    \n");
    printf("=============================================\n");
    printf("Назва монітору    Гарантія      Ціна          \n");
    printf("=============================================\n");
    for(i = 0; i < N; i++)
    {
        printf("%-20s", (*(pst+i)).name);
        printf("%4d", (*(pst+i)).garant);
        printf("%15.1f", (*(pst+i)).tsina);
        printf("\n");
    }
    printf("=============================================\n");
    return 0;
}
```

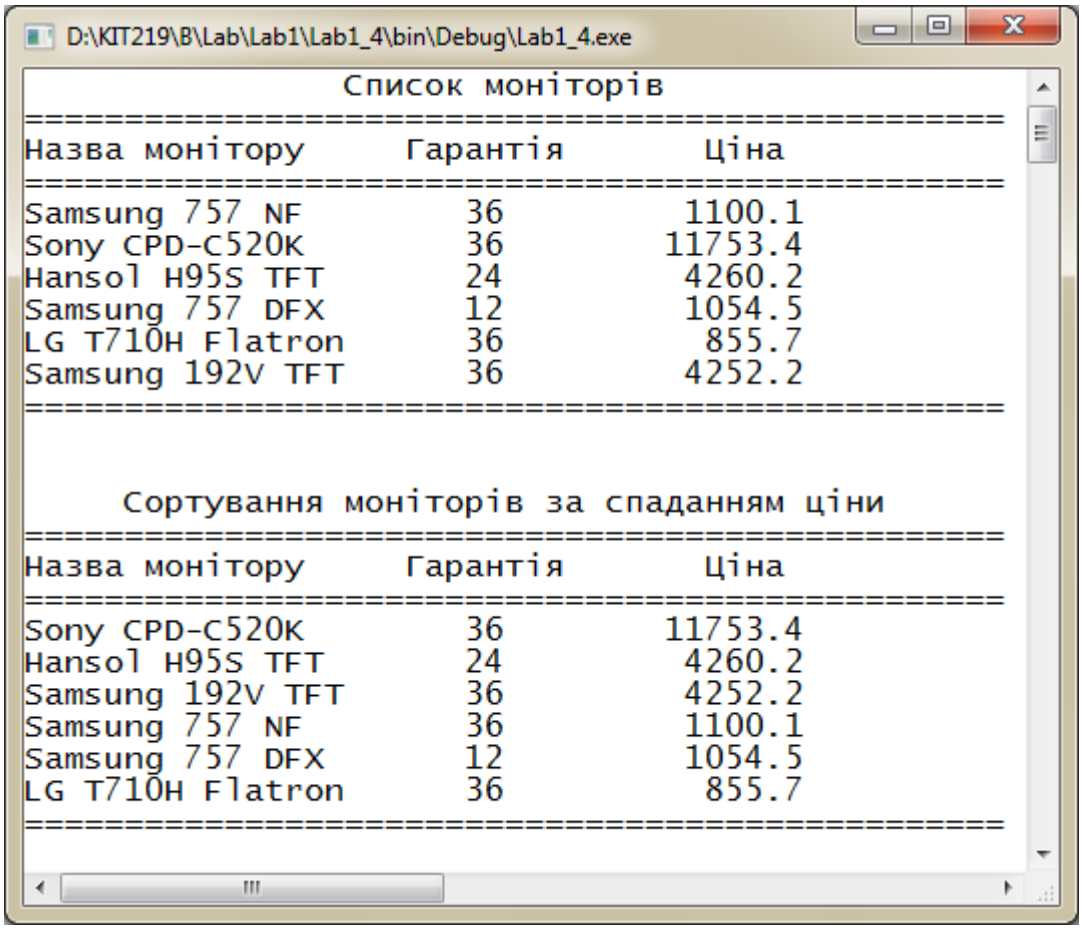


Рисунок 1.4 – Результат виконання програми, коли доступ до елементів масиву структур здійснюється за допомогою вказівників на масив структур

Приклад №4. Необхідно обробити результати зимової сесії в студентській групі, які містять інформацію про прізвища та ініціали, стать і середній бал успішності студентів. Вивести повідомлення про те, чий підсумковий середній бал вищий – хлопців або дівчат.

Програма використовує вказівник на масив структур. Для доступу до елементів структури застосовується операція `'->'`. Текст програми має такий вигляд:

```
#include <stdio.h>
#include <string.h>
#include <windows.h>
#define N 6

int main(void)
{
    struct Student
    {
        char priz[20];          // прізвище та ініціали
        char stat;              // стать (ч – чоловіча; ж – жіноча)
        float sb;               // середній бал
    } stud[N] = { { "Клімов В.С.",   'ч', 3.50 },
                  { "Самсонов М.К.", 'ч', 4.00 },
                  { "Денісова В.М.", 'ж', 5.00 },
                  { "Демідов С.Л.",  'ч', 3.90 },
                  { "Макарова М.Д.", 'ж', 4.20 },
                  { "Панов Т.Р.",    'ч', 4.90 } };

    // pst — вказівник на масив студентів
    struct Student *pst = &stud[0];

    // sbg, sbb — середній бал дівчат та юнаків відповідно
    float sbg = 0, sbb = 0;
    // kg, kb — лічильник кількості дівчат та юнаків у групі
    int i, kg = 0, kb = 0;

    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    /*================================================================
    // Даний фрагмент використовується замість ініціалізації масиву
    // структур, якщо початкові дані необхідно вводити з клавіатури
    for(i = 0; i < N; i++)
    {
        printf("\nВведіть інформацію про %d-го студента\n", i + 1);
        printf("Прізвище та ініціали:  ");
        gets((pst + i)->priz);
        printf("Введіть стать:         ");
        scanf("%c", &(pst + i)->stat);
        printf("Введіть середній бал:  ");
```

```
        scanf("%f", &(pst + i)->sb);
        rewind(stdin);          // Очищуємо буфер обміну
    }
    ============================================================*/

    printf("\n\n     Результати зимової сесії в групі     \n");
    printf("=============================================\n");
    printf("Студент                   Стать   Середній бал \n");
    printf("=============================================\n");
    for(i = 0; i < N; i++)
    {
        printf("%-20s", (pst + i)->priz);
        printf("%8c", (pst + i)->stat);
        printf("%13.2f", (pst + i)->sb);
        printf("\n");
    }
    printf("=============================================\n");
    for(i = 0; i < N; i++)
    {
        // визначення кількості дівчат у групі
        // та їх сумарного середнього балу
        if(((pst+i)->stat) == 'ж')
        {
            kg++;
            sbg += (stud+i)->sb;
        }
        else
        {
            kb++;
            sbb += (stud+i)->sb;
        }
    }
    printf("\n\n=============================================\n");
    printf("Середній бал дівчат:              %7.2f\n", sbg/kg);
    printf("Середній бал хлопців:             %7.2f\n", sbb/kb);
    printf("=============================================\n");
    if( sbg/kg > sbb/kb )
        printf("Середній бал дівчат більше.\n");
    else
    {
        if( sbg/kg < sbb/kb )
            printf("Середній бал хлопців більше.\n");
        else
            printf("Середній бал дівчат і хлопців збігається.\n");
    }
    printf("=============================================\n");
    return 0;
}
```

У рядках коментарів наведено інший спосіб запису звернення до полів структури для випадку, коли ім'я масиву використане як вказівник на масив.

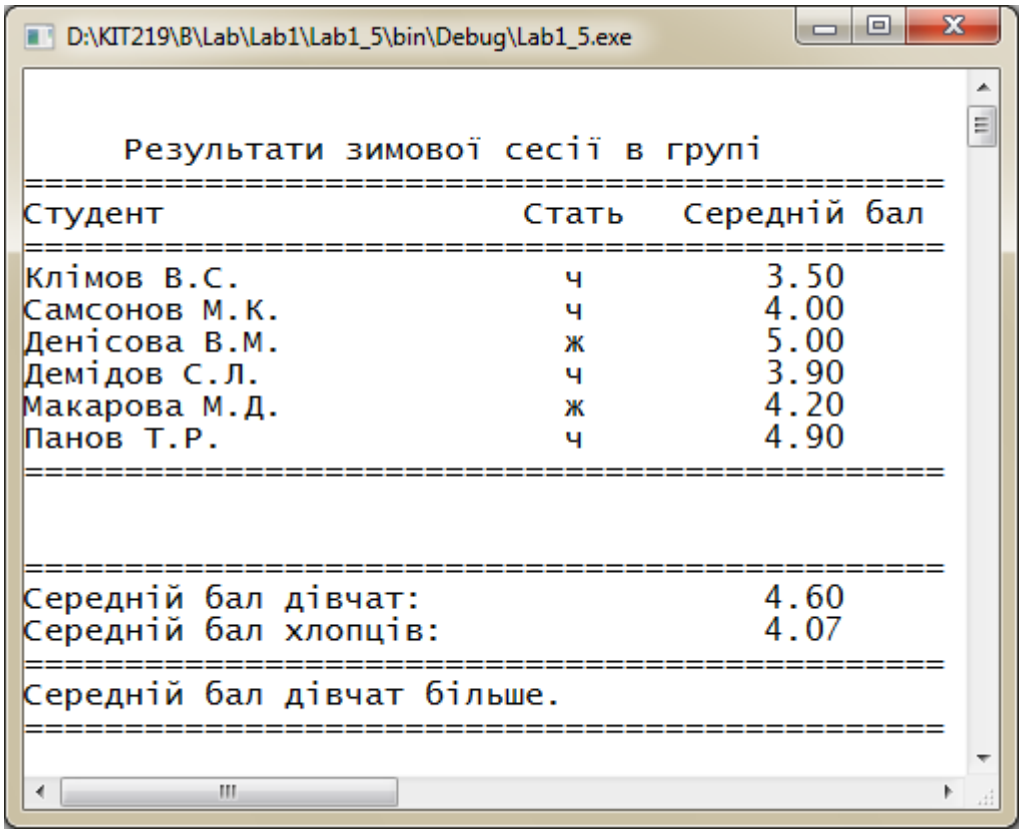


Рисунок 1.5 – Результат виконання програми, коли доступ до елементів структури в масиві структур здійснюється за допомогою операції **'->'**

Приклад №5. Розглянемо другий варіант розв'язання попереднього завдання з визначення середнього балу дівчат та хлопців, який містить змінну–вказівник на масив структур та функцію порівняння рядків `strcmp()`. При вводі інформації стать студента позначається рядками символів відповідно "жін." та "чол.". Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <string.h>
#define  N  6
int main(void)
{
    struct Student
    {
        char priz[20];
        char stat[5];
        float sb;
    };
    struct Student a[N], *p = &a[0];
    float sbg;                          // Середній бал для дівчини
```

```
    float sbb;                          // Середній бал для хлопця
    int kg, kb;
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    // Вводимо інформацію про студентів
    for(int i = 0; i < N; i++, p++)
    {
        printf("\nВведіть інформацію про %d-го студента\n", i+1);
        printf("Прізвище та ініціали: ");
        gets((*p).priz);
        printf("Стать (чол./жін.):    ");
        gets((*p).stat);
        printf("Середній бал:         ");
        scanf("%f", &((*p).sb));
        rewind(stdin);                  // Очищення буферу обміну
    }
    p -= N;   // повернення вказівника на початок масиву структур
    kg = kb = 0;
    sbg = sbb = 0.0;
    for(int i = 0; i < N; i++, p++)
    {   // функція strcmp() порівнює два рядка символів
        if(strcmp((*p).stat, "жін.")== 0)
        {
            kg++;
            sbg += (*p).sb;           // sbg += p->sb;
        }
        else
        {
            if(strcmp((*p).stat, "чол.")== 0)
            {
                kb++;
                sbb += (*p).sb;       // sbb += p->sb;
            }
        }
    }
    printf("\n\n=============================================\n");
    printf("Середній бал дівчат:              %7.2f\n", sbg/kg);
    printf("Середній бал хлопців:             %7.2f\n", sbb/kb);
    printf("=============================================\n");
    if(sbg/kg > sbb/kb)
        printf("Середній бал дівчат більше.\n");
    else
    {
        if(sbg/kg < sbb/kb)
            printf("Середній бал хлопців більше.\n");
        else
            printf("Середні бали дівчат і хлопців "
                   "збігаються.\n");
    }
    printf("=============================================\n");
    return 0;
}
```

У програмі для реалізації функції порівняння рядків `strcmp()` необхідно підключити заголовний файл **`string.h`**. Результат виконання програми наведено на рис. 1.6.

Хід виконання роботи:

1. Опрацюйте матеріал лекції «Структури» та теоретичний матеріал даної лабораторної роботи.

2. Відповідно до вашого варіанта (табл. 1.1) організуйте в *С*–програмах масиви структур для 1–го і 2–го завдань та ініціалізуєте їх необхідною інформацією. Масиви структур повинні складатися не менш ніж з 10 записів (рядків). Номери в табл. 1.1 відповідають номерам задач, які наведені в додатку 1.

Таблиця 1.1 – Номери задач для виконання 1–го і 2–го завдань

№ варіанта	1 завдання	2 завдання
1	23	3
2	18	5
3	8	25
4	9	17
5	15	23
6	1	21
7	5	24
8	6	15
9	16	7
10	19	24
11	17	8
12	4	13
13	14	23
14	11	26
15	2	12
16	3	22
17	24	5
18	12	7
19	15	25
20	7	17

3. В *С*–програмах для 1–го та 2–го завдань необхідно використовувати різні способи доступу до даних до членів структур.

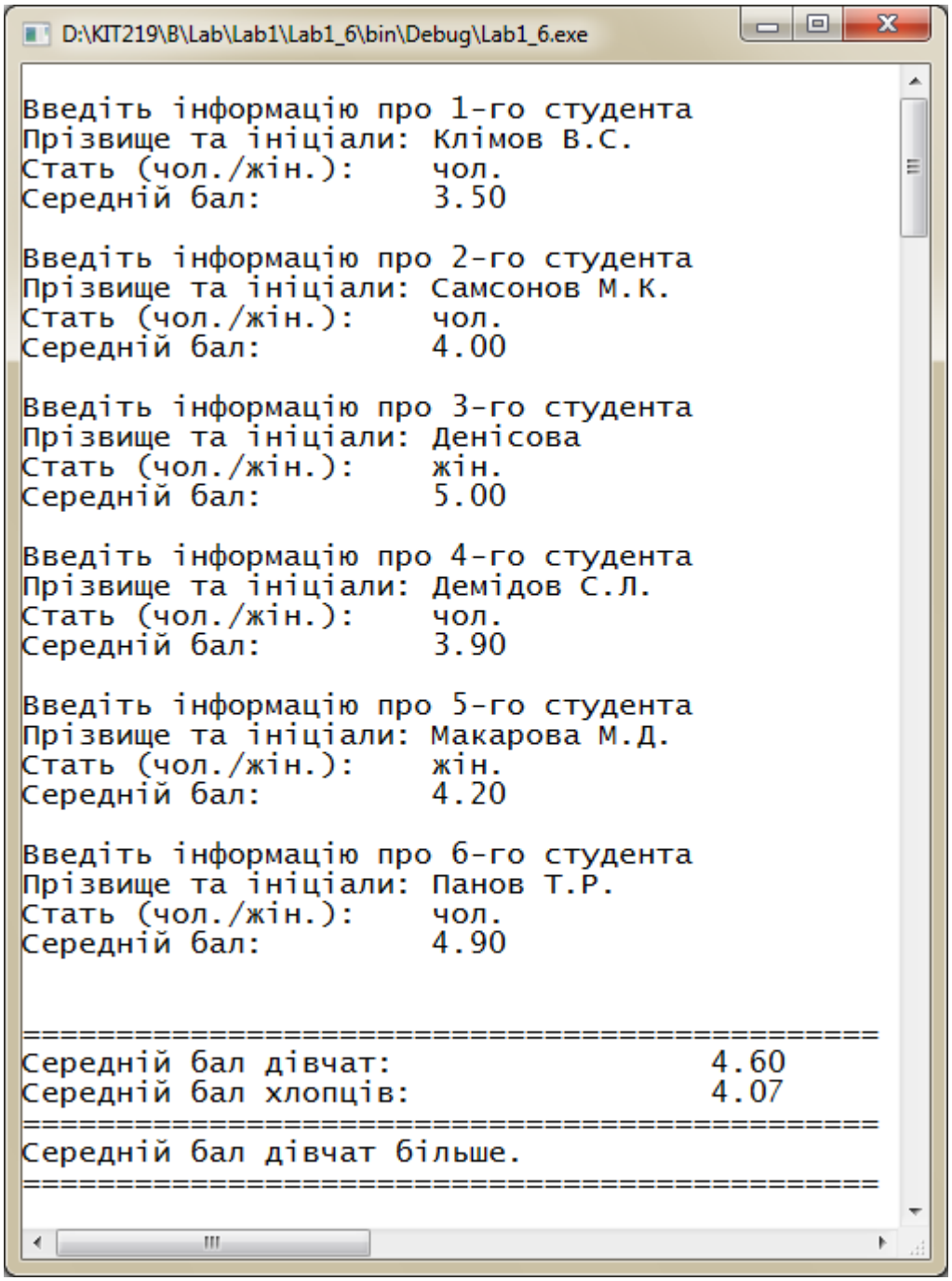


Рисунок 1.6 – Результат виконання програми, яка містить змінну–вказівник на масив структур та функцію порівняння рядків `strcmp()`

Звіт про виконання лабораторної роботи повинен містити:

1. Титульний аркуш.
2. Мету роботи.
3. Результати виконання роботи:
   – формулювання текстів завдання;
   – рисунки, що визначають структури баз даних (див. рис. Д 1.1);
   – тексти *С*–програм;
   – результати виконання *С*–програм (копії екранів).
4. Висновки по роботі.

Контрольні запитання:

1. Що таке структура?
2. З чого складається формат (шаблон, схема) структури?
3. Як визначити змінну типу «структура»?
4. Як ініціалізувати структуру?
5. Які існують способи доступу до членів структури?
6. Як визначити масив структур?
7. Що означає поняття «вкладені структури»? Які вкладені структури використовуються у вашій роботі?
8. Як оголосити та ініціалізувати вказівник на структуру?
9. Як здійснити доступ до членів структури за допомогою вказівника?
10. Чи можна присвоювати структури?

Лабораторна робота №2

ДОСЛІДЖЕННЯ ОСНОВНИХ МОЖЛИВОСТЕЙ ЗАСТОСУВАННЯ ОБ'ЄДНАНЬ, ПЕРЕЛІЧЕНИХ ТИПІВ ТА БІТОВИХ ПОЛІВ

Мета роботи: ознайомитися з основними можливостями мови *C* при роботі з об'єднаннями, переліченими типами та бітовими полями; навчитися оголошувати та ініціалізовувати об'єднання, звертатися до їх полів, використовувати об'єднання в програмах на *C*; отримати навички роботи з переліченими типами, які використовуються у якості полів структури, а також з бітовими полями.

Теоретична частина

2.1. Об'єднання

Об'єднання (union) – це похідний тип даних, подібний до структур, поля якого займають одну й ту ж саму область пам'яті [1].

В одних ситуаціях краще можуть підходити змінні одного типу, в інших – другого типу. Тому свого часу винайшли об'єднання, які дозволяють економно використовувати одну й ту ж саму область пам'яті для змінних різного типу в різних ситуаціях.

Поле об'єднання може мати будь–який тип даних. Кількість байтів, що використовується для зберігання екземпляру об'єднання, дорівнює розміру найбільшого поля.

В більшості випадків об'єднання містять декілька типів даних. В кожен конкретний момент звернутися можна тільки до одного типу даних. Забезпечення коректності використання типів в об'єднаннях повністю покладається на програміста. Звернення до даних в об'єднанні за допомогою змінної невідповідного типу є логічною помилкою. Результат операції отримання даних з об'єднання з іншим типом, що відрізняється від того, з яким вони були збережені, залежить від реалізації.

Використання об'єднань допомагає створювати машинно–незалежний

код. Оскільки компілятор відстежує справжні розміри змінних, які утворюють об'єднання, зменшується залежність від комп'ютера. Не слід турбуватися про розмір цілих або дійсних чисел, символів або ще про щось.

Об'єднання часто використовуються у разі необхідності перетворення типів, оскільки можна звертатися до даних, що зберігаються в об'єднанні абсолютно різними способами.

2.1.1. Оголошення об'єднань

Оголошення об'єднання відповідає тому ж самому формату, що й оголошення структури. Наступне визначення об'єднання

```
union Number     // визначення об'єднання Number
{
    int x;
    double y;
};
```

повідомляє, що **Number** – це об'єднання полів **int** x і **double** y. Зазвичай визначення об'єднань поміщають у заголовні файли, які підключаються в файлах з текстами *C*–програм, де використовуються ці об'єднання.

Як і визначення структур, визначення об'єднань просто створюють нові типи. Розміщення визначення об'єднання або структури за межами функцій не призводить до створення глобальних змінних.

2.1.2. Операції над об'єднаннями

Над об'єднаннями можуть виконуватися наступні операції:

– присвоювання одного об'єднання іншому, що має такий самий тип;

– взяття адреси (**&**) змінної–об'єднання;

– звернення до полів об'єднання за допомогою операторів «крапки» (**.**) і «стрілки» (**->**).

Об'єднання не можуть порівнюватися за допомогою (**==**) і (**!=**).

2.1.3. Ініціалізація об'єднань в оголошеннях

Для ініціалізації екземпляру об'єднання можна використовувати значення того ж самого типу, що й перше поле об'єднання. Наприклад, для об'єднання **Number** інструкція

```
union Number value = { 10 };
```

надає допустимий спосіб ініціалізації змінної–об’єднання, тому що вона ініціалізується значенням типу **int**.

В наступній інструкції відбудеться усічення дробової частини дійсного числа, що призведе до появи попередження компілятора:

```
union Number value = { 1.43 };
```

Об’єм пам’яті, який необхідний для зберігання об’єднання, залежить від реалізації, але він ніколи не буде меншим ніж найбільше поле об’єднання.

Деякі об’єднання не можна переносити безпосередньо в інші системи. Ця можливість часто залежить від правил вирівнювання, що застосовуються до типів даних, які входять до складу об’єднання в цій системі.

2.1.4. Приклади використання об’єднань

Приклад №1. Наступна програма використовує змінну **value** типу **union** Number для відображення значення, що зберігається в об’єднанні (як для значення типу **int**, так і для значення типу **double**). Результат виконання програми залежить від реалізації. Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>

union Number             // визначення об’єднання Number
{
    int x;
    double y;
};

int main(void)
{
    union Number value;  // визначаємо змінну–об’єднання

    SetConsoleOutputCP(1251);

    value.x = 100;       // зберігаємо ціле число в об’єднанні
    printf("Розміщуємо 100 в цілочисловому члені об'єднання\n");
    printf("та виводимо на консоль обидва типи.\n");
    printf("=============================================\n");
    printf(" int:      %d\n", value.x);
    printf(" double:   %f\n", value.y);
    printf("=============================================\n\n");
    value.y = 100.0;     // зберігаємо дійсне число в об’єднанні
```

```
    printf("Розміщуємо 100.0 в дійсному члені об'єднання\n");
    printf("та виводимо на консоль обидва типи.\n");
    printf("=============================================\n");
    printf(" int:      %d\n", value.x);
    printf(" double:   %f\n", value.y);
    printf("=============================================\n\n");
    return 0;
}
```

Результат виконання програми наведено на рис. 2.1. З отриманого виводу видно, що внутрішнє представлення значення типу **`double`** може суттєво відрізнятися від представлення значення типу **`int`**.

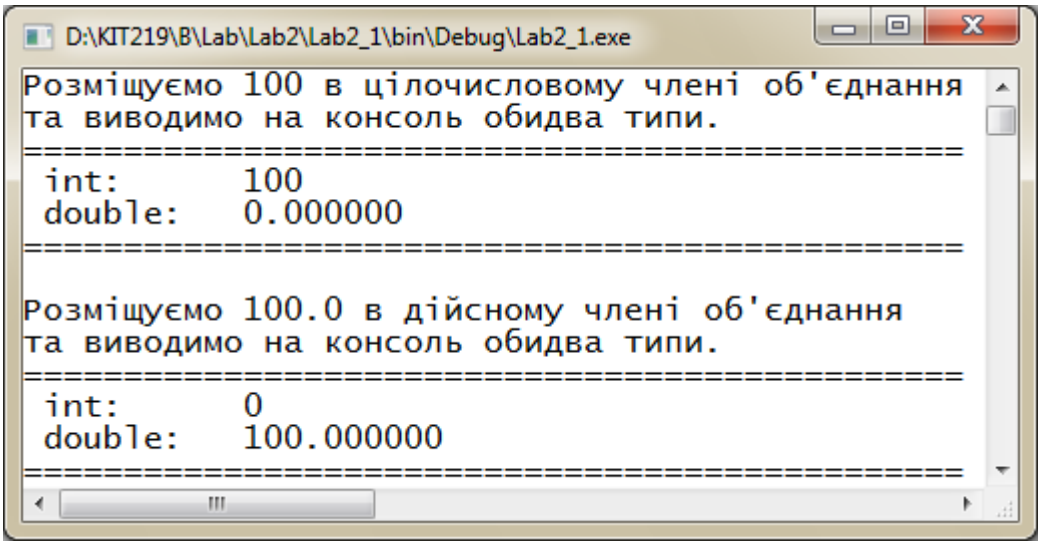


Рисунок 2.1 – Результат виконання програми, яка демонструє приклад використання об’єднання

Розглянемо питання запису цілого числа до файлу. Хоча до файлу можна записати будь–який тип даних (в тому числі й цілий) за допомогою функції `fwrite()`, для цієї операції використання `fwrite()` є не зовсім виправданим. Використовуючи об’єднання, можна легко створити функцію, що побайтно записує двійкове представлення цілого до файлу. Хоча існує декілька способів створення даної функції, є один спосіб виконання цього за допомогою об’єднання.

В наступному прикладі передбачається використання 32–бітних цілих. Об’єднання складається з одного цілого та 4–байтного масиву символів.

Приклад №2. Об’єднання дозволяє здійснювати доступ до окремих байтів, які утворюють ціле число, як до символів. В нашому випадку об’єднання **Pw** будемо використовувати для створення функції `write_int()`.

Текст програми буде мати наступний вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>

union Pw
{
    int i;
    char ch[4];
};

void write_int(int num, FILE *fp);

int main(void)
{
    FILE *fp;
    unsigned int i = 0;

    SetConsoleOutputCP(1251);

    fp = fopen("test.txt", "w");
    if(fp == NULL)
    {
        printf("Неможливо відкрити файл test.txt.\n");
        exit(1);
    }
    write_int(2916203245, fp);
    fclose(fp);
    fp = fopen("test.txt", "rb");
    fread(&i, sizeof(int), 1, fp);
    fclose(fp);
    printf("Значення змінної типу int: %u\n", i);
    printf("Розмір типу int: %d\n\n", sizeof(int));
    return 0;
}

// виводимо ціле число за допомогою об’єднання
void write_int(int num, FILE *fp)
{
    union Pw wrd;

    wrd.i = num;
    putc(wrd.ch[0], fp);     // перша частина числа
    putc(wrd.ch[1], fp);     // друга частина числа
    putc(wrd.ch[2], fp);     // третя частина числа
    putc(wrd.ch[3], fp);     // четверта частина числа
}
```

Результат виконання програми наведено на рис. 2.2. Вміст файлу **`test.txt`** наведено на рис. 2.3.

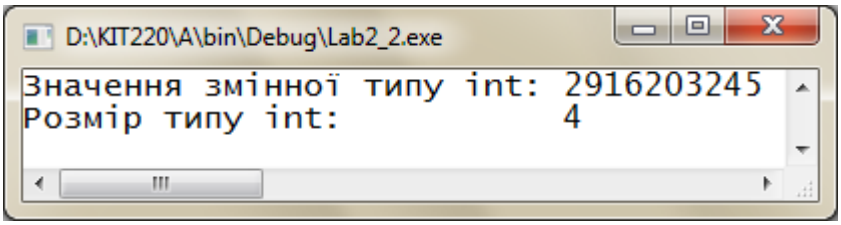


Рисунок 2.2 – Результат виконання програми, яка демонструє приклад використання об’єднання для доступу до окремих байтів

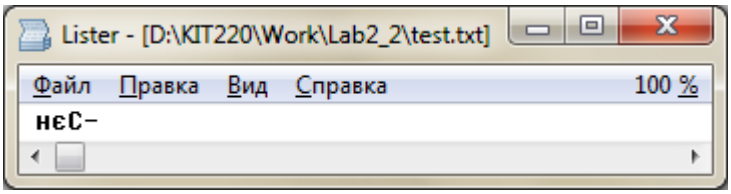


Рисунок 2.3 – Вміст файлу `test.txt`

Хоча функція `write_int()` і викликається з цілим параметром, вона використовує об’єднання для запису усіх частин цілого до файлу побайтно. Щоб переконатися в цьому оберемо для файлу **`test.txt`** шістнадцяткове представлення (рис. 2.4).

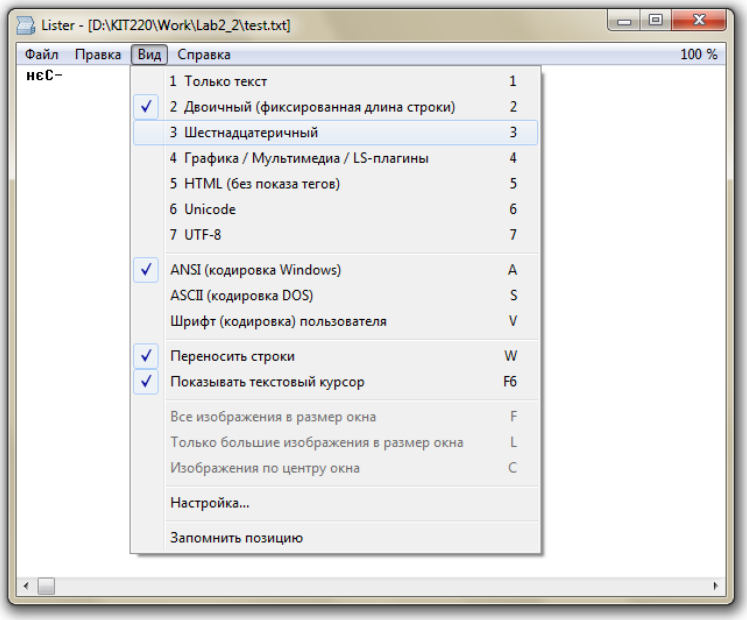


Рисунок 2.4 – Вибір шістнадцяткового представлення вмісту файлу `test.txt`

Тепер вміст файлу **`test.txt`** змінився (рис. 2.5). Нас цікавить інформація, яка розміщується в прямокутнику. В ньому наведено результат виводу чотирьох пар шістнадцяткових цифр, які виводилися побайтно.

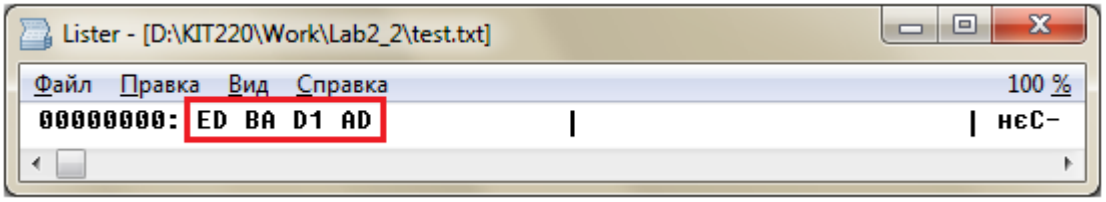


Рисунок 2.5 – Вміст файлу `test.txt`

Тепер скористаємося програмою «Калькулятор» в режимі «Програміст» і введемо в десятковій системі числення число 2916203245, яке застосовувалося в розглянутій програмі (рис. 2.6).

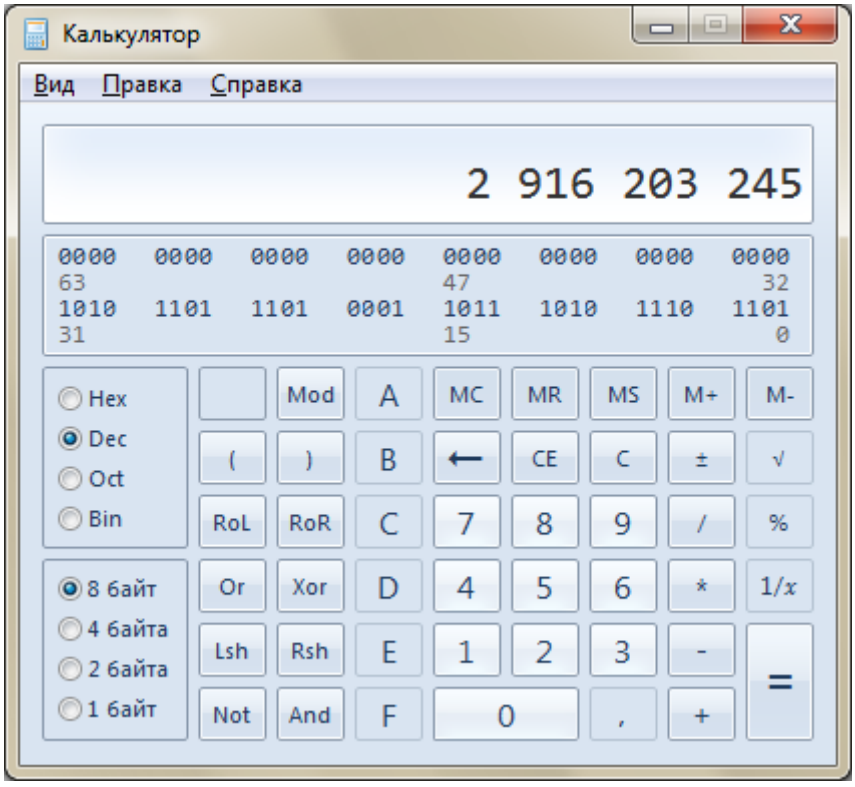


Рисунок 2.6 – Число `2916203245`, яке було введене в програмі «Калькулятор»

Подивимось як буде виглядати це число у двійковій системі числення з урахуванням того, що воно займає 4 байти (рис. 2.7).

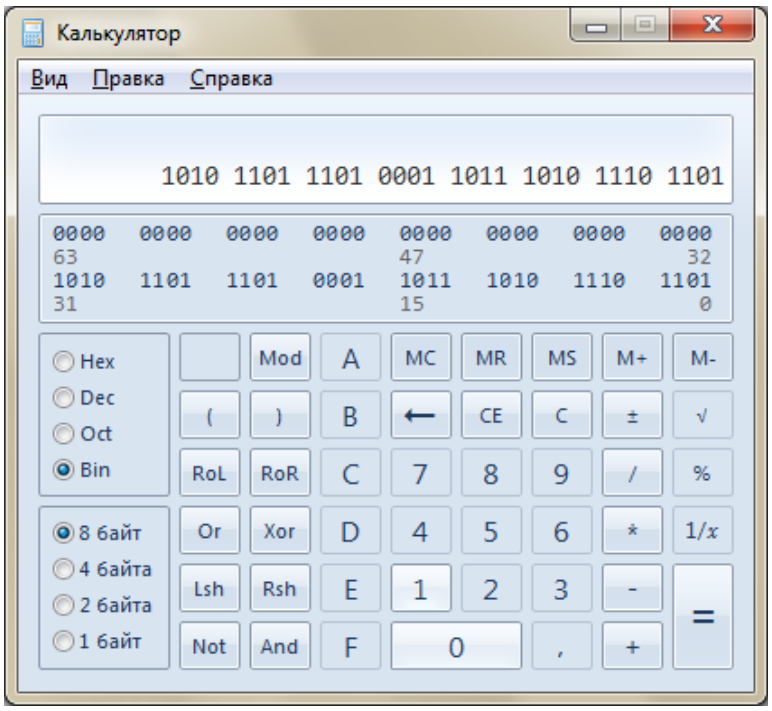


Рисунок 2.7 – Число 2916203245 в двійковій системі числення

Якщо обрати шістнадцяткову систему числення (рис. 2.8), то можна побачити, що це саме та інформація, яка була розміщена в прямокутнику на рис. 2.5, але на відміну від неї байти числа розташовані у зворотному порядку.

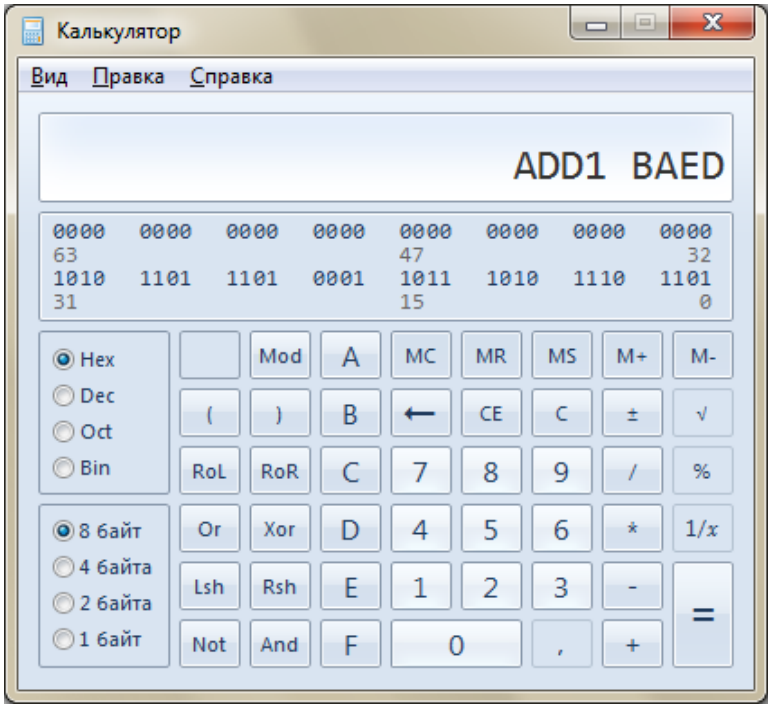


Рисунок 2.8 – Число 2916203245 в шістнадцятковій системі числення

2.2. Перелічені типи

Перелічені типи визначаються за допомогою ключового слова **enum** і є множинами цілочислових констант перелічень, що представляються ідентифікаторами [1]. Значення констант в переліченнях починаються з 0, якщо явно не зазначено інше, і кожній наступній константі присвоюється значення, більше ніж попереднє на 1. Наприклад, перелічення

```
enum Months // визначення перелічення Months
{
    JAN, FEB, MAR, APR,
    MAY, JUN, JUL, AUG,
    SEP, OCT, NOV, DEC
};
```

створить новий тип **enum** Months, в якому ідентифікатори отримають послідовні значення від 0 до 11 включно.

Щоб пронумерувати місяці числами від 1 до 12, можна скористатися наступним переліченням:

```
enum Months
{
    JAN = 1, FEB, MAR, APR,
    MAY,     JUN, JUL, AUG,
    SEP,     OCT, HOV, DEC
};
```

Оскільки першій константі перелічення явно присвоєно значення 1, інші константи будуть отримувати значення, які послідовно збільшуються, аж до 12. Ідентифікатори в переліченнях повинні бути унікальними. Значення кожної константи в переліченні можна встановити явно (у визначенні перелічення), шляхом присвоювання необхідного значення ідентифікатору. Одне й те ж саме значення може бути присвоєно декільком членам перелічення.

Приклад №3. В даному прикладі використовується змінна **month** типу «перелічення» в циклі **for** для виводу назви місяців року з масиву **monthName.** Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
```

```
// перелічення констант, що являють собою номери місяців року
enum Months
{
    JAN = 1, FEB, MAR, APR,
    MAY,     JUN, JUL, AUG,
    SEP,     OCT, NOV, DEC
};

int main(void)
{
    enum Months month = 0;

    // ініціалізація масиву вказівників
    const char *monthName[] =
    {
        "", "Січень", "Лютий",
        "Березень", "Квітень", "Травень",
        "Червень", "Липень", "Серпень",
        "Вересень", "Жовтень", "Листопад",
        "Грудень"
    };

    SetConsoleOutputCP(1251);
    for(int i = 0; i < 12; i++)
    {
        month++;
        printf("%2d %-11s\n", month, monthName[month]);
    }
    return 0;
}
```

Результат виконання програми наведено на рис. 2.9.

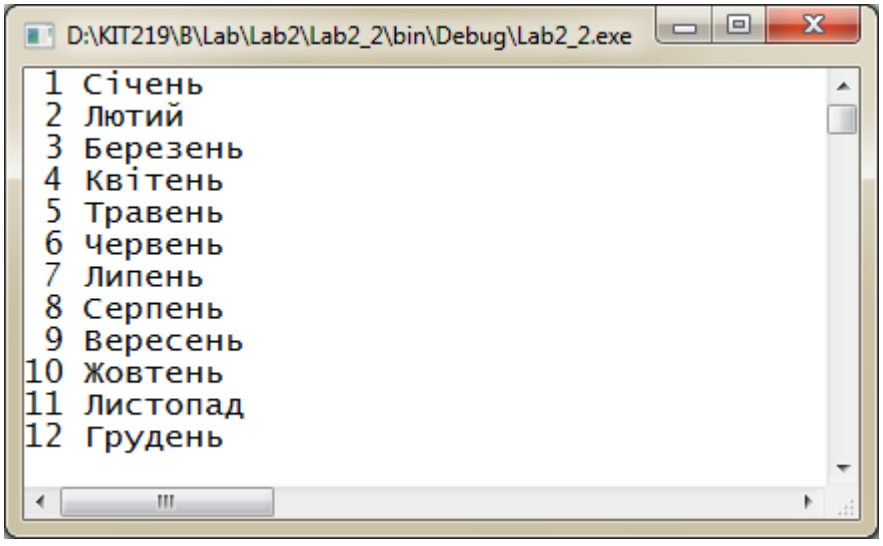


Рисунок 2.9 – Результат виконання програми, яка демонструє приклад використання перелічення

В елемент **monthName**[0] був поміщений пустий рядок "". Можна помістити в нього такий рядок, як "***ERR0R***", щоб одразу помітити появу логічної помилки.

Для іменування констант в переліченнях використовуйте тільки букви верхнього регістру. Це зробить константи більш помітними в тексті програми та буде нагадувати, що константи не є змінними.

Приклад №4. В даному прикладі використовуються два перелічення, які є полями структури. Необхідно написати програму, що використовує логічну функцію Kick(c1, c2, cs), яка перевіряє, чи б'є карта **c1** карту **c2**, з урахуванням того, що масть **cs** є козирною. Текст програми має такий вигляд:

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

struct Card
{
    enum { SPADES, CLUBS, DIAMONDS, HEARTS } suit;
    enum {
           SIX,  SEVEN, EIGHT, NINE, TEN,
           JACK, QUEEN, KING,  ACE
    } value;
} c1, c2;

int cs; // Козирна карта
bool result;

// Функція перевіряє чи б'є карта 1 карту 2
bool Kick(struct Card c1, struct Card c2, int cs)
{
    if(c1.suit == c2.suit)    // Якщо масті однакові
        if(c1.value > c2.value)
            return true;
        else
            return false;
    else // Якщо масті не однакові
    {
        if(c1.suit == cs)    // Якщо це козирна карта
            return true;
        else
            return false;
    }
}

void Message(struct Card c)
{
```

```
    switch(c.value)
    {
        case   SIX: printf("6 ");
                    break;
        case SEVEN: printf("7 ");
                    break;
        case EIGHT: printf("8 ");
                    break;
        case  NINE: printf("9 ");
                    break;
        case   TEN: printf("10 ");
                    break;
        case  JACK: printf("валет ");
                    break;
        case QUEEN: printf("дама ");
                    break;
        case  KING: printf("король ");
                    break;
        case   ACE: printf("туз ");
                    break;
           default: break;
    }
    switch(c.suit)
    {
        case   SPADES: printf("пік");
                       break;
        case    CLUBS: printf("треф");
                       break;
        case DIAMONDS: printf("бубей");
                       break;
        case   HEARTS: printf("чірва");
                       break;
              default: break;
    }
}
void trump_card(int cs)
{
    printf("Козирна масть: ");
    switch(cs)
    {
        case 0: printf("піки");
                break;
        case 1: printf("трефи");
                break;
        case 2: printf("бубни");
                break;
        case 3: printf("чирви");
                break;
       default: break;
    }
    printf("\n");
}
```

```
int main(void)
{
    SetConsoleOutputCP(1251);

    // Карта 1 не б'є карту 2
    c1.suit  = DIAMONDS;
    c1.value = SEVEN;
    c2.suit  = DIAMONDS;
    c2.value = NINE;
    cs       = CLUBS;

    result = Kick(c1, c2, cs);

    trump_card(cs);
    Message(c1);
    if(result) printf(" б'є ");
    else printf(" не б'є ");
    Message(c2);
    printf("\n=========================================\n");

    // Карта 1 б'є карту 2
    c1.suit  = DIAMONDS;
    c1.value = JACK;
    c2.suit  = DIAMONDS;
    c2.value = NINE;
    cs       = CLUBS;

    result = Kick(c1, c2, cs);

    trump_card(cs);
    Message(c1);
    if(result) printf(" б'є ");
    else printf(" не б'є ");
    Message(c2);
    printf("\n=========================================\n");

    // Карта 1 б'є карту 2
    c1.suit  = CLUBS;
    c1.value = SEVEN;
    c2.suit  = DIAMONDS;
    c2.value = ACE;
    cs       = CLUBS;

    result = Kick(c1, c2, cs);

    trump_card(cs);
    Message(c1);
    if(result) printf(" б'є ");
    else printf(" не б'є ");
    Message(c2);
    printf("\n=========================================\n");
    return 0;
}
```

Результат виконання програми наведено на рис. 2.10.

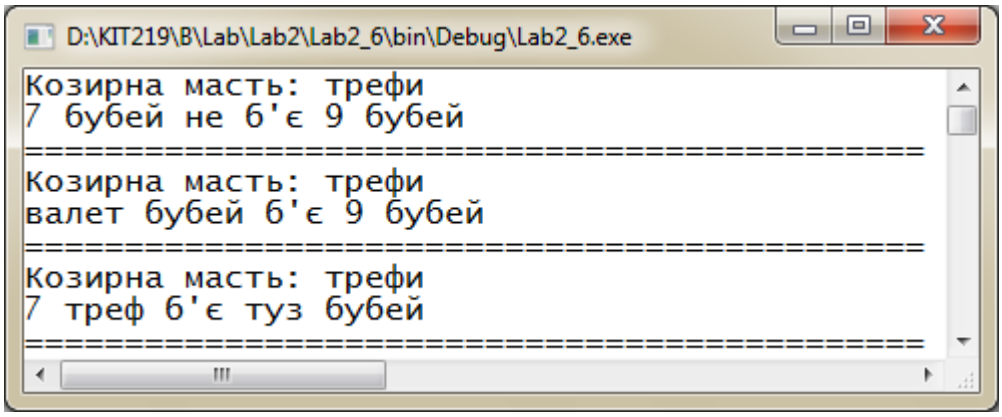


Рисунок 2.10 – Результат виконання програми, яка демонструє приклад використання двох перелічень у якості полів структури

2.3. Бітові поля

Мова *C* дозволяє вказувати кількість бітів, що займають поля типу **`unsigned int`** або **`int`** в структурі або об'єднанні [1]. Такі поля називаються бітовими полями. Вони забезпечують більш економне використання пам'яті для зберігання даних.

Наступне визначення структури

```
struct BitCard
{
    unsigned int face  : 4;
    unsigned int suit  : 2;
    unsigned int color : 1;
};
```

містить три бітових поля типу **`unsigned int`** (**`face`**, **`suit`** і **`color`**), що використовуються для представлення колоди з 52 карт. Бітове поле визначається двокрапкою (`:`), що йде після імені поля, та цілочисловою константою, яка визначає ширину поля в бітах. Значення ширини повинне бути цілим числом в діапазоні від 0 до загальної кількості бітів у значеннях типу **`int`** включно. Далі розглядаються приклади, де цілі числа займають в пам'яті 4 байти (32 біти).

Попереднє визначення структури вказує, що поле **`face`** займає 4 біти, поле **`suit`** займає 2 біти і поле **`color`** займає 1 біт. Кількість бітів була визначена з діапазону можливих значень для кожного поля в структурі. Поле **`face`** повинне зберігати значення в діапазоні від 0 (Ace – Туз) до 12 (King – Король) – 4 бітами

можна представити значення в діапазоні 0–15. Поле **suit** повинне зберігати значення від 0 до 3 (0 – Diamonds (бубни), 1 – Hearts (чирви), 2 – Clubs (трефи), 3 – Spades (піки)) – 2 бітами можна представити значення в діапазоні 0–3. Нарешті, поле **color** повинне зберігати усього два значення – 0 (червоний) або 1 (чорний) – 1 бітом можна представити два значення – 0 і 1.

Існує можливість визначати неіменовані бітові поля, які виконують роль доповнень у структурі.

Наприклад, в наступному визначенні структури

```
struct Example
{
    unsigned int a : 13;
    unsigned int   : 19;
    unsigned int b : 4;
};
```

використовується неіменоване бітове поле шириною 19 бітів, яке відіграє роль доповнення – в ньому нічого не зберігається. Поле **b** (в системах с 4–байтним машинним словом) буде зберігатися в окремому машинному слові.

Наступна програма створює масив **deck**, який містить 52 структури **struct** BitCard [1]. Функція fillDeck() заповнює масив структур **deck** інформацією про 52 карти, а функція deal() виводить вміст цього масиву. Зверніть увагу, що доступ до бітових полів структури здійснюється так само, як доступ до будь–яких інших полів структур. Поле **color** було додано з метою відображення кольору масті карти в системах, які мають кольоровий дисплей. Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#define CARDS 52
// Визначення структури BitCard з бітовими полями
struct BitCard
{
    unsigned int face  : 4;   // 4 біти (0-15)
    unsigned int suit  : 2;   // 2 біти (0-3)
    unsigned int color : 1;   // 1 біт (0-1)
};
typedef struct BitCard Card;
void fillDeck(Card *const wDeck);
void deal(const Card *const wDeck);
```

```
int main(void)
{
    Card deck[CARDS];     // створюємо масив структур

    SetConsoleOutputCP(1251);
    fillDeck(deck);
    deal(deck);
    return 0;
}

// ініціалізація масиву wDeck
void fillDeck(card *const wDesk)
{
    size_t i; // лічильник

    for(i = 0; i < CARDS; i++)
    {
        wDesk[i].face  = i % (CARDS / 4);
        wDesk[i].suit  = i / (CARDS / 4);
        wDesk[i].color = i / (CARDS / 4);
    }
}
// виводить на екран карти в дві колонки
// карти з індексами 0-25 - в колонку 1
// карти з індексами 26-51 - в колонку 2

void deal(const Card *const wDeck)
{
    size_t k1;    // індекси 0-25
    size_t k2;    // індекси 26-51

    // цикл по елементах масиву wDeck
    for(k1 = 0, k2 = k1 + 26; k1 < CARDS / 2; k1++, k2++)
    {
        printf("Карта:%3d Масть:%2d Колір:%2d ",
        wDeck[k1].face, wDeck[k1].suit, wDeck[k1].color);
        printf("Карта:%3d Масть:%2d Колір:%2d\n",
        wDeck[k2].face, wDeck[k2].suit, wDeck[k2].color);
    }
}
```

Результат виконання програми наведено на рис. 2.11.

Неіменоване бітове поле нульової ширини використовується для вирівнювання наступних бітових полів по межах одиниць зберігання. Наприклад, у визначенні структури

```
struct Example
{
    unsigned int a : 13;
    unsigned int   : 0;
    unsigned int b : 4;
};
```

використовується неіменоване бітове поле нульової ширини, щоб пропустити решту незайнятих бітів (скільки б їх не було) в одиниці зберігання, де зберігається поле **a**, і вирівнює поле **b** по межі наступної одиниці зберігання.

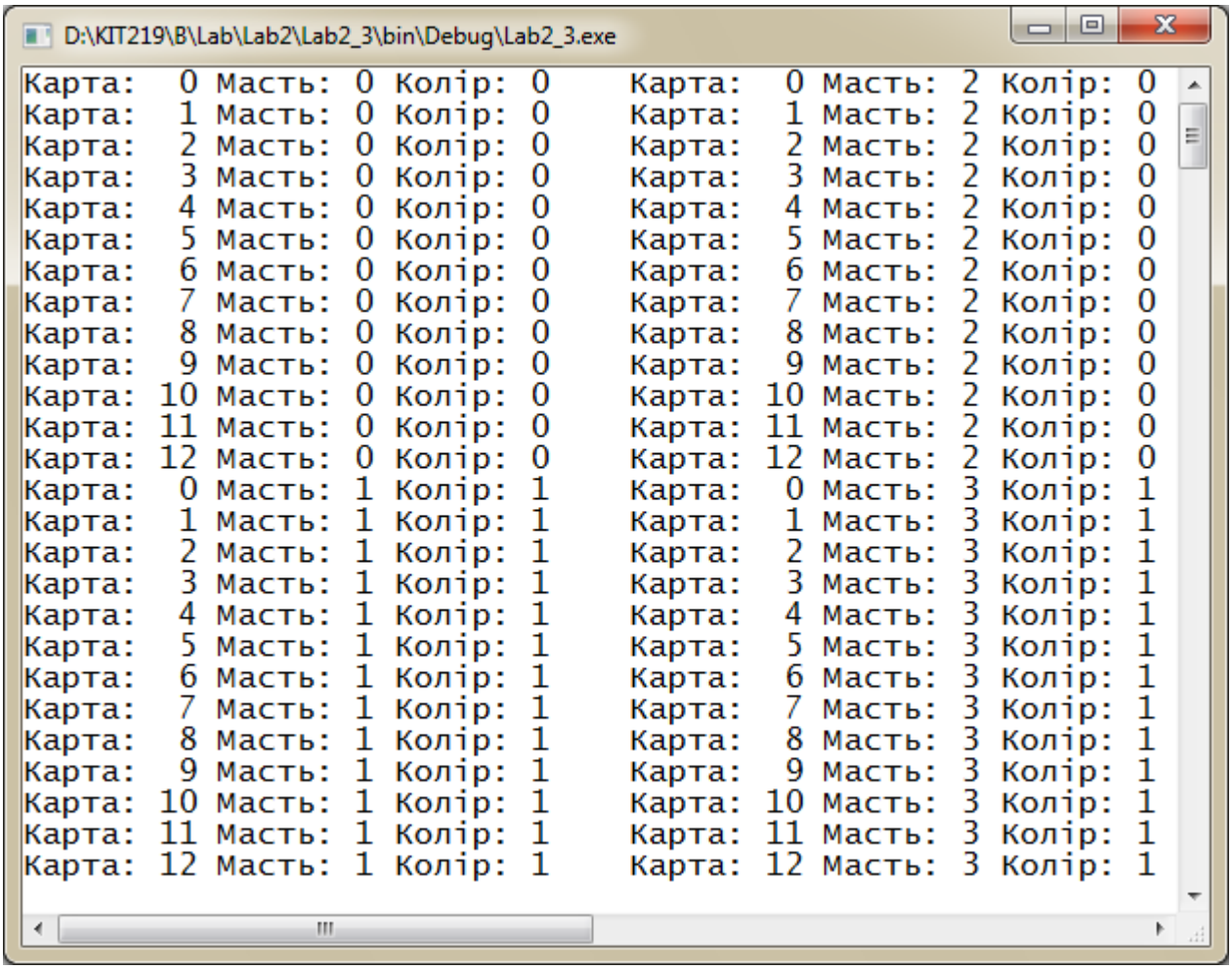


Рисунок 2.11 – Результат виконання програми для ініціалізації гральних карт за допомогою бітових полів у структурі

Операції з бітовими полями залежать від реалізації. Намагання звернення до окремих бітів бітового поля як до елементів масиву є синтаксичною помилкою, оскільки бітові поля не є «масивами бітів». Намагання отримати адресу бітового поля є також синтаксичною помилкою (оператор `'&'` не може застосовуватися до бітових полів, оскільки вони не є адресованими елементами).

Незважаючи на те, що бітові поля дозволяють економити пам’ять, для виконання операцій з ними компілятор генерує більш повільний машинний код.

Це обумовлено необхідністю виконання додаткових операцій для доступу до адресованих фрагментів одиниць зберігання. Це один з багатьох прикладів економії пам'яті за рахунок зменшення швидкості виконання, які часто можна зустріти в програмуванні.

Хід виконання роботи:

1. Опрацюйте матеріал лекції «Об'єднання, перелічувані типи та бітові поля», а також теоретичний матеріал даної лабораторної роботи.

2. Напишіть *C*–програму для демонстрації можливостей роботи з об'єднанням, яке повинно містити різні типи даних (**`short`**, **`int`**, **`long`**, **`float`**, **`double`**, **`char`**), а також масив символьного типу розміром 8 байтів. Необхідно створити шаблон об'єднання, присвоїти значення різним його полям та отримати на екрані значення полів різного типу. Крім того, необхідно визначити розмір об'єднання за допомогою **`sizeof`** та пояснити отримані результати. Під час роботи з кожним типом даних, визначте вміст окремих байтів об'єднання. Значення, які необхідно присвоїти полям різного типу для кожного варіанта, наведені в табл. 2.1. Приклад програми, який треба взяти за основу для виконання даного завдання, наведено в додатку 2.

3. Напишіть *C*–програму, яка б продемонструвала можливості роботи з переліченими типами. Для цього сформуйте набір цілочислових констант і виведіть значення кожної з них на екран. Присвойте двом довільним константам будь–які додатні значення та продемонструйте на екрані як це вплине на інші константи. Присвойте одній константі будь–яке додатне значення, а другій – будь–яке від'ємне значення, та продемонструйте на екрані як це вплине на інші константи. Приклад виконання даного завдання наведено в додатку 3.

4. Придумайте власний варіант трьох комбінацій карт для прикладу №4, запустіть *C*–програму на виконання та виведіть результат роботи на екран.

5. Напишіть *C*–програму для роботи з бітовими полями. Присвойте полям різні значення та виведіть їх на екран. Приклад програми для виконання цього завдання наведено в додатку 4.

Таблиця 2.1 – Дані для виконання завдання 2

№ варіанта	Типи даних					
	short	**int**	**long**	**float**	**double**	**char**
1	3530	594221574	298808479	1119.16	9049.97	А
2	3136	747507770	1155421309	1674.10	5984.97	д
3	4368	1562668627	2072391829	570.60	3286.16	ц
4	3868	873990190	1224809639	1452.71	4526.32	ш
5	30232	868748076	2141090330	1881.83	10403.28	Б
6	27406	1585861809	289495671	1305.55	7646.22	Ф
7	20698	51927206	1071737324	766.68	9288.81	и
8	8366	1496913912	503925195	745.93	9330.33	д
9	4751	1846880023	1599632136	1403.79	7028.10	я
10	5590	1624261752	2030074046	956.59	9502.28	Х
11	6576	1862303507	306864810	714.44	3518.87	ж
12	9631	1361997030	1158748078	1858.74	6035.97	в
13	16887	1733980528	776165456	354.02	10876.49	к
14	12386	896799798	1315129657	1312.58	5027.12	Р
15	29788	803432258	973671862	1648.93	8915.37	Ч
16	11191	1476492703	442528314	713.34	3671.31	у
17	7531	534223774	294408479	2319.13	4067.32	С
18	4186	347307970	2355421309	1674.12	5984.93	К
19	1320	1561118627	1072391829	672.65	2286.16	Ц
20	9822	673290194	1114809639	2463.72	2557.31	Й

Примітка: До символьного масиву необхідно занести своє прізвище, а у випадку, коли воно за довжиною перевищує 7 символів, – занести лише 7 букв.

Звіт про виконання лабораторної роботи повинен містити:

1. Титульний аркуш.
2. Мету роботи.
3. Результати виконання роботи:
   – умови завдань і тексти чотирьох *C*–програм;
   – результати виконання *C*–програм.
4. Висновки по роботі.

Контрольні запитання:

1. Що таке об’єднання?
2. З чого складається шаблон об’єднання?
3. Як визначити змінну типу «об’єднання»?
4. Як ініціалізувати об’єднання?
5. Які існують способи доступу до членів об’єднання?
6. Як визначити масив об’єднань?
7. Чи може бути об’єднання членом структури?
8. Що таке перелічення та з яким ключовим словом воно пов’язане?
9. Якого типу повинні бути константи в переліченні?
10. Як визначити бітові поля та для чого вони використовуються?

Лабораторна робота №3
ДОСЛІДЖЕННЯ ПОБІТОВИХ ОПЕРАЦІЙ МОВИ *C*

Мета роботи: засвоїти основні побітові операції над двійковими числами; навчитися використовувати операцію "доповнення до одиниці" або "побітове заперечення" `(~)`, побітову операцію "І" `(&)`, побітову операцію "АБО" `(|)`, побітову операцію "виключне АБО" `(^)`; відпрацювати навички роботи з побітовими операціями "зсув вліво" `(<<)` і "зсув вправо" `(>>)`; навчитися застосовувати маски для виконання операцій над бітами.

Теоретична частина

Будь–які дані, що записані в пам’ять комп’ютера, являють собою послідовність бітів. Кожен біт може приймати одне зі значень: нуль (0) або одиниця (1). Тобто уся інформація будь–якого виду в пам’яті комп’ютера це лише послідовність нулів та одиниць [4].

Мінімальний об’єм пам’яті, що використовується для адресації, складає один байт, який містить 8 бітів. Як правило, будь–яке число цілого типу буде

займати чотири байти в пам’яті, тобто 32 біти. При виконанні різних операцій цю множину бітів можна розглядати або як ціле число, або як послідовність бітів, що стає можливим при використанні бітових операцій.

При відображенні байту або слова (2 або 4 байти залежно від системи) молодші біти розташовуються праворуч (рис. 3.1).

Номери бітів															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Значення бітів															
0	1	0	0	1	0	1	1	0	1	1	1	0	1	1	0

Рисунок 3.1 – Приклад відображення 16–розрядного слова

Бітові операції виконуються незалежно над кожним бітом даних. Якщо операція двомісна, то вона виконується над відповідними бітами двох операндів. У мові *C* існують чотири бітові операції, які наведені в табл. 3.1.

Таблиця 3.1 – Побітові операції мови *C*

Назва операції	Вид операції	Операція *C*
Побітове заперечення	одномісна	**~**
Побітова операція "І"	двомісна	**&**
Побітова операція "АБО"	двомісна	**¦**
Побітова операція "виключне АБО"	двомісна	**^**

Результат застосування цих операцій для можливих комбінацій бітів двох операндів **op1** і **op2** наведено в табл. 3.2.

Таблиця 3.2 – Результати застосування побітових логічних операцій

op1	**op2**	**~op1**	**op1 & op2**	**op1 ^ op2**	**op1 ¦ op2**
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	0	1

Розглянемо декілька прикладів з операцій з операндами, які містять по чотири біти (табл. 3.3).

Таблиця 3.3 – Приклади виконання операцій над бінарними даними

Операнди	Операція	Результат
1010	~	0101
1010 1100	&	1000
1010 1100	^	0110
1010 1100	¦	1110

Двомісні операції в програмуванні часто використовуються для виконання операції маскування. У цьому випадку значення результату сприймається як значення першого операнда, але модифіковане значенням маски (другого операнда) з урахуванням виконаної операції. Такі операції дозволяють:

– виділяти один або кілька бітів у байті (8 бітів) або слові (16 або 32 біти);

– встановлювати окремий біт або кілька бітів у значення 1;

– встановлювати окремий біт або кілька бітів у значення 0.

Мова програмування *С* не має команд аналізу та операцій з окремим бітом. Немає таких команд і у процесора, але завдяки використанню «маски» можна легко встановлювати окремі біти операнда в необхідні значення. Для встановлення біту в значення 0 використовується команда "І", а для встановлення в 1 – "АБО". При цьому треба пам’ятати, що інші біти в підсумковій комбінації повинні залишитися в операнді без змін.

Якщо уважно проаналізувати табл. 3.2, стає зрозуміло, що у другому операнді (масці) в тих бітах, які не повинні змінюватися, необхідно встановити 1 при операції встановлення бітів в значення 0, і в значення 0, якщо необхідно встановити в окремих бітах 1.

Іншими словами, маска в будь–якому випадку складається з двох частин:

перша – біти, які необхідно встановити в конкретне значення, друга – біти, які повинні залишитися незмінними.

При встановленні бітів в 0 використовуємо операцію "І". В масці біти, які необхідно встановити, заповнюємо значенням 0, а незмінні – в 1.

При встановленні бітів в 1 використовуємо операцію "АБО". В масці біти, які необхідно встановити, заповнюємо значенням 1, а незмінні – в 0.

Приклад №1. Встановити 1–й і 2–й біти у значення 0.

Використовуємо операцію "І" та маску `'1001'`.

Тоді, наприклад, комбінація `'0101'` зміниться наступним чином:

```
&0101   число
 1001   маска
 0001   результат
```

Приклад №2. Встановити 1–й і 2–й біти у значення 1.

Використовуємо операцію "АБО" і маску `'0110'`.

Тоді, наприклад, комбінація `'0101'` зміниться наступним чином:

```
¦0101   число
 0110   маска
 0111   результат
```

Іноді під час програмування доводиться використовувати прапорці – біти, які вказують, наприклад, що відбулася певна подія. Часто досить двійкового прапорця. Ці операції зазвичай і використовуються для встановлення або скидання прапорців.

Виділення окремих бітів зазвичай здійснюється в два етапи:

– встановлення «непотрібних» бітів у значення 0 (тобто робиться скидання цих бітів);

– зсув бітів вправо таким чином, щоб самий правий біт, з тих що потрібно виділити, став нульовим (тобто зайняв нульову позицію).

Приклад №3. Виділити 1–й і 2–й біти та дізнатися значення в цих бітах.

Використовуємо операцію "І" та маску `'0110'`. Тобто з такою маскою. будуть встановлені в 0 біти 0 і 3, а біти 1 і 2 залишаться незмінними.

Тоді, наприклад, комбінація **'0101'** після застосування маски і зсуву вправо на одну позицію дасть такий результат:

0101 & 0110 = 0100 >> 1 = 0010_2 = 2_{10}.

Примітка: При зсуві вправо ліві біти заповнюються нулями.

Так можна дізнатися яке значення (число) записано в кількох бітах, що йдуть поспіль, але стоять у середині слова. Якщо потрібне значення записане в кількох молодших розрядах (включаючи 0–й розряд), то встановлюють у значення 0 всі старші розряди (окрім потрібних), і зсув не застосовують (або зсув здійснюється на нуль позицій).

Хід виконання роботи:

1. Опрацюйте матеріал лекції «Маніпулювання бітами», а також теоретичний матеріал даної лабораторної роботи.

2. Напишіть *C*–програму, в якій необхідно:

1) підрахувати кількість одиниць в кожному з операндів **a** і **b** типу **unsigned short**, які відповідають вашому варіанта (додаток 5);

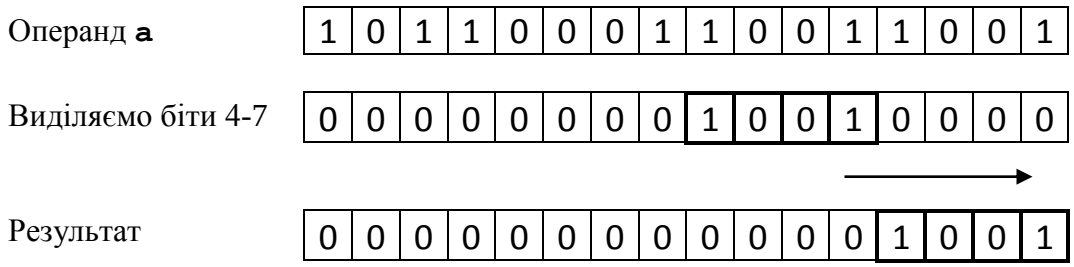
2) виконати над кожним з операндів одномісну операцію "побітове заперечення";

3) здійснити над операндами **a** і **b** двомісні побітові операції "І", "АБО" та "виключне АБО";

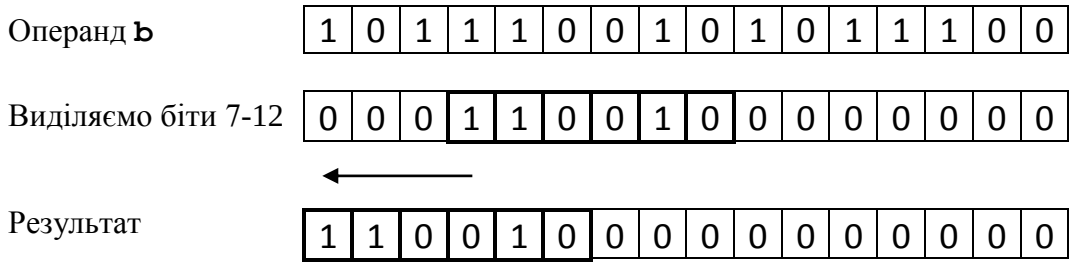
4) встановити біти, які зазначені у вашому варіанті завдання для першого операнда **a**, в значення 0;

5) встановити біти, які зазначені у вашому варіанті завдання для другого операнда **b**, в значення 1;

6) в першому операнді виділити біти, що зазначені у варіанті завдання, та виконати зсув результату вправо таким чином, щоб перший правий виділений біт став у нульову позицію (тобто став нульовим розрядом);



7) у другому операнді виділити біти, що зазначені у варіанті завдання, та виконати зсув результату вліво таким чином, щоб перший лівий виділений біт став у п’ятнадцяту позицію (тобто став лівим крайнім розрядом).



3. Всі результати побітових операцій та операндів **a** і **b** необхідно виводити у двійковому вигляді, а кількість одиниць в операндах – у десятковому.

Приклад зовнішнього вигляду результатів роботи програми наведено в додатку 6 (див. рис. Д 6.1 і рис. Д 6.2). В завданнях 4–7 необхідно виділити червоним кольором біти, для яких застосовуються бітові операції.

Звіт про виконання лабораторної роботи повинен містити:

1. Титульний аркуш.
2. Мету роботи.
3. Результати виконання роботи:
   – текст *C*–програми;
   – результати виконання *C*–програми (на двох екранах).
4. Висновки по роботі.

Контрольні запитання:

1. Які два види побітових операцій існує в мові *C*?
2. Що собою являє операція побітового "І"?
3. Що собою являє операція побітового "АБО"?
4. Що собою являє операція побітового "виключного АБО"?
5. Що таке «маска» та для чого вона застосовується?
6. Як встановити визначений біт в одиницю?
7. Як встановити визначений біт в нуль?
8. Як здійснити перемикання бітів?
9. Як перевірити значення окремого біта?
10. В чому полягає особливість операції побітового зсуву вліво?
11. В чому полягає особливість операції побітового зсуву вправо?

Лабораторна робота №4
ДОСЛІДЖЕННЯ ФУНКЦІЙ ДИНАМІЧНОГО ВИДІЛЕННЯ ПАМ'ЯТІ

Мета роботи: засвоїти основні поняття, що застосовуються при динамічному виділенні пам'яті: купа, розмір купи, витік пам'яті та інші; навчитися використовувати різні способи виділення пам'яті для одновимірних, двовимірних та багатовимірних масивів; ознайомитися з особливостями роботи з функціями динамічного виділення пам'яті `malloc()`, `calloc()` і `realloc()`, а також з функцією звільнення пам'яті `free()`.

Теоретична частина

4.1. Купа (пам'ять)

Купа (англ. heap) в програмуванні – це назва структури даних, за допомогою якої реалізована пам'ять застосунків, що виділяється динамічно [6].

Розмір купи – розмір пам'яті, яка виділяється операційною системою (ОС) для зберігання купи.

4.1.1. Принцип роботи купи

Під час запуску будь–якого процесу ОС виділяє пам’ять для розміщення купи. В подальшому пам’ять під купу може виділятися динамічно.

Програма користувача, що застосовує функції динамічного виділення пам’яті, може отримувати вказівники на області пам’яті, які належать купі. Програми використовують купу для розміщення динамічно створюваних структур даних. Програма може звільнити пам’ять за допомогою відповідної функції.

Пам’ять купи можна розділити на зайняту (яка виділена за допомогою функцій виділення пам’яті) і вільну (яка ще не зайнята або вже звільнена за допомогою функції звільнення пам’яті).

Для зберігання даних про те, яка область купи є зайнятою, а яка – вільною, зазвичай використовується додаткова область пам’яті.

Перед початком роботи програми виконується ініціалізація купи, в ході якої пам’ять, що виділена під купу, позначається як вільна.

Функція виділення динамічної пам’яті виконує наступні дії:

– переглядає список зайнятих/вільних областей пам’яті, які розміщені в купі, в пошуках вільної області потрібного розміру;

– у випадку нестачі вільної пам’яті може зробити запит на додаткову пам’ять у ОС;

– додає знайдену область до списку зайнятих областей (або позначає область як зайняту);

– повертає вказівник на початок області пам’яті;

– якщо виділити пам’ять не вдалося, повідомляє про помилку.

Функція звільнення пам’яті виконує наступні дії:

– переглядає список зайнятих/вільних областей пам’яті, які розміщені в купі, в пошуках вказаної області;

– видаляє зі списку зайнятих областей пам’яті знайдену область;

– додає знайдену область до списку вільних областей (або позначає область як вільну).

Програма може бути впевнена в тому, що між викликами функцій звільнення пам’яті, позначена зайнятою область не буде виділена повторно. Після виклику функції звільнення пам’яті, виділена область буде звільнена і в подальшому може використовуватися повторно або може бути віддана ОС. Використання вказівника на звільнену область пам’яті буде призводити до перебоїв або непередбачуваної роботи програми.

4.1.2. Алгоритми та продуктивність

Купа являє собою неперервну область пам’яті, яка поділена на зайняті та вільні області (блоки) різного розміру.

Інформація про вільні та зайняті області купи може зберігатися в списках різних форматів. Від обраного формату списку напряму залежить продуктивність функцій виділення та звільнення пам’яті, оскільки більшу частину часу ці функції витрачають на пошук у списку потрібних областей.

Для збільшення розміру купи функція виділення пам’яті використовує системний виклик (викликає функцію ОС). При цьому відбувається перемикання контексту з простору користувача на простір ядра ОС та у зворотному напрямку. Пошук у списку зайнятих/вільних областей купи виконується швидше, ніж перемикання контекстів, тому вигідніше один раз використати системний виклик для виділення великої області пам’яті під купу, а в подальшому виділяти програмі області меншого розміру з наявної великої області з веденням списку зайнятих/вільних областей.

Кількість елементів, що входять до списку зайнятих/вільних областей купи, може бути зменшено шляхом злиття елементів, які містять інформацію про області, що йдуть одна за одною. Це дозволить зменшити час обходу списку.

Кожен елемент списку може зберігати адресу області пам’яті, її розмір, інформацію про наступну (для пошуку в прямому напрямку) та/або попередню (для пошуку в зворотному напрямку) області.

4.2. Функція `malloc()`

Для виділення пам'яті в купі в *C* використовується функція `malloc()` (memory allocation) з бібліотеки **`stdlib.h`**, прототип якої має такий вид:

```
void *malloc(size_t size);
```

Функція виділяє **`size`** байтів пам'яті та повертає вказівник на неї. Якщо пам'ять виділити не вдалося, то функція повертає `NULL`.

Оскільки `malloc()` повертає вказівник типу **`void`**, то його необхідно явно приводити до потрібного типу.

Приклад №1. Напишемо *C*–програму, в якій створимо вказівник на **`int`** та виділимо пам'ять розміром 16 байт за допомогою функції `malloc()`. Після роботи з пам'яттю, звільнимо її за допомогою функції `free()`. Врахуємо той факт, що з виділеною пам'яттю можна працювати як з масивом.

Текст програми буде мати такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>

int main(void)
{
    int *p = NULL;

    SetConsoleOutputCP(1251);

    p = (int *) malloc(16);

    printf("Значення адреси для вказівника на int:     %p\n", &p);
    printf("Значення, що зберігається за цією адресою: %p\n\n",p);

    for(int i = 0; i < 4; i++)
        printf("p[%d] = %10d; &p[%d] = %p\n", i, p[i], i, &p[i]);
    printf("\n\n");

    for(int i = 0; i < 4; i++)
    {
        p[i] = i;
        printf("p[%d] = %10d; &p[%d] = %p\n", i, p[i], i, &p[i]);
    }
    printf("\n\n");
    free(p);
    return 0;
}
```

Результат виконання двох викликів програми наведено на рис. 4.1 і рис. 4.2. Проаналізувавши отримані результати, можна зробити наступні висновки:

1) При виділенні пам’яті за допомогою функції `malloc()` необхідно в дужках вказувати кількість байтів, яка буде кратною розміру типу даних, що виділяється. Тому в нашому випадку правильніше було б виділяти пам’ять за допомогою оператора

```
p = (int *) malloc(4 * sizeof(int));
```

2) Пам’ять, що виділяється для змінних програми (в нашому випадку, під вказівник **p**), і пам’ять, яка виділяється функцією `malloc()`, знаходяться в різних діапазонах адрес. Причому адреса, що виділяється для вказівника **p** при першому (див. рис. 4.1) і другому (див. рис. 4.2) запусках програми, не змінюється і дорівнює 0022ff04, а адреса початку блоку пам’яті, що виділяється за допомогою функції `malloc()`, – змінюється. Це говорить про те, що пам’ять у другому випадку виділяється в купі (з більшими адресами 005717e0 і 002d17e0 відповідно).

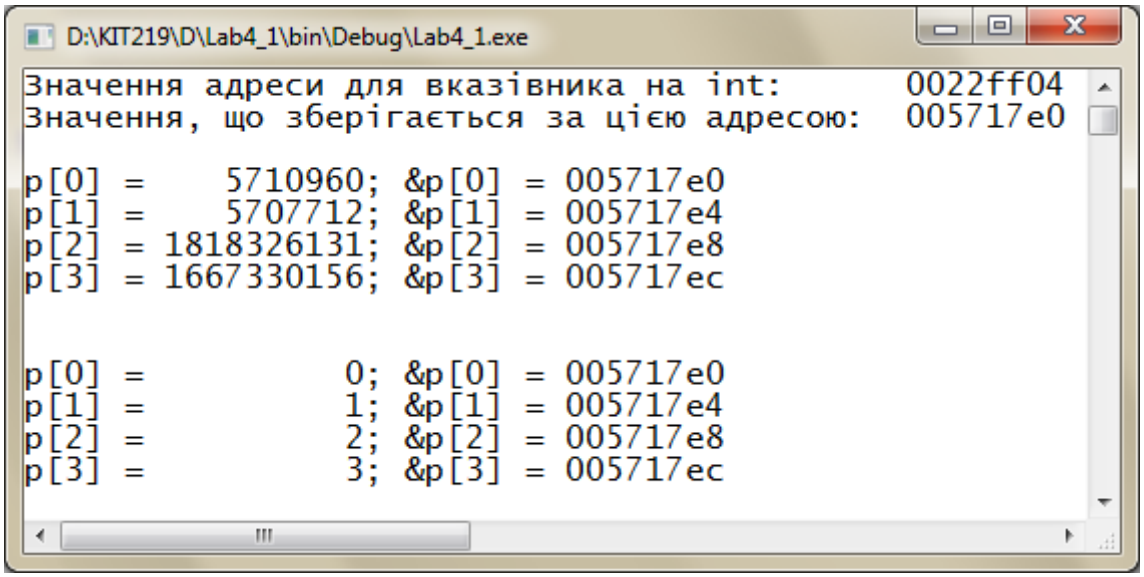


Рисунок 4.1 – Результат роботи програми під час першого запуску

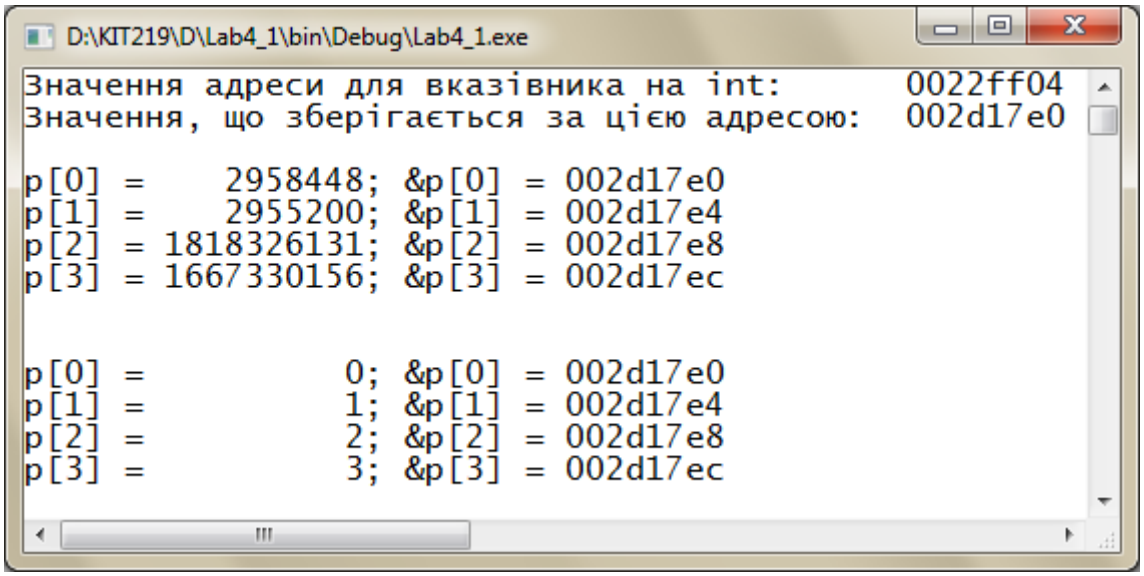


Рисунок 4.2 – Результат роботи програми під час другого запуску

3) Пам’ять, що виділяється функцією `malloc()`, не ініціалізується нулями, а містить випадкові величини. Кожне наступне ціле число розташовується за адресою, яка є кратною розміру типу (в нашому випадку 4).

Приклад №2. Напишемо *C*–програму, яка створює одновимірний масив необхідного розміру за допомогою функції `malloc()` і заповнює його квадратами чисел від 0 до **`size`**. Розмірність масиву будемо вводити з клавіатури.

Виведемо вміст масиву на екран і видалимо виділену пам’ять за допомогою функції `free()`.

Текст програми буде мати такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>

int main(void)
{
    const int maxNumber = 10;
    int *p = NULL;
    unsigned i, size;

    SetConsoleOutputCP(1251);

    do
    {
        printf("==============================\n");
        printf("Введіть число від 0 до %d: ", maxNumber);
```

```
        scanf("%d", &size);
        printf("==============================\n");
        if(size <= maxNumber) break;
    } while(1);

    p = (int *) malloc(size * sizeof(int));

    for(i = 0; i < size; i++)
    {
        p[i] = i * i;
        printf("Квадрат числа %d дорівнює %4d\n", i, p[i]);
    }
    printf("==============================\n\n");
    free(p);
    return 0;
}
```

Результат виконання програми наведено на рис. 4.3.

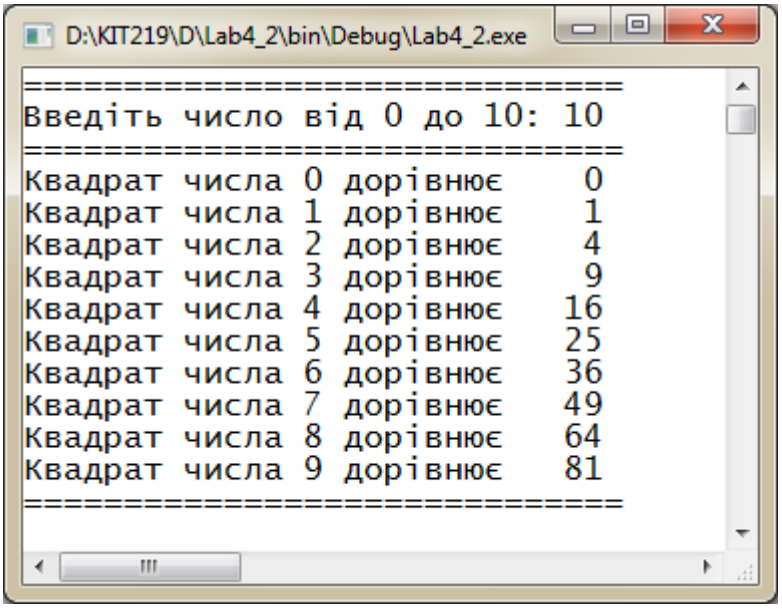


Рисунок 4.3 – Результат роботи програми для демонстрації динамічного виділення пам’яті за допомогою функції `malloc()`

Проаналізуємо код програми. Розглянемо оператор

```
p = (int *) malloc(size * sizeof(int));
```

де `(int *)` – приведення типу (пишемо такий самий тип, як і у вказівника); `size*sizeof(int)` – кількість байтів, яку необхідно виділити; `sizeof(int)` – розмір одного елемента масиву.

Після цього працюємо з вказівником так само, як і з масивом. Наприкінці програми не забуваємо видалити пам’ять, яка була виділена.

Тепер представимо графічно, що у нас буде відбуватися. Якщо, наприклад, ввести число 5, то будемо мати картину, яка наведена на рис. 4.4.

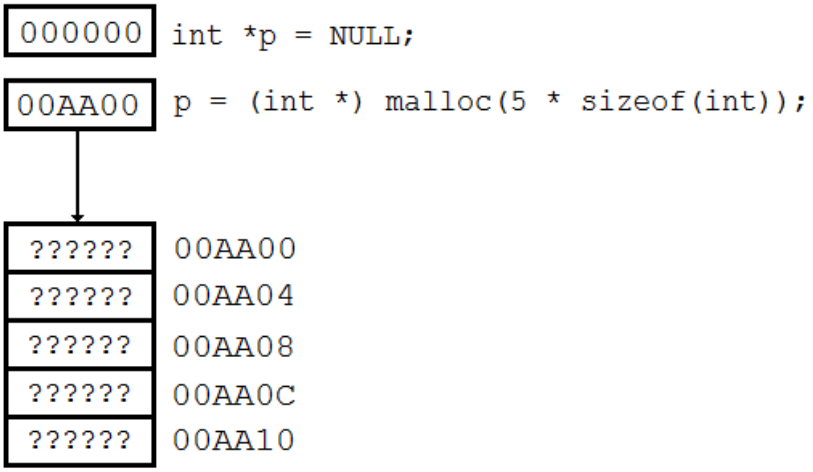


Рисунок 4.4 – Виділення пам’яті для п’яти цілих чисел

Функція `malloc()` виділила пам’ять в купі за певною адресою. Після цього вказівник **`p`** зберігає цю адресу і може нею користуватися для роботи. В принципі, він може користуватися й будь–якою іншою адресою.

Коли функція `malloc()` виділяє пам’ять, вона резервує місце в купі та повертає адресу цієї ділянки пам’яті. При цьому існує гарантія, що комп’ютер не віддасть цю пам’ять ще комусь. Коли ми викликаємо функцію `free()`, то ми звільняємо пам’ять, тобто говоримо комп’ютеру, що ця пам’ять може бути використана десь в іншому місці. Він може використовувати цю пам’ять, а може й не використовувати, але тепер у нас вже немає гарантії, що ця пам’ять є нашою. При цьому сама змінна не обнуляється, вона продовжує зберігати адресу, якою ми раніше користувалися.

Іноді вважають, що відбувається «створення» або «видалення» пам’яті. Насправді відбувається лише перерозподіл ресурсів.

4.3. Звільнення пам’яті за допомогою функції `free()`

Тепер розглянемо як відбувається звільнення пам’яті. Змінна–вказівник зберігає адресу області пам’яті, починаючи з якої вона може використовуватися. Однак вона не зберігає розміру цієї області. Звідки ж тоді функція `free()` знає скільки пам’яті необхідно звільнити?

Очевидно, що інформація про розмір виділеної ділянки пам’яті повинна десь зберігатися. Є декілька рішень цієї проблеми.

По–перше, можна створити карту, в якій буде зберігатися розмір виділеної ділянки. Кожного разу під час звільнення пам’яті комп’ютер буде звертатися до цих даних і отримувати потрібну інформацію.

По–друге, інформація про розмір може зберігається в купі, разом з самими даними. Таким чином, при виділенні пам’яті резервується більше місця і туди записується інформація про виділену ділянку. При звільненні пам’яті функція `free()` «підглядає» скільки пам’яті необхідно видалити.

Функція `free()` звільняє пам’ять, але при цьому не змінює значення вказівника, про що треба обов’язково пам’ятати.

4.4. Динамічне виділення пам’яті для двовимірних масивів

1–й спосіб. Для динамічного виділення пам’яті під двовимірний масив спочатку необхідно створити масив вказівників, після чого кожному з елементів цього масиву необхідно присвоїти адресу нового масиву.

Графічно це пояснюється на рис. 4.5.

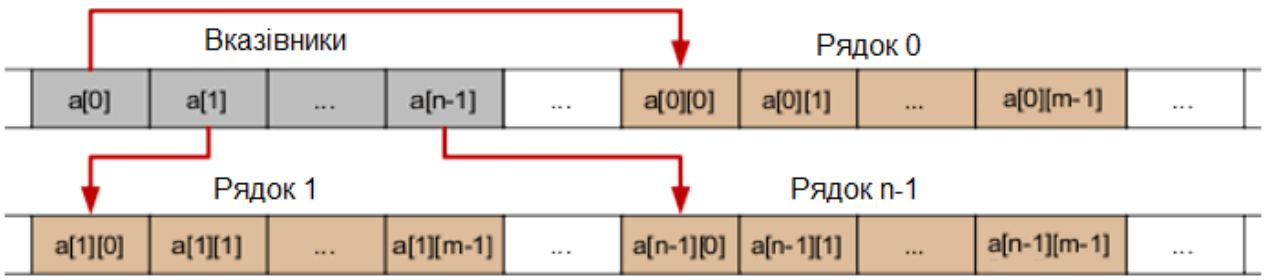


Рисунок 4.5 – Спосіб динамічного виділення пам’яті під двовимірний масив з використанням масиву вказівників

Для видалення двовимірного масиву необхідно повторити усі розглянуті операції у зворотному порядку: видалити спочатку масиви рядків, а потім і сам масив вказівників.

Приклад №3. Напишемо *C*–програму для створення двовимірного масиву дійсних чисел розмірністю 5×3 з використанням функції `malloc()`, занесемо до нього випадкові числа з діапазону від 2.50 до 7.00 та виведемо вміст цього масиву на екран з подальшим звільненням пам'яті за допомогою функції `free()`.

Текст програми буде мати такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <time.h>
#define ROW_NUM 5       // Кількість рядків
#define COL_NUM 3       // Кількість стовпців

int main(void)
{
    float **p = NULL;
    unsigned i, j;

    SetConsoleOutputCP(1251);

    //=============================================================
    // Декоративна частина
    //=============================================================
    if(COL_NUM >= 4)
        for(i = 0; i < 6 * COL_NUM; i++) printf("=");
    else for(i = 0; i < 24; i++) printf("=");
    printf("\n");
    printf("Масив розмірністю %d x %d\n", ROW_NUM, COL_NUM);
    if(COL_NUM >= 4)
        for(i = 0; i < 6 * COL_NUM; i++) printf("=");
    else for(i = 0; i < 24; i++) printf("=");
    printf("\n");
    //=============================================================
    // Виділення пам'яті під двовимірний масив типу float,
    // та заповнення його випадковими числами в заданому
    // діапазоні від 2.50 до 7.00
    //=============================================================
    srand(time(NULL));
    // Виділення пам'яті під масив вказівників на рядки
    p = (float **) malloc(ROW_NUM * sizeof(float *));

    for(i = 0; i < ROW_NUM; i++)
    {
        // Виділення пам'яті під рядки
        p[i] = (float *) malloc(COL_NUM * sizeof(float));
```

```
            for(j = 0; j < COL_NUM; j++)
            {
                p[i][j] = (float)rand()/RAND_MAX*(7.00-2.50)+2.50;
                printf("%6.2f", p[i][j]);
            }
            printf("\n");
        }
        //=========================================================
        // Декоративна частина
        //=========================================================
        if(COL_NUM >= 4)
            for(i = 0; i < 6 * COL_NUM; i++) printf("=");
        else for(i = 0; i < 24; i++) printf("=");
        printf("\n");
        //=========================================================
        // Звільнення пам'яті, яка відводилася
        // під двовимірний масив типу float
        //=========================================================
        // Звільнення пам'яті під рядки
        for(i = 0; i < ROW_NUM; i++)
            free(p[i]);
        // Звільнення пам'яті під масив вказівників на рядки
        free(p);
        return 0;
}
```

Результат роботи програми наведено на рис. 4.6.

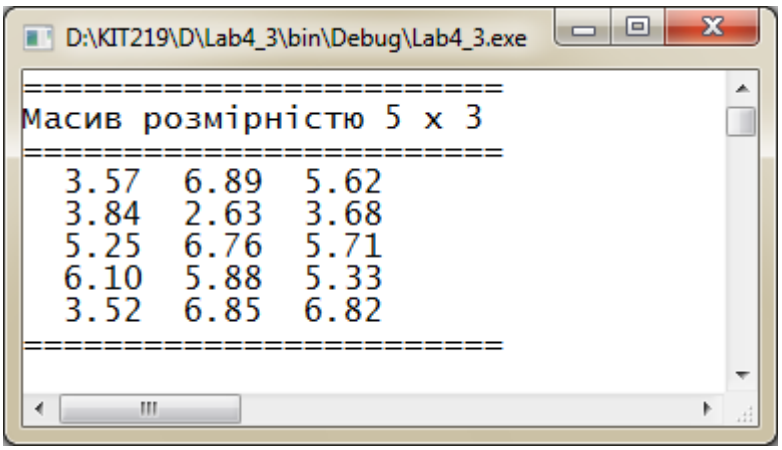


Рисунок 4.6 – Результат роботи програми, яка створює двовимірний масив чисел розмірністю 5×3 з використанням функції `malloc()`

Розглянутий спосіб дозволяє:

– створювати масиви «неправильної форми», тобто масив рядків, кожен з яких має свій розмір;

– працювати окремо з кожним рядком масиву: звільняти пам’ять або змінювати розмір рядка.

Приклад №4. Напишемо *С*–програму для створення двовимірного трикутного масиву цілих чисел за допомогою функції `malloc()`. Масив буде мати 10 рядків, і кожен наступний рядок буде містити на один елемент більше ніж попередній. Потім занесемо до нього добутки індексів масиву і виведемо результат на екран з подальшим звільненням пам’яті за допомогою `free()`.

Текст програми буде мати такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#define SIZE 10
int main(void)
{
    int **a;
    int i, j;

    SetConsoleOutputCP(1251);

    a = (int **) malloc(SIZE * sizeof(int *));
    for(i = 0; i < SIZE; i++)
        a[i] = (int *) malloc((i + 1) * sizeof(int));
    for(i = 0; i < SIZE; i++)
    {
        for(j = i; j >= 0; j--)
            a[i][j] = i * j;
    }
    printf("========================================\n");
    printf("Трикутний масив цілих чисел             \n");
    printf("========================================\n");
    for(i = 0; i < SIZE; i++)
    {
        for(j = i; j >= 0; j--)
            printf("%4d", a[i][j]);
        printf("\n");
    }
    printf("========================================\n\n");
    for(i = SIZE-1; i > 0; i--)
        free(a[i]);
    free(a);
    return 0;
}
```

Результат роботи програми наведено на рис. 4.7.

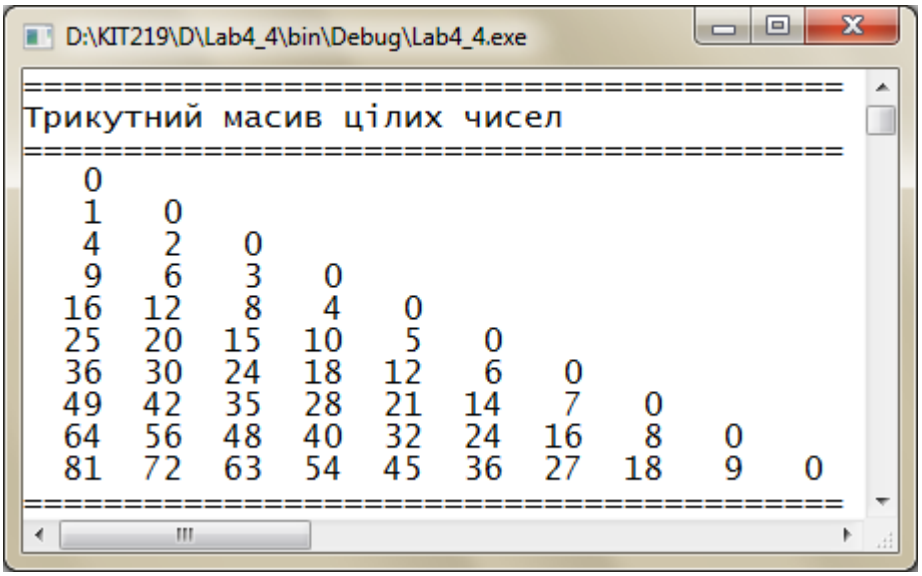


Рисунок 4.7 – Результат роботи програми, яка демонструє виділення пам’яті під трикутний масив

Приклад №5. Напишемо *C*–програму для створення двовимірного масиву **a** цілих чисел з рядками змінної довжини за допомогою функції `malloc()`. Інформацію про випадкову кількість елементів в кожному рядку будемо зберігати в масиві **m**. Масив **a** заповнимо випадковими цілими числами від **N2** до **N3**. Наприкінці програми звільнимо виділену пам'ять за допомогою функції `free()`.

Текст програми буде мати такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <time.h>
#define N1   1
#define N2 100
#define N3 200

int main(void)
{
    int **a;     // Масив вказівників на вказівники на цілі числа
    int i, j;
    int n;       // Кількість рядків масиву
    int k;       // Максимальна кількість елементів в рядку
    int *m;      // Масив кількостей елементів в рядках масиву a

    SetConsoleOutputCP(1251);

    printf("================================================\n");
    printf("Введіть кількість рядків масиву:                ");
    scanf("%d", &n);
```

```
    printf("Введіть максимальну кількість елементів в рядку: ");
    scanf("%d", &k);
    printf("================================================\n");
    // Виділення пам'яті під масив вказівників на рядки
    a = (int **) malloc(n * sizeof(int *));
    // масив кількостей елементів в рядках масиву a
    m = (int *) malloc(n * sizeof(int));
    srand(time(NULL));
    // Заповнення та вивід елементів масиву
    for(i = 0; i < n; i++)
    {
        m[i] = rand() % (k - N1 + 1) + N1;
        a[i] = (int *) malloc(m[i] * sizeof(int));
        for(j = 0; j < m[i]; j++)
        {
            a[i][j] = rand() % (N3 - N2 + 1) + N2;
            printf("%6d", a[i][j]);
        }
        printf("\n");
    }
    printf("================================================\n");
    // Звільнення пам'яті
    for(i = 0; i < n; i++)
        free(a[i]);
    free(a);
    free(m);
    return 0;
}
```

Результат роботи програми наведено на рис. 4.8.

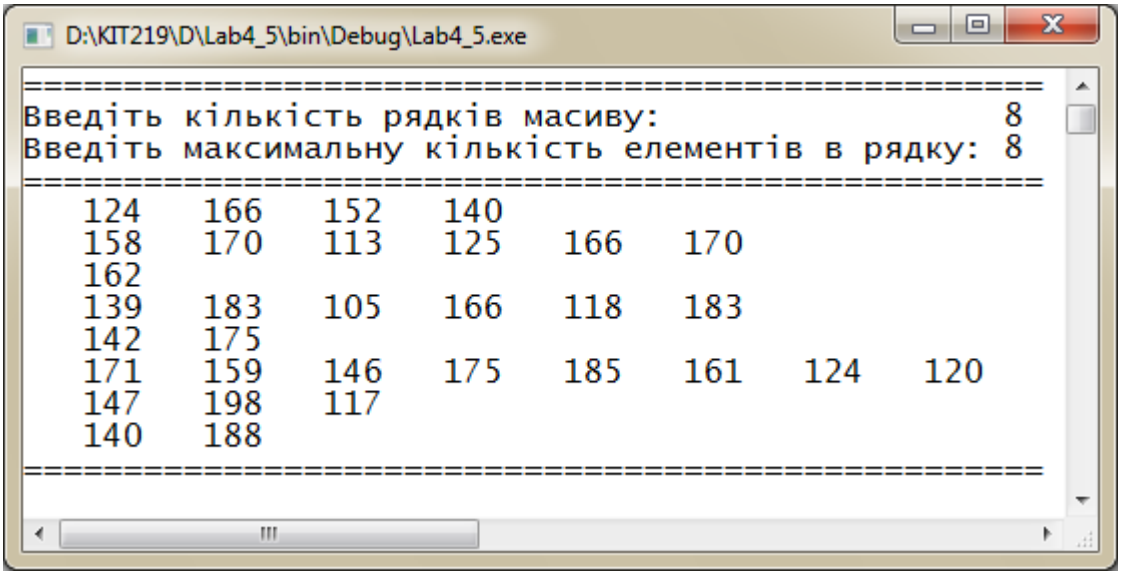


Рисунок 4.8 – Результат роботи програми, яка демонструє виділення пам’яті під масив цілих чисел з рядками змінної довжини

Щоб виділити пам’ять під тривимірний масив необхідно спочатку визначити вказівник на вказівник на вказівник, потім виділити пам’ять під масив вказівників на вказівник, проініціалізувати кожен з масивів і т. д.

2–й спосіб. Нехай треба розмістити в динамічний пам’яті матрицю **a**, яка містить **n** рядків і **m** стовпців. Двовимірна матриця (масив) буде розташовуватися в оперативній пам’яті в формі стрічки, яка складається з елементів рядків. При цьому індекс будь–якого елемента двовимірної матриці можна отримати за формулою

$$index = i \times m + j;$$

де i – номер поточного рядка; j – номер поточного стовпця.

Розглянемо матрицю 3×4 (рис. 4.9). Тоді індекс виділеного елемента можна визначити за формулою

$$index = 1 \times 4 + 2 = 6.$$

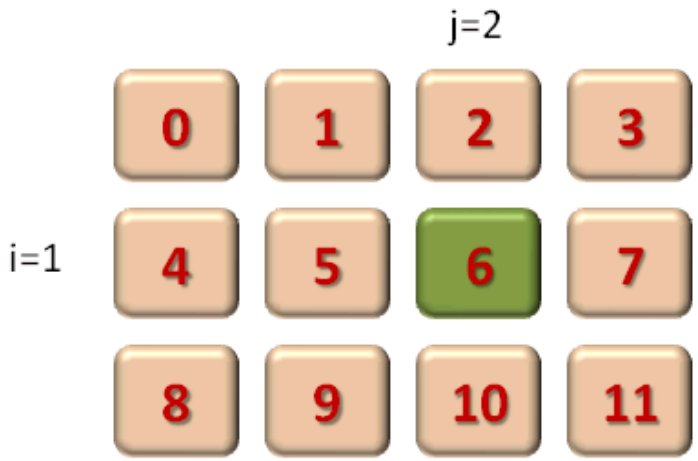


Рисунок 4.9 – Позначення виділеного елемента з номером 6 в матриці 3×4

Об’єм пам’яті, який необхідний для розміщення двовимірного масиву, визначається за формулою

$$n \times m \times (\text{розмір елемента})$$

Оскільки при такому оголошенні компілятору явно не вказується кількість елементів в рядку і стовпці двовимірного масиву, традиційне звернення до елемента шляхом зазначення індексу рядка та індексу стовпця **a**[i][j] є некоректним.

Правильне звернення до елемента з використанням вказівника **p** буде виглядати наступним чином:

```
*(p + i * m + j),
```

де **p** – вказівник на масив; **m** – кількість стовпців; **i** – індекс рядка; **j** – індекс стовпця.

Приклад №6. Напишемо *C*–програму для створення в динамічній пам'яті двовимірного масиву цілих чисел, який містить **n** рядків та **m** стовпців. Двовимірний масив буде розташовуватися в оперативній пам'яті в формі стрічки, яка складається з елементів рядків. Розглянемо можливість доступу до елементів масиву двома способами. Наприкінці програми звільнимо виділену пам'ять.

Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <malloc.h>
#include <stdlib.h>
#include <time.h>
#define N2 300
#define N3 400

int main(void)
{
    int *a;                 // вказівник на масив
    int i, j, n, m;

    SetConsoleOutputCP(1251);

    printf("==============================\n");
    printf("Введіть кількість рядків:   ");
    scanf("%d", &n);
    printf("Введіть кількість стовпців: ");
    scanf("%d", &m);
    printf("==============================\n");

    srand(time(NULL));
    // Виділення пам'яті під масив та вивід його на екран
    a = (int *) malloc(n * m * sizeof(int));
```

```
    // 1-й спосіб
    for(i = 0; i < n; i++)       // Цикл по рядках
    {
        for(j = 0; j < m; j++)   // Цикл по стовпцях
        {
            *(a + i * m + j) = rand() % (N3 - N2 + 1) + N2;
            printf("%5d", *(a + i * m + j));
        }
        printf("\n");
    }
    printf("==============================\n");

    // 2-й спосіб
    for(i = 0; i < n; i++)       // Цикл по рядках
    {
        for(j = 0; j < m; j++)   // Цикл по стовпцях
        {
            a[i * m + j] = rand() % (N3 - N2 + 1) + N2;
            printf("%5d", a[i * m + j]);
        }
        printf("\n");
    }
    printf("==============================\n");

    free(a);
    return 0;
}
```

Результат роботи програми наведено на рис. 4.10.

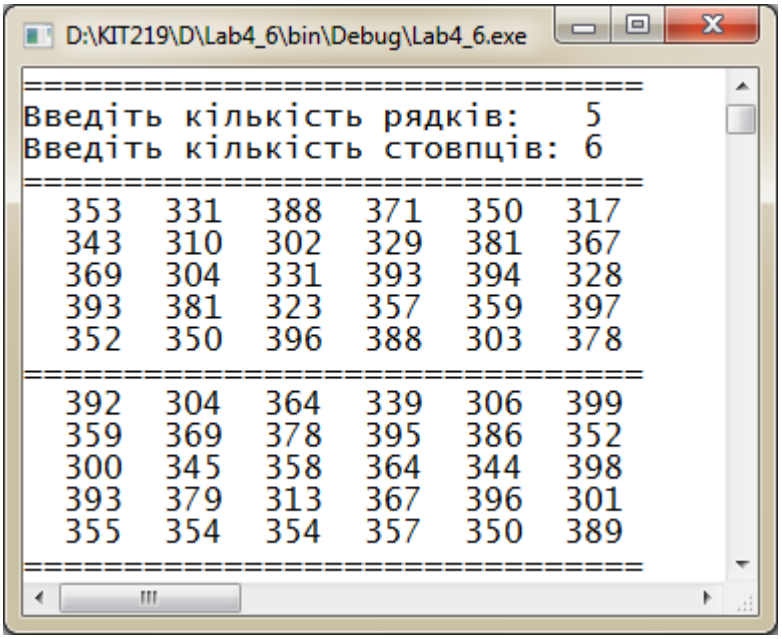


Рисунок 4.10 – Результат роботи програми, яка демонструє виділення пам’яті під масив цілих чисел у формі стрічки

4.5. Функція `calloc()`

Функція `calloc()`з бібліотеки **`stdlib.h`** виділяє **`n`** об'єктів розміром **`m`** і заповнює їх нулями. Зазвичай вона використовується для виділення пам'яті під масиви. Прототип цієї функції має такий вигляд:

```
void *calloc(size_t num, size_t size);
```

Наводити приклади для даної функції немає сенсу, оскільки достатньо в операторах виділення пам'яті під масив замість знаку множення записати кому. Тобто замість оператору

```
p = (int *) malloc(size * sizeof(int));
```

записати

```
p = (int *) calloc(size, sizeof(int));
```

і пам'ять буде виділена, з тією лише різницею, що усі елементи масиву будуть заповнені нулями.

Інтерес представляє лише випадок з символьним масивом, який не розглядався для функції `malloc()`.

Приклад №7. Напишемо *С*–програму для виділення пам'яті під символьний масив змінної довжини за допомогою функції `calloc()`, в якій присвоїмо елементам різні значення кодів символів для цифр від 0 до 9 і виведемо вміст масиву на екран. Наприкінці програми звільнимо виділену пам'ять.

Текст програми буде мати такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
    char **str; // Масив вказівників на символьні рядки
    int i, j;
    int n;      // Кількість рядків масиву
    int k;      // Максимальна кількість символів у рядку
    int *m;     // Масив кількостей символів в рядках масиву str

    SetConsoleOutputCP(1251);
    printf("=============================================\n");
```

```
    printf("Введіть кількість рядків масиву:                ");
    scanf("%d", &n);
    printf("Введіть максимальну кількість символів у рядку: ");
    scanf("%d", &k);

    printf("================================================\n");
    printf("Масив рядків цифр змінної довжини                \n");
    printf("================================================\n");
    // Виділення пам'яті під масив вказівників на символьні рядки
    str = (char **) calloc(n, sizeof(char *));

    // масив кількостей символів в рядках масиву str
    m = (int *) calloc(n, sizeof(char));
    srand(time(NULL));

    // Заповнення та вивід елементів масиву
    for(i = 0; i < n; i++)
    {
        m[i] = rand() % k + 1;
        str[i] = (char *) calloc(m[i] + 1, sizeof(char));

        for(j = 0; j < m[i]; j++)
        {
            str[i][j] = rand() % (57 - 48 + 1) + 48;
            printf("%c", str[i][j]);
        }
        str[i][j] = '\x0';
        printf("\n");
    }
    printf("================================================\n\n");

    // Звільнення пам'яті
    for(i = 0; i < n; i++)
        free(str[i]);
    free(str);
    free(m);
    return 0;
}
```

Результати роботи програми для двох запусків наведені на рис. 4.11 і рис. 4.12.

4.6. Функція `realloc()`

Розглянемо ще одну важливу функцію `realloc()` (reallocation). Вона дозволяє змінити розмір раніше виділеної пам’яті та отримує в якості аргументів старий вказівник і новий розмір пам’яті в байтах:

```
void *realloc(void *ptr, size_t size);
```

Функція `realloc()` може використовувати як раніше виділену ділянку пам’яті, так і нову. При цьому не важливо, меншим або більшим буде новий розмір. Менеджер пам’яті сам вирішує де виділяти пам’ять.

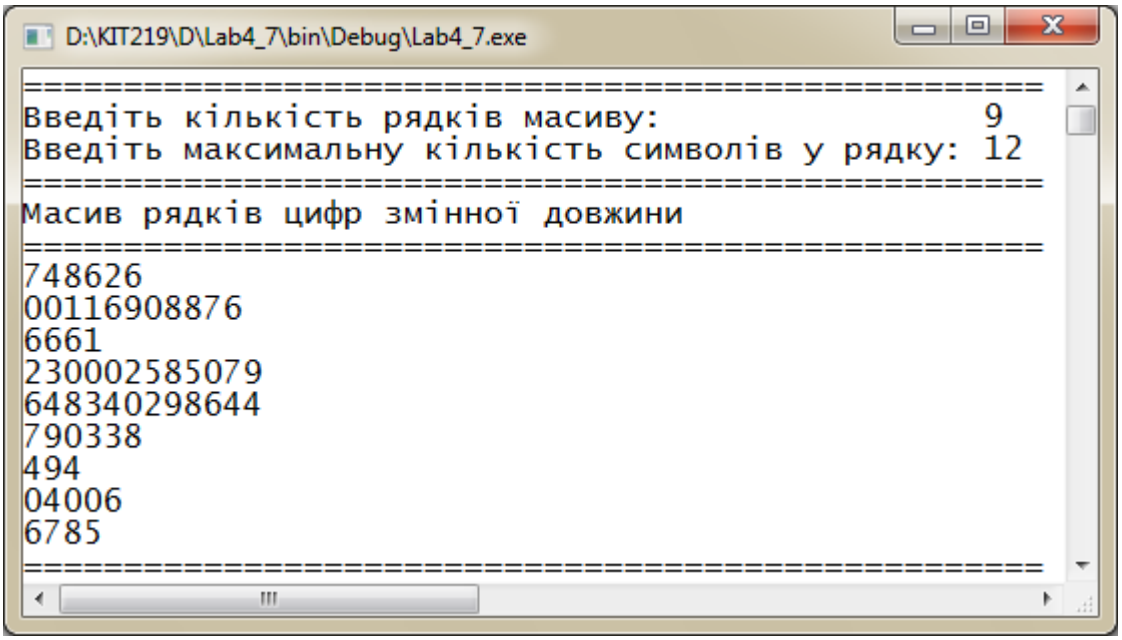


Рисунок 4.11 – Результат першого запуску програми, яка демонструє виділення пам’яті під символьний масив рядків цифр змінної довжини

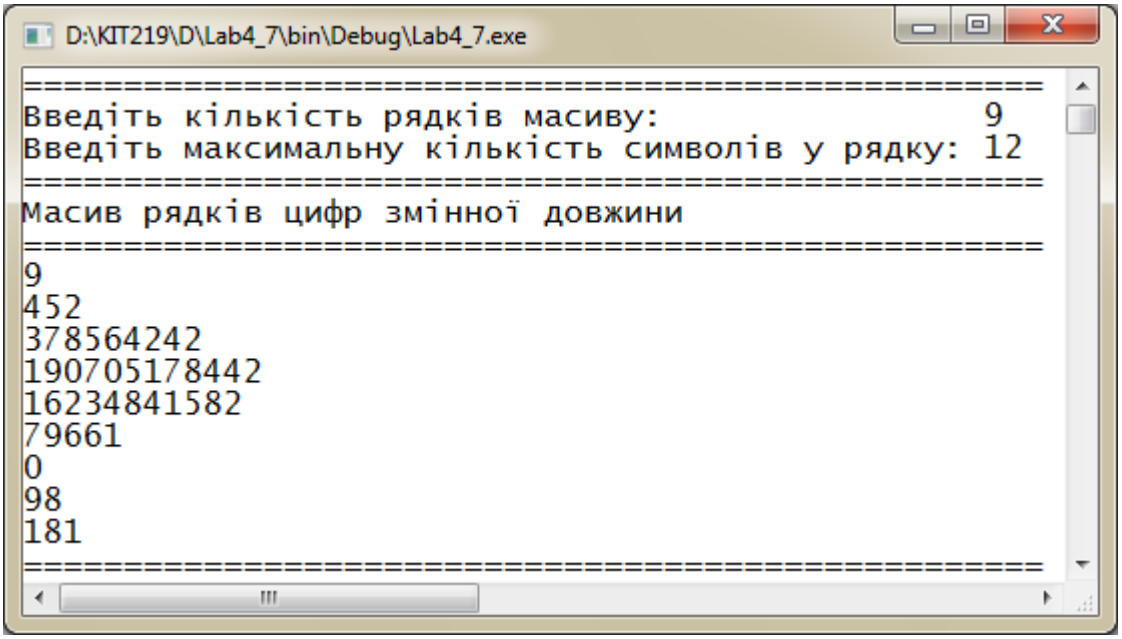


Рисунок 4.12 – Результат другого запуску програми, яка демонструє виділення пам’яті під символьний масив рядків цифр змінної довжини

Приклад №8. Напишемо *C*–програму, яка за допомогою функції `malloc()` виділяє пам'ять для 2 слів. Якщо ми введемо більше слів, розмір пам'яті буде збільшено за допомогою функції `realloc()`. Коли буде введено слово «Кінець», програма припинить роботу і виведе на екран усі введені раніше слова.

Текст програми буде мати такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <string.h>
#define TERM_WORD "Кінець"
#define SIZE_INCREMENT 2
int main(void)
{
    // Масив вказівників на слова
    char **words;
    // Рядок, який використовується для зчитування введеного
    // користувачем слова
    char buffer[128];
    // Лічильник слів
    unsigned wordCounter = 0;
    // Довжина введеного слова
    unsigned length;
    // Розмір масиву слів для зменшення витрат
    // на виділення пам'яті. Кожного разу будемо
    // збільшувати масив слів не на одне значення,
    // а на SIZE_INCREMENT слів
    unsigned size = SIZE_INCREMENT;
    int i;
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    // Виділяємо пам'ять під масив з size вказівників
    printf("====================================\n");
    printf("Введіть декілька слів.\n");
    printf("Ознакою завершення є слово 'Кінець'.\n");
    printf("====================================\n");
    words = (char **) malloc(size * sizeof(char *));
    do
    {
        printf("Лічильник: %2d   Слово: ", wordCounter);
        scanf("%127s", buffer);
        // Функція strcmp повертає 0, якщо два рядки збігаються
        if(strcmp(TERM_WORD, buffer) == 0) break;
        // Визначаємо довжину слова
        length = strlen(buffer);
        // У випадку, коли введено слів більше, ніж передбачено,
        // збільшуємо масив слів
        if(wordCounter >= size-1)
        {
```

```
            size += SIZE_INCREMENT;
            printf("Додаємо пам'ять ще для %d слів\n",
                   SIZE_INCREMENT);
            words = (char **) realloc(words, size*sizeof(char *));
        }
        // Виділяємо пам’ять безпосередньо під слово
        // на 1 байт більше, оскільки необхідно зберігати
        // символ завершення рядка
        words[wordCounter] = (char *) malloc(length + 1);
        // Копіюємо слово з буфера за адресою, яка
        // зберігається в масиві вказівників на слова
        strcpy(words[wordCounter], buffer);
        wordCounter++;
    } while(1);
    printf("\n=====================================\n");
    printf("Введені слова:\n");
    printf("=====================================\n");
    for(i = 0; i < wordCounter; i++)
        printf("%s\n", words[i]);
    printf("=====================================\n\n");
    for(i = 0; i < wordCounter; i++) free(words[i]);
    free(words);
    return 0;
}
```

Результат роботи програми наведено на рис. 4.13.

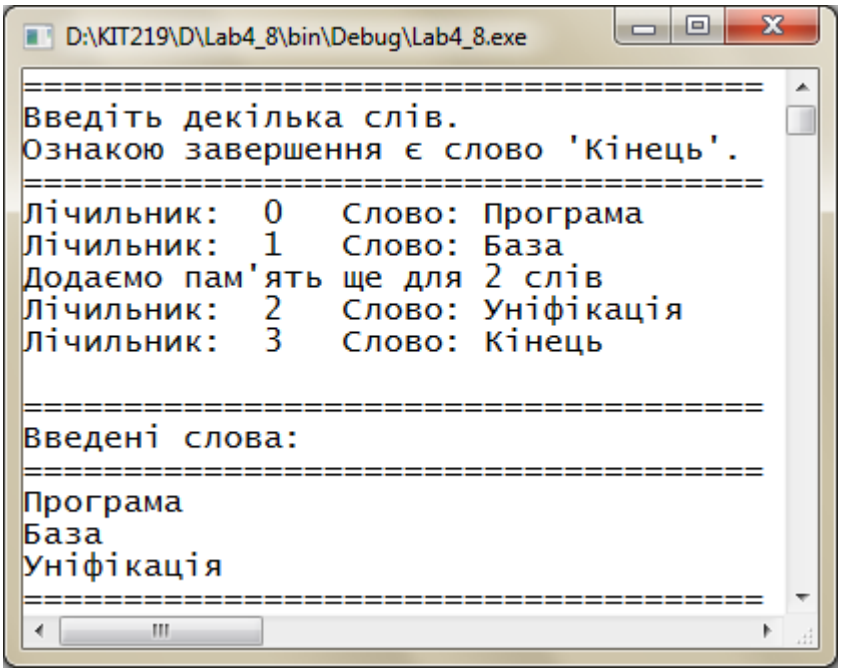


Рисунок 4.13 – Результат роботи програми, яка демонструє виділення блоку пам’яті під символьний масив та збільшує його у разі необхідності

Слід звернути увагу на те, що при виділенні пам'яті ми пишемо **sizeof**(**char** *), тому що розмір вказівника на **char** не дорівнює одному байту, на відміну від розміру змінної типу **char**.

4.7. Помилки при виділенні пам'яті

Існують ситуації, при яких пам'ять не може бути виділена. В цьому випадку функції `malloc()` і `calloc()` повертають `NULL`. Тому перед виділенням пам'яті необхідно обнулити вказівник, а після виділення перевірити, чи не дорівнює він нулю (`NULL`). Також поводить себе і функція `realloc()`.

Коли ми використовуємо функцію `free()`, перевіряти на `NULL` немає необхідності, оскільки відповідно до документації `free(NULL)` не робить жодних дій. Відповідно до останнього прикладу, оператори, що містять функції `malloc()` і `realloc()` повинні виглядати наступним чином:

```
if(!(words = (char **) malloc(size * sizeof(char *))))
{
   printf("Неможливо виділити пам'ять функцією malloc()\n");
   exit(1);
}
if(!(words = (char **) realloc(words, size * sizeof(char *))))
{
   printf("Неможливо перевизначити пам'ять функцією realloc()\n");
   exit(2);
}
if(!(words[wordCounter] = (char *) malloc(length + 1)))
{
   printf("Неможливо виділити пам'ять функцією malloc()\n");
   exit(3);
}
```

Слід додати, що помилки виділення пам'яті можуть відбуватися час від часу, або просто може здійснюватися вихід з програми і видаватися помилка аварійного завершення. Рішення залежить від ситуації. Наприклад, якщо не вистачає пам'яті, то можна почекати деякий час і після цього знову намагатися виділити пам'ять. Можна також для тимчасового зберігання даних відкрити файл і перемістити туди частину об'єктів, або виконати очищення даних, скоротивши пам'ять, яка використовується, і видалити непотрібні об'єкти.

До помилки може призвести зміна вказівника, який зберігає адресу ділянки пам’яті, яка виділяється. Як вже згадувалося раніше, в області пам’яті, що виділяється, зберігаються дані про об’єкт, а саме про його розмір. При видаленні `free()` отримує цю інформацію. Однак якщо ми змінили вказівник, то видалення призведе до помилки.

Таким чином, якщо вказівник зберігає адресу, то його не треба змінювати. Для роботи краще створити додаткову змінну–вказівник, з якою треба працювати далі не змінюючи початкову адресу.

До помилки може призвести також використання вже звільненої області. Ця помилка спричиняє іншу – так звані висячі вказівники (dangling pointers або wild pointers). Ви видаляєте об’єкт, але при цьому забуваєте змінити значення вказівника на `NULL`. У підсумку, він зберігає адресу області пам’яті, якою вже неможна скористатися. При цьому перевірити, чи є валідною ця область чи ні, у нас немає можливості. Тому слід писати:

```
free(p);
p = NULL;
```

Таким чином, якщо ви звільняєте пам’ять і використовуєте вказівник у подальшому, то обов’язково обнуліть його.

Хід виконання роботи:

1. Опрацюйте матеріал лекції «Специфікатори класів зберігання. Виділення пам’яті», матеріал практичного заняття «Динамічне виділення пам’яті та багатофайлова компіляція», а також теоретичний матеріал даної лабораторної роботи.

2. Напишіть *C*–програму, яка виділяє за допомогою функції `malloc()` пам’ять для зберігання **`n`** байтів. Вхідні дані для програми наведені в табл. 4.1. Визначте кількість елементів масиву заданого в табл. 4.1 типу, яку можна зберігати у виділеній пам’яті. Переконайтеся, що після виділення пам’яті, масив буде містити довільні числа. Звільніть за допомогою функції `free()` пам’ять, яка була виділена в програмі.

Таблиця 4.1 – Вхідні дані для виконання завдань 2 – 5

№ вар.	Номери завдань												
	2		3				4					5	
	n	тип	**n**	тип	**a**	**b**	**n**	**m**	тип	**a**	**b**	**n**	операція
1	8	**short**	9	**int**	49	87	3	4	**double**	3.76	8.34	4	**^**
2	16	**int**	7	**double**	2.65	4.66	5	5	**int**	10	27	7	**&**
3	32	**double**	4	**float**	1.12	2.23	6	3	**short**	23	45	9	**¦**
4	8	**long**	8	**short**	111	160	2	9	**double**	0.56	3.09	6	**^**
5	24	**short**	6	**float**	3.45	5.86	7	7	**int**	7	11	4	**&**
6	16	**float**	5	**long**	200	340	6	4	**float**	2.07	6.59	5	**¦**
7	32	**double**	7	**int**	25	125	5	4	**double**	1.29	4.55	10	**^**
8	64	**double**	5	**double**	3.33	7.65	6	6	**long**	122	345	12	**&**
9	8	**int**	6	**float**	5.51	5.87	3	3	**int**	23	64	8	**¦**
10	16	**float**	9	**long**	650	987	5	8	**double**	2.30	5.67	11	**^**
11	24	**int**	4	**int**	70	90	7	6	**float**	1.22	6.59	7	**&**
12	32	**double**	3	**double**	1.18	2.68	8	4	**short**	43	98	8	**¦**
13	8	**int**	8	**float**	2.68	7.79	4	6	**int**	14	57	6	**^**
14	16	**float**	5	**double**	1.56	5.62	7	5	**long**	128	845	9	**&**
15	64	**double**	7	**short**	361	612	6	2	**double**	3.48	5.49	12	**¦**
16	24	**long**	8	**long**	164	629	8	7	**float**	4.53	9.21	7	**^**
17	8	**short**	9	**int**	55	72	3	4	**double**	2.51	7.13	4	**¦**
18	16	**int**	7	**double**	1.47	6.72	5	5	**int**	13	52	7	**&**
19	32	**double**	4	**float**	4.65	8.11	6	3	**short**	43	85	9	**¦**
20	8	**long**	8	**short**	241	369	2	9	**double**	1.56	5.11	6	**^**

3. Напишіть *C*–програму, яка виділяє за допомогою функції `malloc()` пам'ять для зберігання **n** елементів масиву заданого в табл. 4.1 типу, ініціалізуйте їх випадковими числами від **a** до **b** та виведіть вміст масиву на екран. Наприкінці програми звільніть виділену пам'ять за допомогою функції `free()`.

4. Напишіть *C*–програму, яка виділяє за допомогою функції `malloc()` пам'ять для зберігання елементів двовимірного масиву заданого в табл. 4.1 типу розмірності **m**×**n**, ініціалізуєте їх випадковими числами від **a** до **b**. Виведіть вміст масиву на екран з подальшим звільненням за допомогою функції `free()` пам'яті, яка була для нього виділена.

5. Напишіть *C*–програму, яка виділяє за допомогою функції `malloc()` пам'ять для зберігання елементів трикутного масиву заданого в табл. 4.1 типу, що має **n** рядків і кожен наступний рядок містить на один елемент більше ніж

попередній. Потім занесіть до нього результати визначеної бітової операції з індексами масиву і виведіть результат на екран з подальшим звільненням за допомогою функції `free()` пам'яті, яка була виділена.

6. Напишіть *C*–програму, яка виділяє за допомогою функції `malloc()` пам'ять для зберігання елементів двовимірного масиву **`a`** цілих чисел з **`n`** рядками змінної довжини. Додатковий масив **`m`**, в якому будуть зберігатися кількість елементів в кожному рядку, необхідно заповнити випадковими цілими числами від **`N1`** до **`k`**. Масив **`a`** необхідно заповнити випадковими цілими числами від **`N2`** до **`N3`**. Вхідні дані для програми наведені в табл. 4.2.

Таблиця 4.2 – Вхідні дані для виконання завдань 6 – 8

№ вар.	Номери завдань													
	6					7				8		9		
	n	**N1**	**k**	**N2**	**N3**	**n**	**m**	**N2**	**N3**	**n**	**k**	**n**	**m**	слово
1	4	1	5	11	32	4	4	76	83	4	5	2	6	один
2	8	2	6	14	25	5	5	10	27	7	6	3	7	два
3	5	3	7	25	78	3	3	23	45	9	5	4	8	три
4	7	1	8	52	93	9	9	56	91	6	7	2	6	чотири
5	8	2	5	17	28	7	7	17	31	4	5	3	7	шість
6	9	3	6	32	95	4	4	20	65	5	6	4	8	сім
7	4	1	7	54	77	5	5	19	45	6	10	2	6	вісім
8	5	2	8	42	93	6	6	12	52	4	12	3	7	десять
9	6	3	5	21	47	3	3	23	64	8	6	4	8	одинадцять
10	7	1	6	16	38	8	8	30	67	3	11	2	6	дванадцять
11	8	2	7	43	96	6	6	22	59	7	6	3	7	тринадцять
12	9	3	8	45	55	4	4	33	78	8	5	4	8	шістнадцять
13	5	1	5	87	99	6	6	14	73	6	4	2	6	сімнадцять
14	6	2	6	32	48	5	5	28	84	4	9	3	7	вісімнадцять
15	5	3	7	52	75	7	7	48	59	4	12	4	8	двадцять
16	7	1	8	12	46	9	9	45	92	7	5	2	6	тридцять
17	4	1	5	17	44	4	4	13	53	4	5	2	6	чотири
18	8	2	6	12	38	5	5	17	64	7	6	3	7	шість
19	5	3	7	35	84	3	3	32	65	9	5	4	8	сім
20	7	1	8	22	63	9	9	32	55	6	7	2	6	вісім

7. Напишіть *C*–програму, яка виділяє за допомогою функції `malloc()` пам'ять для зберігання елементів двовимірного масиву випадкових цілих чисел,

який містить **n** рядків і **m** стовпців. Двовимірний масив буде розташовуватися в оперативній пам'яті в формі стрічки, яка складається з елементів рядків. Заповніть його випадковими цілими числами від **N2** до **N3** двома способами та виведіть на екран. Вхідні дані для програми наведені в табл. 4.2.

8. Напишіть *C*–програму, яка виділяє за допомогою функції `calloc()` пам'ять для зберігання елементів двовимірного символьного масиву **str** змінної довжини з **n** рядками. Додатковий масив **m**, в якому будуть зберігатися кількості цифр в кожному рядку, необхідно заповнити випадковими цілими числами від 1 до **k**. Масив **str** необхідно заповнити випадковим чином кодами символів цифр від 0 до 9 і вивести його на екран. Вхідні дані для програми наведені в табл. 4.2.

9. Напишіть *C*–програму, яка повинна ввести **m** слів. Перші чотири введені слова повинні містити прізвище, ім'я, по батькові та номер групи. Інші слова повинні бути ключовими словами мови *C*. Після введення слова, що зазначене в табл. 4.2 для вашого варіанта, програма повинна припинити роботу і вивести на екран усі введені раніше слова.

Алгоритм програми повинен враховувати, що спочатку за допомогою функції `malloc()` треба виділити пам'ять для **n** слів. Коли користувач буде вводити слово, для якого не вистачає пам'яті, необхідно збільшити її розмір для можливості вводу ще **n** слів за допомогою функції `realloc()`.

Звіт про виконання лабораторної роботи повинен містити:

1. Титульний аркуш.
2. Мету роботи.
3. Результати виконання роботи:
   – тексти восьми *C*–програм;
   – результати виконання *C*–програм.
4. Висновки по роботі.

Контрольні запитання:

1. Які функції в мові *C* застосовуються для динамічного виділення пам’яті?

2. Яка функція застосовується для звільнення пам’яті? Як вона виглядає та скільки має параметрів?

3. Яке призначення має функція `malloc()` і в чому її особливість?

4. Чи можна за допомогою функції `malloc()` створювати масиви «неправильної форми»?

5. Яке призначення має функція `calloc()` і в чому її особливість?

6. Для чого потрібна функція `realloc()`? Коли вона застосовується?

7. Які способи виділення динамічної пам’яті для двовимірного масиву ви знаєте? Чим вони відрізняються?

8. Коли відбувається «витік пам’яті»?

9. В чому полягає принцип виділення динамічної пам’яті для багатовимірних масивів?

Лабораторна робота №5
ДОСЛІДЖЕННЯ ПРИНЦИПІВ СОРТУВАННЯ ЗВ’ЯЗНИХ СПИСКІВ

Мета роботи: розглянути найбільш популярні алгоритми сортування абстрактних типів даних на прикладі зв’язних списків; навчитися використовувати різні методи сортування (бульбашкове сортування, швидке сортування та сортування злиттям) і порівняти їх ефективність.

Теоретична частина

Сортування – це процес переставлення об’єктів скінченої множини в певному порядку, яке призначене для полегшення подальшого пошуку елементів у вже відсортованій множині [8].

Для сортування можна використовувати велику кількість алгоритмів, то-

му часто потрібно виконати порівняльний аналіз декількох алгоритмів, щоб визначити оптимальний вибір для конкретного завдання.

Методи сортування зазвичай поділяються на дві категорії: сортування масивів (або списків) і сортування файлів. Вони також мають назву внутрішнього та зовнішнього сортування, оскільки масиви (списки) розташовуються у внутрішній пам'яті, а файли – у зовнішній.

Найважливішою вимогою до алгоритмів сортування масивів і зв'язних списків є економне витрачання пам'яті, тому використання додаткових масивів для сортування не вважається оптимальним рішенням.

Основним критерієм для класифікації алгоритмів сортування є їх швидкодія. При сортуванні виконуються два типи операцій – порівняння ключів і переставлення елементів (у випадку зі зв'язними списками – перепризначення вказівників). Якщо масив складається з **n** елементів, то алгоритм вважається добрим, якщо число переставлень дорівнює добутку числа елементів масиву **n** на логарифм цього числа `log(n)`. Наприклад, стандартне бульбашкове сортування вимагає число переставлень, яке дорівнює квадрату від **n.**

Методи сортування поділяються на три основні класи:

– сортування включеннями;

– сортування вибором;

– сортування обміном.

Принцип, що лежить в основі першого класу, полягає в тому, що масив (список) поділяється на дві частини – підсумкову та вхідну. Береться елемент з вхідної половини і вставляється у підсумкову в порядку сортування. Число переставлень елементів в загальному випадку дорівнює квадрату від **n.**

У разі сортування вибором береться мінімальний елемент масиву і ставиться на початок масиву, і т. д. Число переставлень елементів в загальному випадку менше, ніж у першому випадку, і буде дорівнювати добутку числа елементів масиву на логарифм від цього числа. Швидке сортування відноситься до другого класу.

5.1. Бульбашкове сортування

Алгоритм бульбашкового сортування (Bubble sort) потребує порядку **n×n** переставлень, тому є не найкращим вибором. У функції `llist_bubble_sort()`, що наведена нижче, відбувається вкладена ітерація списком, за допомогою якої реалізується алгоритм бульбашкового сортування на мові *C*.

Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <time.h>
#define MAX 10

typedef struct lnode
{
    int data;
    struct lnode *next;
} LNode;

LNode *head, *visit;

void llist_add(LNode **q, int num);
void llist_bubble_sort(void);
void llist_print(void);
int main(void)
{
    LNode *newnode = NULL;
    int i = 0;

    SetConsoleOutputCP(1251);

    srand(time(NULL));
    for(i = 0; i < MAX; i++)
        llist_add(&newnode, rand() % 100);

    head = newnode;
    printf("=======================================\n");
    printf("До бульбашкового сортування:\n");
    printf("=======================================\n");
    llist_print();
    printf("=======================================\n");
    printf("Після бульбашкового сортування:\n");
    printf("=======================================\n");
    llist_bubble_sort();
    llist_print();
    printf("=======================================\n");
    return 0;
}
```

```
void llist_add(LNode **q, int num)
{
    LNode *tmp;

    tmp = *q;
    if(*q == NULL)
    {
        *q = malloc(sizeof(LNode));
        tmp = *q;
    }
    else
    {
        while(tmp->next != NULL)
            tmp = tmp->next;
        tmp->next = malloc(sizeof(LNode));
        tmp = tmp->next;
    }
    tmp->data = num;
    tmp->next = NULL;
}

void llist_print(void)
{
    visit = head;

    while(visit != NULL)
    {
        printf("%4d", visit->data);
        visit = visit->next;
    }
    printf("\n");
}

void llist_bubble_sort(void)
{
    LNode *a   = NULL;
    LNode *b   = NULL;
    LNode *c   = NULL;
    LNode *e   = NULL;
    LNode *tmp = NULL;

    while(e != head->next)
    {
        c = a = head;
        b = a->next;
        while(a != e)
        {
            if(a->data > b->data)
            {
                if(a == head)
                {
                    tmp = b->next;
                    b->next = a;
```

```
                    a->next = tmp;
                    head = b;
                    c = b;
                }
                else
                {
                    tmp = b->next;
                    b->next = a;
                    a->next = tmp;
                    c->next = b;
                    c = b;
                }
            }
            else
            {
                c = a;
                a = a->next;
            }
            b = a->next;
            if(b == e)
                e = a;
        }
    }
}
```

Програма містить три основні функції. Функція `llist_add()` призначена для створення нового елемента списку, функція `llist_bubble_sort()` – здійснює сам алгоритм бульбашкового сортування, а функція `llist_print()` – виводить на екран список до та після сортування. Результат виконання програми бульбашкового сортування наведено на рис. 5.1.

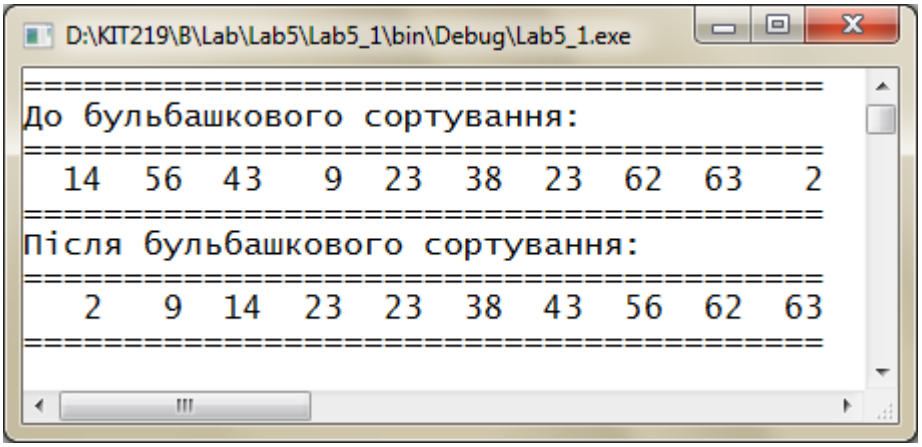


Рисунок 5.1 – Результат бульбашкового сортування для зв’язного списку

5.2. Швидке сортування

Розглянемо алгоритм швидкого сортування (Quick sort) двобічно зв'язного списку на основі рекурсивної функції `QuickSortList()`.

Дана функція працює наступним чином:

– виконується перегляд усіх елементів списку;

– в ході перегляду визначається максимальне значення, яке записується в кінець списку;

– під час наступних переглядів попередні максимальні значення не враховуються.

Текст функції `QuickSortList()` має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <time.h>
#define MAXDLIST 10
// елемент двобічно зв'язного списку
typedef struct intList
{
    int nInt;
    struct intList *pPrev;
    struct intList *pNext;
} IntList;

// вказівники на перший та останній елементи списку
IntList *pFirstItem = NULL;
IntList *pLastItem  = NULL;

// додавання елементу до списку
void AddListItem(int nInt)
{
    IntList *pItem = (IntList *) malloc(sizeof(IntList));
    pItem->nInt = nInt;
    if(pFirstItem == NULL)
    {
        pFirstItem = pLastItem = pItem;
        pItem->pNext = pItem->pPrev = NULL;
    }
    else
    {
        pLastItem->pNext = pItem;
        pItem->pPrev = pLastItem;
        pLastItem = pItem;
        pItem->pNext = NULL;
    }
}
```

```
// видалення усіх елементів списку
void RemoveAllItems()
{
    IntList *pDelItem, *pItem = pFirstItem;

    while(pItem != NULL)
    {
        pDelItem = pItem;
        pItem = pItem->pNext;
        free(pDelItem);
    }
    pFirstItem = pLastItem = NULL;
}

// вивід списку на екран
void PrintList()
{
    IntList *pItem = pFirstItem;

    while(pItem != NULL)
    {
        printf("%4d", pItem->nInt);
        pItem = pItem->pNext;
    }
    printf("\n");
}

// швидке сортування списку
void QuickSortList(IntList *pLeft, IntList *pRight)
{
    IntList *pStart;
    IntList *pCurrent;
    int nCopyInt;

    // сортування закінчено - вихід
    if(pLeft == pRight) return;
    // встановлення двох робочих вказівників
    // pStart і pCurrent
    pStart = pLeft;
    pCurrent = pStart->pNext;
    // перегляд списку зліва направо
    while(1)
    {
        // елемент з максимальним значенням поміщається
        // на початок списку
        if(pStart->nInt < pCurrent->nInt)
        {
            nCopyInt = pCurrent->nInt;
            pCurrent->nInt = pStart->nInt;
            pStart->nInt = nCopyInt;
        }
        if(pCurrent == pRight) break;
        pCurrent = pCurrent->pNext;
    }
```

```
    // переключення pFirst і pCurrent – максимум потрапляє
    // до правого кінця списку
    nCopyInt = pLeft->nInt;
    pLeft->nInt = pCurrent->nInt;
    pCurrent->nInt = nCopyInt;

    // зберігання pCurrent
    IntList *pOldCurrent = pCurrent;

    // рекурсія
    pCurrent = pCurrent->pPrev;
    if(pCurrent != NULL)
    {
        if((pLeft->pPrev != pCurrent) &&
           (pCurrent->pNext != pLeft))
            QuickSortList(pLeft, pCurrent);
    }

    pCurrent = pOldCurrent;
    pCurrent = pCurrent->pNext;
    if(pCurrent != NULL)
    {
        if((pCurrent->pPrev != pRight) &&
           (pRight->pNext != pCurrent))
            QuickSortList(pCurrent, pRight);
    }
}

int main(void)
{
    SetConsoleOutputCP(1251);

    srand(time(NULL));
    for(int i = 0; i < MAXDLIST; i++)
        AddListItem(rand() % 100);

    printf("=======================================\n");
    printf("Список до швидкого сортування:         \n");
    printf("=======================================\n");
    PrintList();
    QuickSortList(pFirstItem, pLastItem);
    printf("=======================================\n");
    printf("Список після швидкого сортування:      \n");
    printf("=======================================\n");
    PrintList();
    RemoveAllItems();
    printf("=======================================\n");
    return 0;
}
```

Програма містить чотири основні функції. Функція `AddListItem()` призначена для створення нового елемента списку, функція `RemoveAllItems()` –

видаляє усі елементи списку, функція `QuickSortList()` – здійснює сам алгоритм швидкого сортування, а функція `PrintList()` – виводить на екран список до та після сортування. Результат виконання програми швидкого сортування наведено на рис. 5.2.

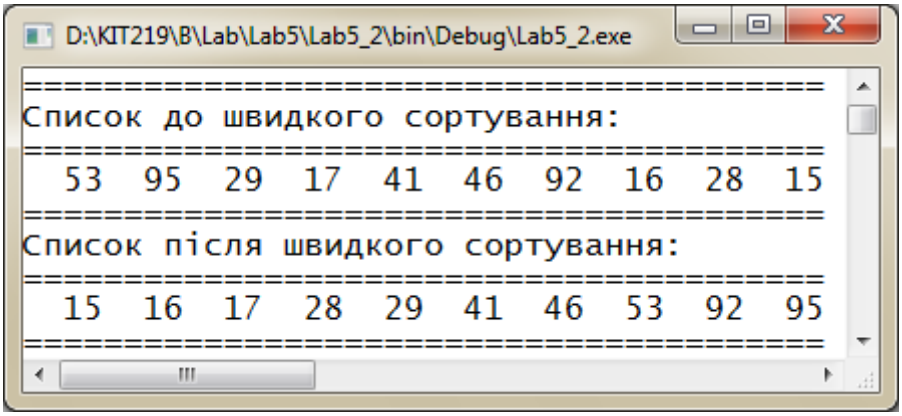


Рисунок 5.2 – Результат роботи програми для швидкого сортування

5.3. Сортування злиттям

При сортуванні злиттям (Merge sort) список поділяється на дві частини. Кожна частина сортується окремо, а потім обидві частини зливаються [7].

Практично це можна пояснити на наступному алгоритмі:

1) колода карт ділиться на дві частини, і кожна частина сортується окремо, щоб зверху виявилася наймолодша карта;

2) з двох верхніх карт вибирають найменшу і перекладають її в результуючу колоду;

3) процес повторюється, поки не закінчаться карти в обох половинах;

4) якщо в одній половині карти закінчаться раніше, тоді усі карти, що залишилися в іншій половині, просто перекладаються в результуючу колоду.

Вважається, що цей алгоритм був придуманий Джоном фон Нейманом у 1945 році. Недоліком сортуванням злиттям вважається те, що воно потребує $O(n)$ пам’яті. Функція, яка здійснює сортування злиттям рекурсивно викликає сама себе, надаючи частини списку (рис. 5.3). Якщо до функції прийшов список

довжиною менш ніж два елементи, то рекурсія припиняється. Далі відбувається зворотне збирання списку. Спочатку з двох списків, кожен з яких зберігає один елемент, створюється відсортований список, далі з таких списків збирається новий відсортований список, поки усі елементи не будуть включені.

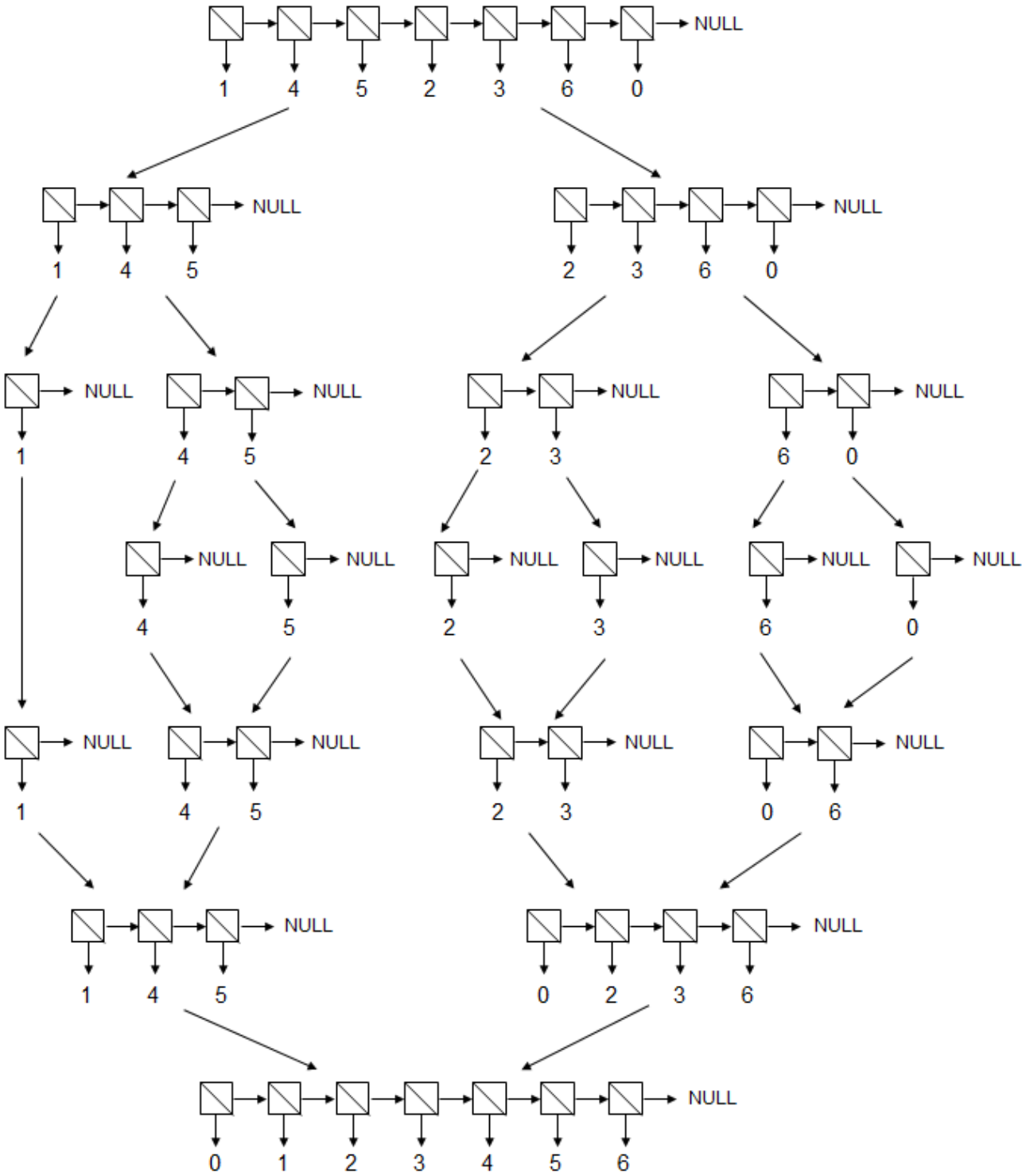


Рисунок 5.3 – Приклад сортування злиттям для зв’язного списку

Текст програми для сортування злиттям має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <time.h>
#define MAXLIST 10

typedef struct node
{
    int number;
    struct node *next;
} LNode;
LNode *head, *visit;
void node_add(LNode **q, int num);
int Length(LNode *head);
void PrintList();
void FrontBackSplit(LNode *source, LNode **frontRef,
                    LNode **backRef);
LNode *SortedMerge(LNode *a, LNode *b);
void MergeSort(LNode **headRef);
int main(void)
{
    LNode *newnode = NULL;
    int i = 0;
    SetConsoleOutputCP(1251);
    srand(time(NULL));
    for(i = 0; i < MAXLIST; i++)
        node_add(&newnode, rand() % 100);
    head = newnode;
    printf("=======================================\n");
    printf("Список до сортування злиттям:          \n");
    printf("=======================================\n");
    PrintList();
    MergeSort(&head);
    printf("=======================================\n");
    printf("Список після сортування злиттям:       \n");
    printf("=======================================\n");
    PrintList();
    printf("=======================================\n");
    return 0;
}
void node_add(LNode **q, int num)
{
    LNode *tmp;
    tmp = *q;
    if(*q == NULL)
    {
        *q = (LNode *) malloc(sizeof(LNode));
        tmp = *q;
    }
    else
    {
```

```
        while(tmp->next != NULL)
            tmp = tmp->next;
        tmp->next = (LNode *) malloc(sizeof(LNode));
        tmp = tmp->next;
    }
    tmp->number = num;
    tmp->next = NULL;
}
int Length(LNode *head)
{
    int count = 0;
    LNode *current = head;
    while(current != NULL)
    {
        count++;
        current = current->next;
    }
    return(count);
}
void PrintList()
{
    visit = head;
    while(visit != NULL)
    {
        printf("%4d", visit->number);
        visit = visit->next;
    }
    printf("\n");
}
void FrontBackSplit(LNode *source, LNode **frontRef,
                    LNode **backRef)
{
    int len = Length(source);
    int i;
    LNode *current = source;
    if(len < 2)
    {
        *frontRef = source;
        *backRef  = NULL;
    }
    else
    {
        int hopCount = (len - 1) / 2;
        for(i = 0; i < hopCount; i++)
            current = current->next;
        // список поділяється на дві частини
        *frontRef = source;
        *backRef = current->next;
        current->next = NULL;
    }
}
```

```
LNode *SortedMerge(LNode *a, LNode *b)
{
    LNode *result = NULL;
    if(a == NULL)
        return(b);
    else
    {
        if(b == NULL)
            return(a);
        if(a->number <= b->number)
        {
            result = a;
            result->next = SortedMerge(a->next, b);
        }
        else
        {
            result = b;
            result->next = SortedMerge(a, b->next);
        }
    }
    return(result);
}
void MergeSort(LNode **headRef)
{
    LNode *head = *headRef;
    LNode *a;
    LNode *b;
    // вироджений випадок - довжина списку дорівнює 0 або 1
    if((head == NULL) || (head->next == NULL))
        return;
    FrontBackSplit(head, &a, &b);
    MergeSort(&a);   // рекурсивне сортування списку a
    MergeSort(&b);   // рекурсивне сортування списку b
    *headRef  = SortedMerge(a, b);
}
```

Результат роботи програми наведено на рис. 5.4.

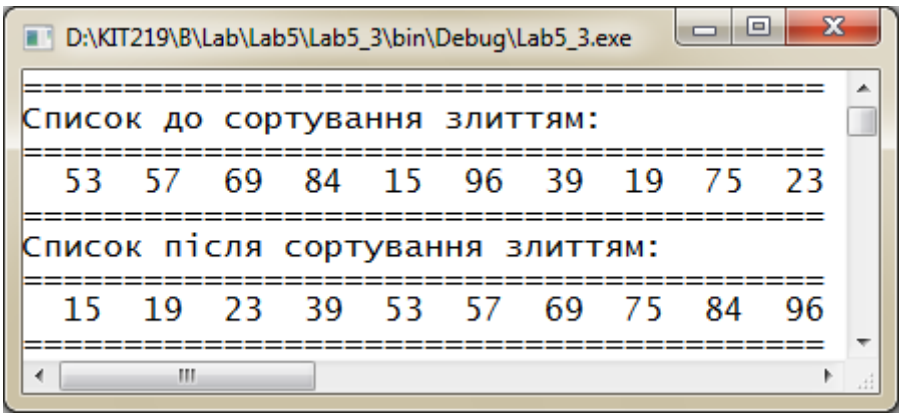


Рисунок 5.4 – Результат роботи програми сортування злиттям

5.4. Порівняння різних алгоритмів сортування

Після розгляду різних алгоритмів сортування можна зробити їх порівнювальне тестування. В якості об'єкту тестування будуть списки довжиною 25000, 30000, 35000 і 40000 елементів, при цьому ключ в елементі може приймати значення від 0 до `RAND_MAX`. Елементи вставляються в довільному порядку.

В ході тестування для різних типів сортування буде фіксуватися фінальний час виконання програми. Найбільший інтерес буде мати час, який безпосередньо витрачається на сортування.

Нижче наведено код функції `main()` для оцінювання продуктивності бульбашкового сортування. Текст програми має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <time.h>
#define MAX 25000L

int main(void)
{
    LNode *newnode = NULL;

    SetConsoleOutputCP(1251);

    srand(time(NULL));
    printf("Зачекайте, відбувається бульбашкове "
           "сортування...\n");
    for(unsigned long i = 0; i < MAX; i++)
        llist_add(&newnode, rand() % RAND_MAX);
    head = newnode;
    clock_t start = clock();
    llist_bubble_sort();
    clock_t end = clock();
    double seconds = (double)(end - start) / CLOCKS_PER_SEC;
    printf("=============================================\n");
    printf("Кількість елементів списку:   %ld\n", MAX);
    printf("Час бульбашкового сортування: %lf\n", seconds);
    printf("=============================================\n");
    return 0;
}
```

Результат виконання програми для різної кількості елементів у списку наведено на рис. 5.5 – 5.8. З отриманих результатів видно, що час бульбашкового сортування збільшується приблизно від 4.5 секунди до 11.6 секунди.

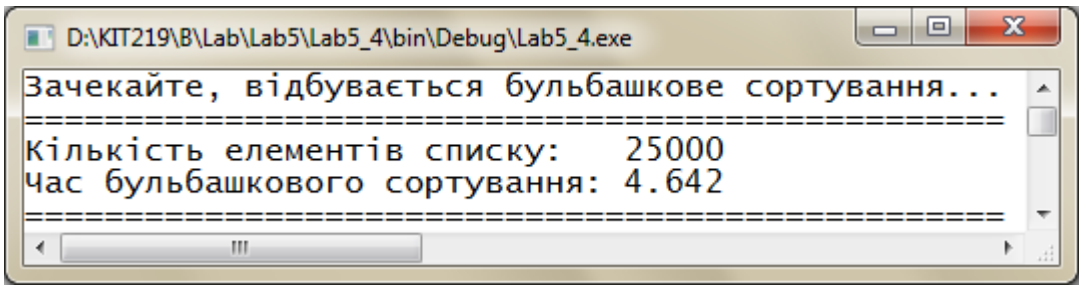


Рисунок 5.5 – Результат бульбашкового сортування для 25000 елементів

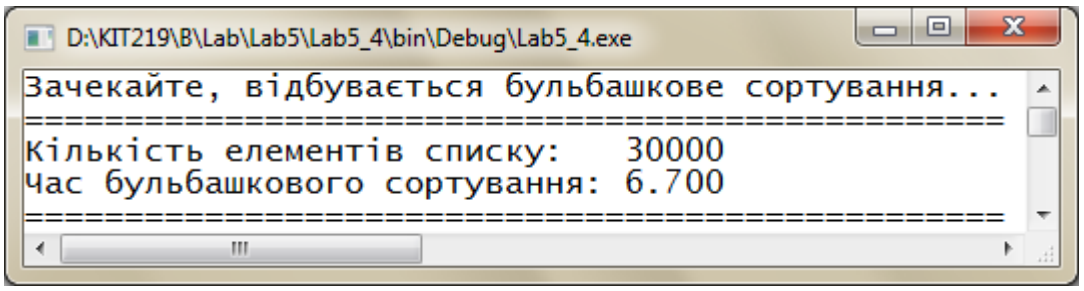


Рисунок 5.6 – Результат бульбашкового сортування для 30000 елементів

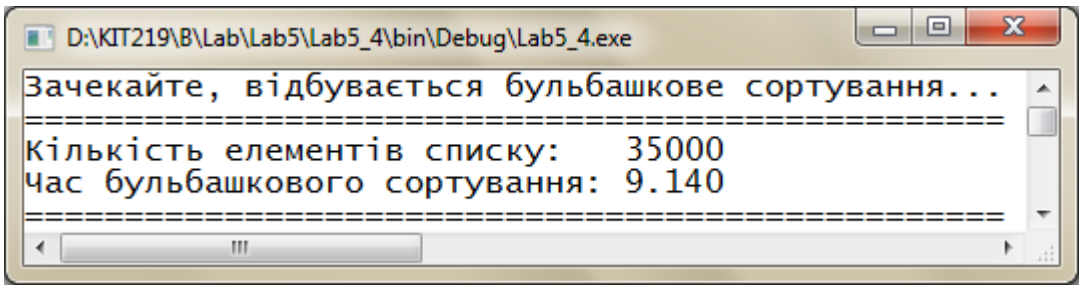


Рисунок 5.7 – Результат бульбашкового сортування для 35000 елементів

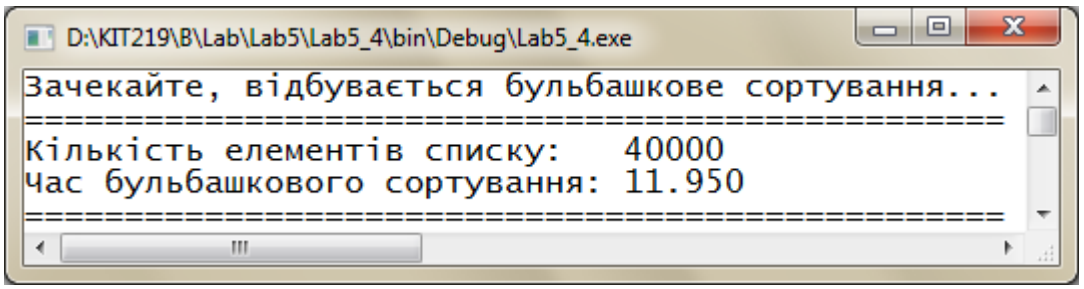


Рисунок 5.8 – Результат бульбашкового сортування для 40000 елементів

Код для перевірки продуктивності швидкого сортування має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <time.h>
#define MAX_DLIST 25000L
int main(void)
{
    SetConsoleOutputCP(1251);
    srand(time(NULL));
    printf("Зачекайте, відбувається швидке сортування...\n");
    for(unsigned long i = 0; i < MAX_DLIST; i++)
        AddListItem(rand() % RAND_MAX);
    clock_t start = clock();
    QuickSortList(pFirstItem, pLastItem);
    clock_t end = clock();
    double seconds = (double)(end - start) / CLOCKS_PER_SEC;
    printf("=============================================\n");
    printf("Кількість елементів списку: %ld\n", MAX_DLIST);
    printf("Час швидкого сортування:    %.3lf\n", seconds);
    printf("=============================================\n");
    RemoveAllItems();
    return 0;
}
```

Результат виконання програми для різної кількості елементів у списку наведено на рис. 5.9 – 5.12. З отриманих результатів видно, що час швидкого сортування збільшується приблизно від 2.1 секунди до 5.3 секунди.

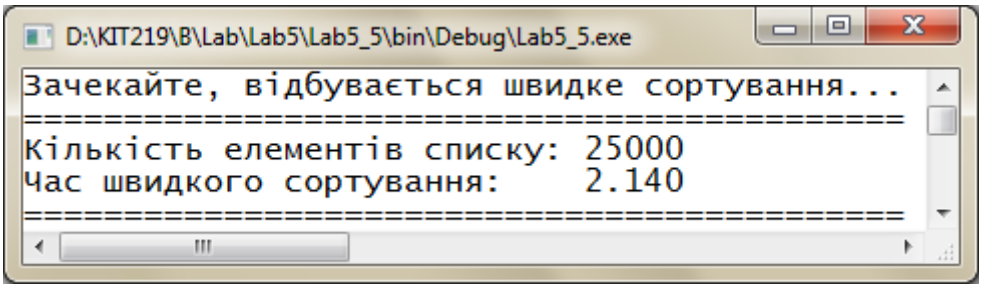


Рисунок 5.9 – Результат швидкого сортування для 25000 елементів

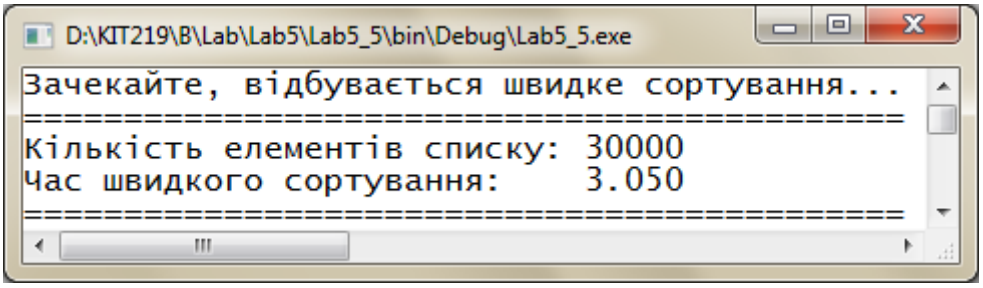


Рисунок 5.10 – Результат швидкого сортування для 30000 елементів

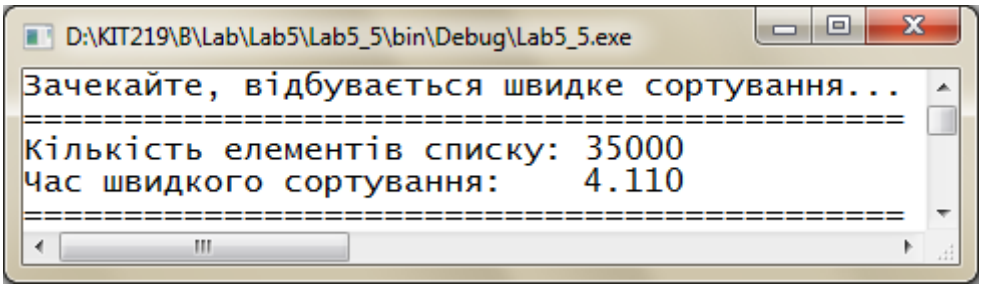


Рисунок 5.11 – Результат швидкого сортування для 35000 елементів

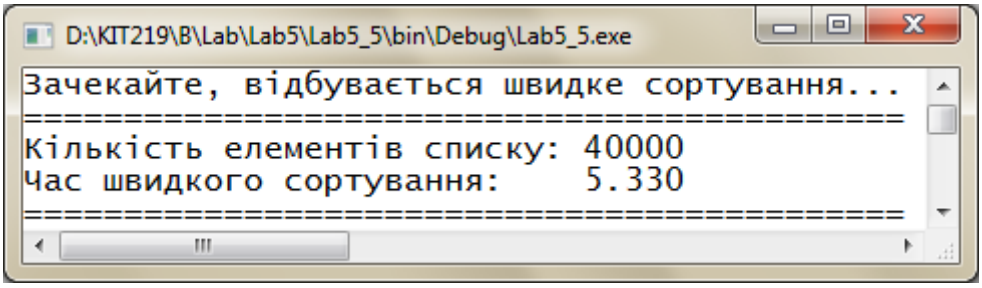


Рисунок 5.12 – Результат швидкого сортування для 40000 елементів

Намагання збільшити кількість елементів списку до 45000 призвело до завершення програми через нестачу пам’яті.

У випадку сортування злиттям головна проблема пов’язана зі збільшенням стеку при рекурсії, тому при збільшенні довжини списку більш ніж 40000 елементів можуть виникнути проблеми. Функція `main()` для сортування злиттям має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <stdlib.h>
#include <time.h>
#define MAXLIST 25000L
int main(void)
{
    LNode *newnode = NULL;
    SetConsoleOutputCP(1251);
    srand(time(NULL));
    printf("Зачекайте, відбувається сортування злиттям...\n");
    for(unsigned long i = 0; i < MAXLIST; i++)
        node_add(&newnode, rand() % RAND_MAX);
    head = newnode;
    clock_t start = clock();
    MergeSort(&head);
    clock_t end = clock();
```

```
    double seconds = (double)(end - start) / CLOCKS_PER_SEC;
    printf("=============================================\n");
    printf("Кількість елементів списку: %ld\n", MAXLIST);
    printf("Час сортування злиттям:     %.3lf\n", seconds);
    printf("=============================================\n");
    return 0;
}
```

Результат виконання програми сортування злиттям для різної кількості елементів у списку наведено на рис. 5.13 – 5.16. З отриманих результатів видно, що час сортування злиттям збільшується приблизно від 0.01 секунди до 0.02 секунди. Намагання збільшити кількість елементів списку до 45000 призвело до завершення програми через нестачу пам’яті.

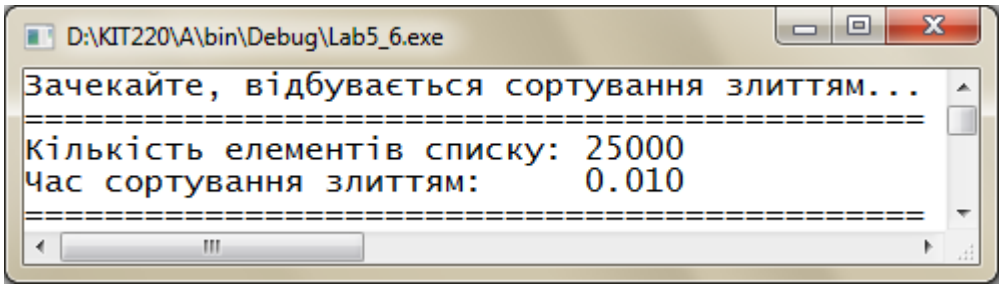


Рисунок 5.13 – Результат сортування злиттям для 25000 елементів

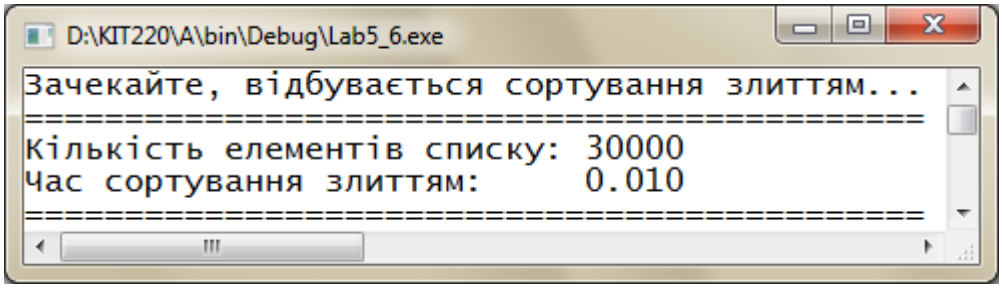


Рисунок 5.14 – Результат сортування злиттям для 30000 елементів

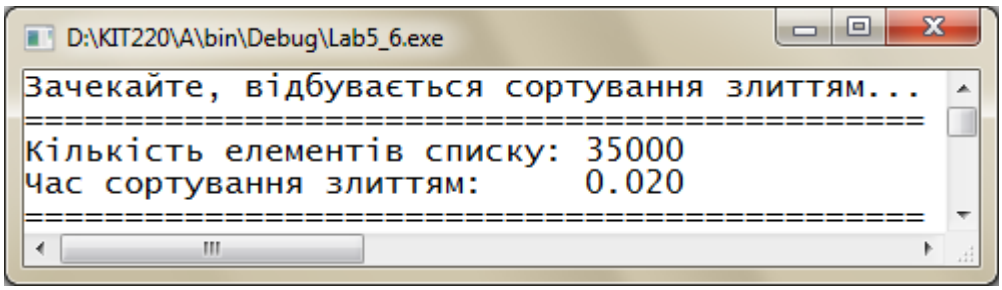


Рисунок 5.15 – Результат сортування злиттям для 35000 елементів

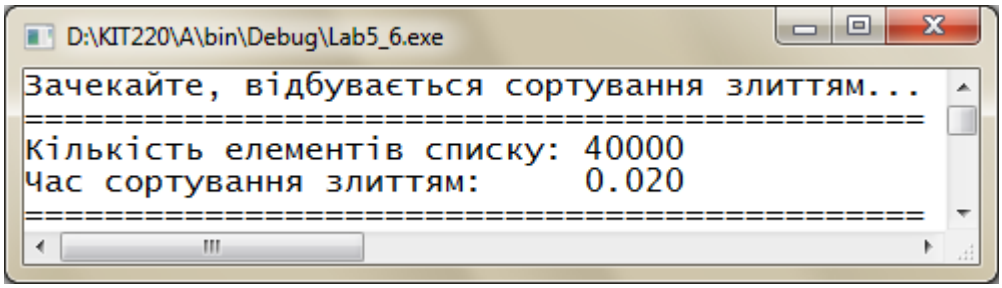


Рисунок 5.16 – Результат сортування злиттям для 40000 елементів

Слід врахувати, що це тестування не претендує на звання «абсолютної істини», оскільки мова йде про конкретну реалізацію, яка застосовується на конкретній системі. Крім того, час, що визначається в операційній системі Windows не претендує на точність, бо протягом виконання програми він може витрачатися на роботу будь–яких інших задач. Найгірший результат, як і очікувалося, показало бульбашкове сортування, яке на те, щоб упорядкувати зв'язний список з 40000 елементів, витратило приблизно 11.9 секунд. Швидке сортування впоралось з цією самою задачею приблизно за 5.3 секунди. Найпродуктивнішим виявилося сортування злиттям, виконавши задачу ще швидше – приблизно за 0.02 секунди.

Також тестування виявило наступні закономірності.

1) Найбільшу кількість переставлень потребує бульбашкове сортування. Потім йде швидке сортування, і менш за все переставлень необхідно для сортування злиттям.

2) При збільшенні довжини списку продуктивність бульбашкового сортування починає знижуватися. Для прикладу можна навести графік залежності кількості елементів від часу виконання в секундах для бульбашкового сортування (рис. 5.17). З графіку видно, що при використанні 40000 елементів час виконання функції бульбашкового сортування складає приблизно 12 секунд, а при використанні 500000 елементів приблизно 71 хвилину (4274 секунди).

3) Хоча бульбашкове сортування і виконувалося найдовше, проте вдалося отримати результат для 500000 елементів (далі експеримент не проводився че-

рез великий час очікування результату). Щодо швидкого сортування і сортування злиттям, то для 45000 елементів обидва експерименти завершилися невдачею через нестачу пам’яті.

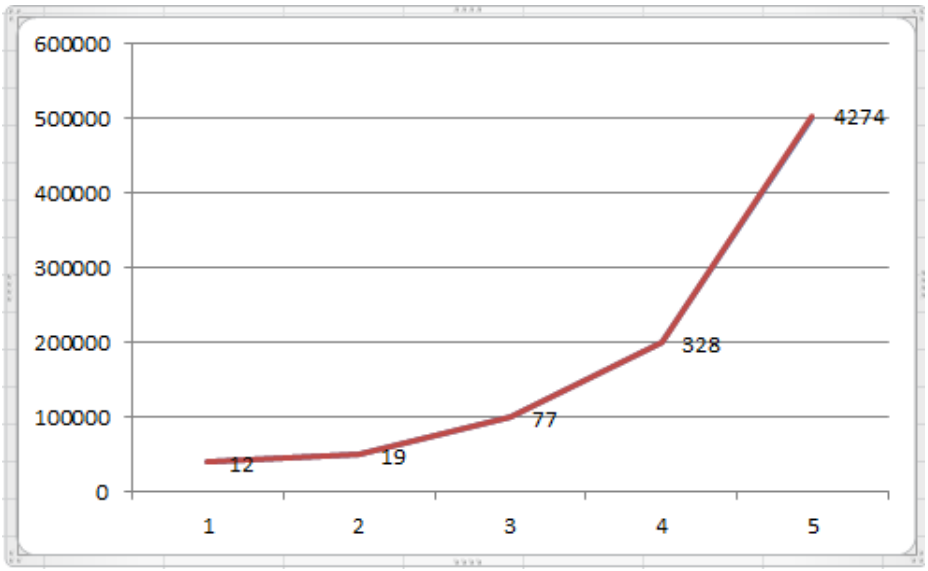


Рисунок 5.17 – Графік залежності кількості елементів у списку від часу (в секундах) для бульбашкового сортування

Таким чином, при використанні різних алгоритмів для сортування зв’язних списків можна домогтися дуже високої продуктивності, незважаючи на великий розмір списку. Однак кінцевий результат залежить від цілого комплексу чинників: ефективного використання пам’яті, глибини рекурсії, продуктивності цільової системи та інших факторів.

Хід виконання роботи:

Напишіть шість *C*–програм для вказаних нижче завдань. Дані для свого варіанта візьміть у табл. 5.1.

1. Відсортуйте зв’язний список, елементами якого є випадкові цілі числа від **`a`** до **`b`**, за допомогою бульбашкового методу. Кількість елементів у списку повинна дорівнювати **`k`**. Напрямок сортування: ↗ – за збільшенням; ↘ – за зменшенням. Виведіть на екран вхідний і відсортований списки.

Таблиця 5.1 – Дані для виконання завдань

№ вар.	Номер завдання														
	1				2				3				4		
	k	**a**	**b**	сорт.	**k**	**a**	**b**	сорт.	**k**	**a**	**b**	сорт.	**a**	**b**	**n**
1	11	34	89	↗	26	25	78	↘	15	11	42	↗	23	2356	23000
2	12	22	77	↘	25	43	94	↗	16	42	76	↘	15	3429	22500
3	13	75	89	↗	24	32	87	↘	17	38	59	↗	62	4391	22000
4	14	14	65	↘	23	37	66	↗	18	42	77	↘	28	5459	21500
5	15	23	45	↗	22	31	83	↘	19	52	88	↗	27	3571	21000
6	16	41	74	↘	21	16	54	↗	20	21	82	↘	18	3629	20500
7	17	27	48	↗	20	21	69	↘	21	32	57	↗	29	4349	20000
8	18	17	82	↘	19	42	76	↗	22	43	76	↘	17	4983	19500
9	19	29	48	↗	18	13	37	↘	11	11	76	↗	28	4092	19000
10	20	46	78	↘	17	32	68	↗	12	25	54	↘	37	3962	18500
11	21	31	63	↗	16	61	94	↘	13	17	29	↗	43	1849	18000
12	22	73	95	↘	15	37	63	↗	14	25	46	↘	28	3209	17500
13	23	52	86	↗	14	44	55	↘	26	37	59	↗	38	3459	17000
14	24	24	68	↘	13	33	84	↗	25	15	27	↘	45	3982	16500
15	25	61	96	↗	12	51	72	↘	24	25	47	↗	42	4398	16000
16	26	32	67	↘	11	27	38	↗	23	18	28	↘	36	3297	15500
17	14	22	58	↗	23	15	46	↘	18	16	67	↗	13	3256	23000
18	13	35	63	↘	24	25	74	↗	17	30	71	↘	18	3769	22500
19	12	43	58	↗	25	36	67	↘	16	47	89	↗	22	3391	22000
20	11	16	47	↘	26	31	63	↗	15	31	47	↘	78	4459	21500

2. Відсортуйте методом швидкого сортування двобічний зв’язний список, елементами якого є випадкові цілі числа від **a** до **b**. Кількість елементів у списку повинна дорівнювати **k**. Напрямок сортування: ↗ – за збільшенням; ↘ – за зменшенням. Виведіть на екран вхідний і відсортований списки.

3. Відсортуйте методом злиття зв’язний список, елементами якого є цілі числа від **a** до **b**. Кількість елементів у списку повинна дорівнювати **k**. Напрямок сортування: ↗ – за збільшенням; ↘ – за зменшенням. Виведіть на екран вхідний і відсортований списки.

4. Порівняйте час виконання розглянутих у роботі методів сортування за зменшенням для списку, який містить **n** елементів цілого типу, що знаходяться в діапазоні від **a** до **b**. Зробіть висновки щодо найбільш продуктивного методу сортування.

Примітка: При виконанні четвертого завдання у звіт необхідно розміщати тексти трьох програм лише для функцій `main()`. Якщо під час виконання програм для вашого варіанта будь яка з трьох програм не зможе виконатися через нестачу памяті, необхідно знизити кількість елементів до такого значення, яке дозволить спрацювати усім програмам.

Звіт про виконання лабораторної роботи повинен містити:

1. Титульний аркуш.
2. Мету роботи.
3. Результати виконання роботи:

   – тексти шести *C*–програм (для програм з визначення часу виконання методів сортування навести лише тексти функцій `main()`);

   – результати виконання *C*–програм (6 копій екранів).
4. Висновки по роботі.

Контрольні запитання:

1. Що собою являє процес сортування?
2. Які методи сортування зв’язних списків були розглянуті в цій роботі?
3. Які існують загальні критерії оцінки алгоритмів сортування?
4. Який з методів сортування виявився найбільш продуктивним?
5. Які з методів сортування мали проблеми з нестачею пам’яті при збільшенні кількості елементів списку?
6. В чому полягає принцип бульбашкового сортування?
7. В чому полягає принцип швидкого сортування?
8. В чому полягає принцип сортування злиттям?

Лабораторна робота №6

ДОСЛІДЖЕННЯ ОСНОВНИХ ФУНКЦІЙ ДЛЯ РОБОТИ З ДЕКОМ

Мета роботи: ознайомитися з основними поняттями та операціями, що застосовуються при роботі з двозв'язними лінійними списками (ДЛС); розглянути вміст основних функцій для роботи з деком та навчитися застосовувати їх у *С*–програмах.

Теоретична частина

Двозв'язний лінійний список (дек) – послідовність елементів, кожен з яких крім поля з даними містить поля вказівників на наступний і попередній елементи списку [9]. Вказівники на попередній і наступний вузли списку (для голови і хвоста відповідно) містять нульове значення (рис. 6.1).

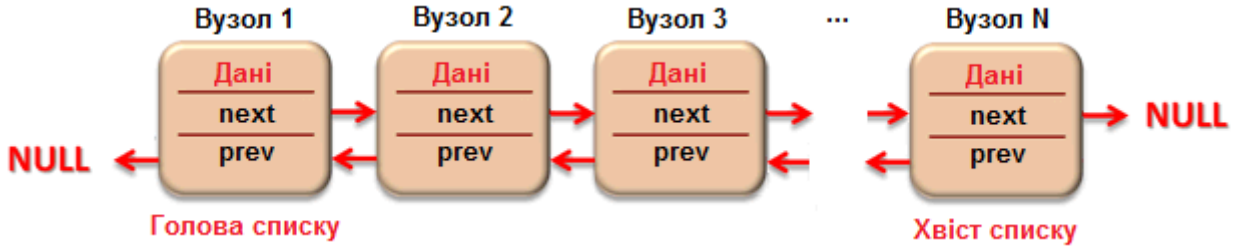


Рисунок 6.1 – Двозв'язний лінійний список

Використання двох вказівників дає декілька переваг, найголовнішою з яких є можливість переміщатися по списку в двох напрямках. Завдяки цьому спрощуються деякі операції (у порівнянні з однозв'язним лінійним списком) і додаються нові.

Вузол ДЛС можна представити у вигляді структури **`list`**:

```
typedef struct list
{
    int value;           // поле даних
    struct list *next;   // вказівник на наступний елемент
    struct list *prev;   // вказівник на попередній елемент
} List;
```

Основні операції, які можна виконувати над вузлами ДЛС:

– ініціалізація списку;

– додавання вузла до списку;

– видалення вузла зі списку;

– видалення голови та хвоста списку;

– вивід елементів списку у прямому та зворотному порядку;

– взаємообмін вузлів (обмін місцями двох вузлів).

6.1. Ініціалізація ДЛС

Ініціалізація деку призначена для створення головного вузла (голови) списку, у якого поля вказівників на наступний і попередній вузли містить нульове значення (рис. 6.2). Текст функції `initList()` наведено нижче.

Рисунок 6.2 – Ініціалізація ДЛС

```
List *initList(int data)
{
    List *list;
    // виділення пам'яті під голову списку
    list = (List *) malloc(sizeof(List));
    list->value = data;
    list->next = NULL;      // вказівник на наступний вузол
    list->prev = NULL;      // вказівник на попередній вузол
    return(list);
}
```

6.2. Додавання вузла до ДЛС

Функція додавання вузла до двозв'язного списку приймає два аргументи:

– вказівник на вузол, після якого буде відбуватися додавання;

– дані для вузла, що додається.

Процедуру додавання вузла можна відобразити схемою на рис. 6.3.

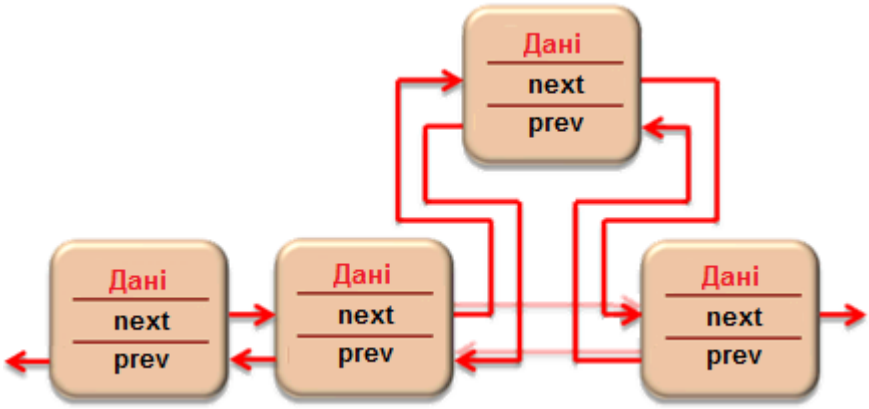


Рисунок 6.3 – Додавання вузла до ДЛС

Додавання вузла до ДЛС складається з наступних етапів:

– створення вузла елемента, що додається, та заповнення його даними;

– спрямування вказівника **next** вузла, який передує вузлу, що додається, на вузол, що додається;

– спрямування вказівника **prev** вузла, що йде за доданим, на вузол, що додається;

– встановлення вказівника **next** вузла, що додається, на наступний вузол (той, на який вказував попередній вузол);

– встановлення вказівника **prev** вузла, який додається, на вузол, що передує вузлу, який додається (вузол, що передається до функції).

Таким чином, функція додавання вузла до ДЛС має такий вигляд:

```
List *addElem(List *list, int data)
{
    List *tmp, *p;
    tmp = (List *) malloc(sizeof(List));
    // зберігаємо вказівник на наступний вузол
    p = list->next;
    // попередній вузол посилається на щойно створений
    list->next = tmp;
    // зберігання поля даних вузла, що додається
    tmp->value = data;
    // створений вузол вказує на наступний вузол
    tmp->next = p;
    // створений вузол вказує на попередній вузол
    tmp->prev = list;
    if(p != NULL) p->prev = tmp;
    return(tmp);
}
```

Значенням функції, що повертається, є адреса доданого вузла.

6.3. Видалення вузла ДЛС

В якості аргументу функції видалення вузла ДЛС передається вказівник на вузол, що видаляється. Оскільки вузол списку має поле вказівника на попередній вузол, немає необхідності передавати вказівник на голову списку.

Функція повертає вказівник на вузол, що йде за вузлом, який видаляється.

Видалення вузла може бути представлено схемою на рис. 6.4.

Рисунок 6.4 – Видалення вузла ДЛС

Видалення вузла ДЛС складається з наступних етапів:

– встановлення вказівника **next** попереднього вузла на наступний вузол, що йде за тим, що видаляється;

– встановлення вказівника **prev** наступного вузла на вузол, який передує вузлу, що видаляється;

– звільнення пам’яті для вузла, що видаляється.

```
List *deleteElem(List *list)
{
    List *prev;
    List *next;

    prev = list->prev;           // вузол, що передує list
    next = list->next;           // вузол, що йде за list

    if(prev != NULL)
        prev->next = list->next; // переставляємо вказівник
    if(next != NULL)
        next->prev = list->prev; // переставляємо вказівник

    free(list);                  // звільняємо пам'ять для
                                 // елемента, що видаляється
    return(prev);
}
```

6.4. Видалення голови ДЛС

Функція видалення голови ДЛС в якості аргументу отримує вказівник на поточну голову списку. Значенням, що повертається, буде нова голова списку – той вузол, на який вказує голова, що видаляється.

```
List *deleteHead(List *head)
{
    List *tmp;

    tmp = head->next;
    tmp->prev = NULL;
    free(head);       // звільнення пам'яті для поточної голови
    return(tmp);      // нова голова списку
}
```

6.5. Вивід елементів ДЛС на екран

Функція виводу елементів ДЛС на екран аналогічна до функції для однозв'язного лінійного списку (ОЛС).

В якості аргументу функції виводу елементів на екран передається вказівник на голову списку.

Функція здійснює послідовний обхід усіх вузлів з подальшим виводом їх значень на екран.

```
void listPrint(List *list)
{
    List *p;

    p = list;
    do
    {
       printf("%d ", p->value); // вивід значення елемента p
       p = p->next;             // перехід до наступного вузла
    } while(p != NULL);         // умова закінчення обходу
}
```

6.6. Вивід елементів ДЛС у зворотному порядку

Для ДЛС також можна викликати функцію виводу елементів списку у зворотному порядку, яка приймає в якості аргументу вказівник на голову списку. Функція переміщує вказівник на останній вузол списку та здійснює послідовний обхід усіх вузлів з виводом їх значень у зворотному порядку.

```
void listPrintRevers(List *list)
{
    List *p;
    p = list;
    while(p->next != NULL)
        p = p->next;                // перехід на кінець списку
    do
    {
        printf("%d ", p->value);    // вивід значення елемента p
        p = p->prev;                // перехід до попереднього
                                    // вузла
    } while(p != NULL);             // умова закінчення обходу
}
```

6.7. Обмін вузлами ДЛС

В якості аргументів функція взаємообміну вузлів ДЛС приймає два вказівника на вузли, що міняються місцями, а також вказівник на голову списку. Функція повертає адресу голови списку.

Взаємообмін вузлів списку здійснюється шляхом переустановлення вказівників. Для цього необхідно визначити попередній і наступний вузли для кожного вузла, що підлягає обміну. При цьому можливі дві ситуації:

– вузли, що підлягають обміну, є сусідніми;

– вузли, що підлягають обміну, не є сусідніми (тобто між ними існує хоча б один вузол).

Під час обміну сусідніх вузлів переустановлення вказівників відбувається за принципом, що зображений на рис. 6.5.

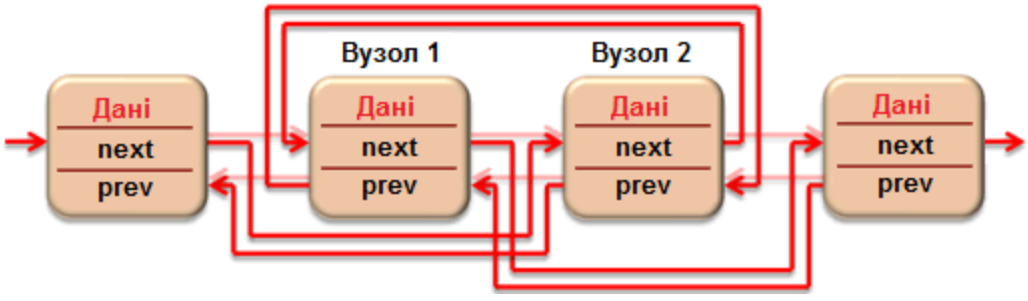


Рисунок 6.5 – Принцип обміну місцями двох сусідніх вузлів

Під час обміну вузлів, що не є сусідніми, переустановлення вказівників відбувається за принципом, що зображений на рис. 6.6.

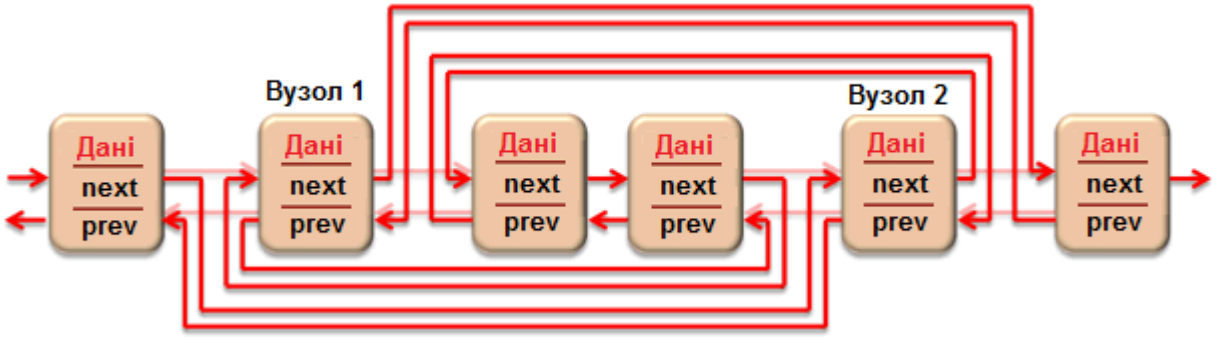


Рисунок 6.6 – Принцип обміну місцями двох вузлів, що не є сусідніми

Функція взаємообміну вузлів списку виглядає наступним чином:

```
List *swap(List *klist1, List *klist2, List *head)
{
    // Повертає нову голову списку
    List *prev1, *prev2, *next1, *next2;

    prev1 = klist1->prev;      // вузол, що передує klist1
    prev2 = klist2->prev;      // вузол, що передує klist2
    next1 = klist1->next;      // вузол, що йде після klist1
    next2 = klist2->next;      // вузол, що йде після klist2
    if(klist2 == next1)        // обмін сусідніми вузлами
    {
        klist2->next = klist1;
        klist2->prev = prev1;
        klist1->next = next2;
        klist1->prev = klist2;
        if(next2 != NULL)
            next2->prev = klist1;
        if(klist1 != head)
            prev1->next = klist2;
    }
    else
    {
        if(klist1 == next2) // обмін сусідніми вузлами
        {
            klist1->next = klist2;
            klist1->prev = prev2;
            klist2->next = next1;
            klist2->prev = klist1;

            if(next1 != NULL)
                next1->prev = klist2;
            if(klist2 != head)
                prev2->next = klist1;
        }
        else  // обмін вузлів, які не є сусідніми
        {
              // вказівник prev можна встановити тільки для
```

```
                // елемента, що не є головою
                if(klist1 != head)
                    prev1->next = klist2;
                klist2->next = next1;
                if(klist2 != head)
                    prev2->next = klist1;
                klist1->next = next2;
                klist2->prev = prev1;
                // вказівник next можна встановити тільки для
                // елемента, що не є останнім
                if(next2 != NULL)
                    next2->prev = klist1;
                klist1->prev = prev2;
                if(next1 != NULL)
                    next1->prev = klist2;
            }
        }
        if(klist1 == head)
            return(klist2);
        if(klist2 == head)
            return(klist1);
        return(head);
}
```

6.8. Пошук необхідного вузла ДЛС

Для реалізації функції взаємообміну вузлів ДЛС дуже корисною може бути функція, що знаходить адресу вузла, який необхідно обміняти місцями з іншим вузлом. В якості параметрів функції передається номер вузла, адресу якого необхідно знайти, та голова ДЛС. Текст цієї функції має такий вигляд:

```
List *getNth(int k, List *head)
{
    List *tmp = head;
    size_t i = 0;

    while(tmp && i < k-1)
    {
        tmp = tmp->next;
        i++;
    }
    return tmp;
}
```

Повний код програми для реалізації деяких розроблених функцій при роботі з деком має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
```

```
#include <stdlib.h>
#define  MAX   6
typedef struct list
{
    int value;             // поле даних
    struct list *next;     // вказівник на наступний елемент
    struct list *prev;     // вказівник на попередній елемент
} List;

List *initList(int data);
List *addElem(List *list, int data);
List *deleteElem(List *list);
List *deleteHead(List *head);
void  listPrint(List *list);
void  listPrintRevers(List *list);
List *swap(List *klist1, List *klist2, List *head);
List *getNth(int k, List *head);

int main(void)
{
    List *head;
    List *cur1;
    List *cur2;
    int num;
    int i = 1;

    SetConsoleOutputCP(1251);

    // Створюємо список з MAX елементів
    printf("Введіть елемент списку №%d: ", i);
    scanf("%d", &num);
    head = initList(num);
    cur1 = head;
    for(i = 0; i < MAX-1; i++)
    {
        printf("Введіть елемент списку №%d: ", i+2);
        scanf("%d", &num);
        cur1 = addElem(cur1, num);
    }
    printf("\nЕлементи списку\n");
    printf("==============================\n");
    listPrint(head);
    printf("\n==============================\n");
    printf("\nМіняємо місцями елементи %d і %d\n", 1, 2);
    cur1 = getNth(1, head);
    cur2 = getNth(2, head);
    head = swap(cur1, cur2, head);
    printf("==============================\n");
    listPrint(head);
    printf("\n==============================\n");
    printf("\nВидаляємо елемент №%d\n", 3);
    cur1 = head->next->next;
    deleteElem(cur1);
    printf("==============================\n");
```

```
    listPrint(head);
    printf("\n==============================\n\n");
    return 0;
}

List *initList(int data)
{
    List *list;
    // виділення пам'яті під голову списку
    list = (List *) malloc(sizeof(List));
    list->value = data;
    list->next = NULL;  // вказівник на наступний вузол
    list->prev = NULL;  // вказівник на попередній вузол
    return(list);
}

List *addElem(List *list, int data)
{
    List *tmp, *p;
    tmp = (List *) malloc(sizeof(List));
    // зберігаємо вказівник на наступний вузол
    p = list->next;
    // попередній вузол посилається на той,
    // який щойно створено
    list->next = tmp;
    // зберігання поля даних вузла, що додається
    tmp->value = data;
    // створений вузол вказує на наступний вузол
    tmp->next = p;
    // створений вузол вказує на попередній вузол
    tmp->prev = list;
    if(p != NULL)
        p->prev = tmp;
    return(tmp);
}

List *deleteElem(List *list)
{
    List *prev;
    List *next;
    prev = list->prev;           // вузол, що передує list
    next = list->next;           // вузол, що йде за list
    if(prev != NULL)
        prev->next = list->next; // переставляємо вказівник
    if(next != NULL)
        next->prev = list->prev; // переставляємо вказівник
    free(list);                  // звільняємо пам'ять для
                                 // елемента, що видаляється
    return(prev);
}

List *deleteHead(List *head)
{
    List *tmp;
```

```
    tmp = head->next;
    tmp->prev = NULL;
    free(head);       // звільнення пам'яті для поточної голови
    return(tmp);      // нова голова списку
}
void listPrint(List *list)
{
    List *p;

    p = list;
    do
    {
        printf("%d ", p->value); // вивід значення елемента p
        p = p->next;            // перехід до наступного вузла
    } while(p != NULL);         // умова закінчення обходу
}

void listPrintRevers(List *list)
{
    List *p;

    p = list;
    while(p->next != NULL)
        p = p->next;             // перехід на кінець списку
    do
    {
        printf("%d ", p->value); // вивід значення елемента p
        p = p->prev;             // перехід до попереднього
                                 // вузла
    } while(p != NULL);          // умова закінчення обходу
}

List *swap(List *klist1, List *klist2, List *head)
{
    // Повертає нову голову списку
    List *prev1, *prev2, *next1, *next2;

    prev1 = klist1->prev;        // вузол, що передує klist1
    prev2 = klist2->prev;        // вузол, що передує klist2
    next1 = klist1->next;        // вузол, що йде після klist1
    next2 = klist2->next;        // вузол, що йде після klist2
    if(klist2 == next1)          // обмін сусідніми вузлами
    {
        klist2->next = klist1;
        klist2->prev = prev1;
        klist1->next = next2;
        klist1->prev = klist2;
        if(next2 != NULL)
            next2->prev = klist1;
        if(klist1 != head)
            prev1->next = klist2;
    }
```

```
    else
    {
        if(klist1 == next2)       // обмін сусідніми вузлами
        {
            klist1->next = klist2;
            klist1->prev = prev2;
            klist2->next = next1;
            klist2->prev = klist1;
            if(next1 != NULL)
                next1->prev = klist2;
            if(klist2 != head)
                prev2->next = klist1;
        }
        else     // обмін вузлів, які не є сусідніми
        {
            // вказівник prev можна встановити тільки для
            // елемента, що не є головою
            if(klist1 != head)
                prev1->next = klist2;
            klist2->next = next1;
            if(klist2 != head)
                prev2->next = klist1;
            klist1->next = next2;
            klist2->prev = prev1;
            // вказівник next можна встановити тільки для
            // елемента, що не є останнім
            if(next2 != NULL)
                next2->prev = klist1;
            klist1->prev = prev2;
            if(next1 != NULL)
                next1->prev = klist2;
        }
    }
    if(klist1 == head)
        return(klist2);
    if(klist2 == head)
        return(klist1);
    return(head);
}

List *getNth(int k, List *head)
{
    List *tmp = head;
    size_t i = 0;
    while(tmp && i < k - 1)
    {
        tmp = tmp->next;
        i++;
    }
    return tmp;
}
```

Результат роботи програми наведено на рис. 6.7.

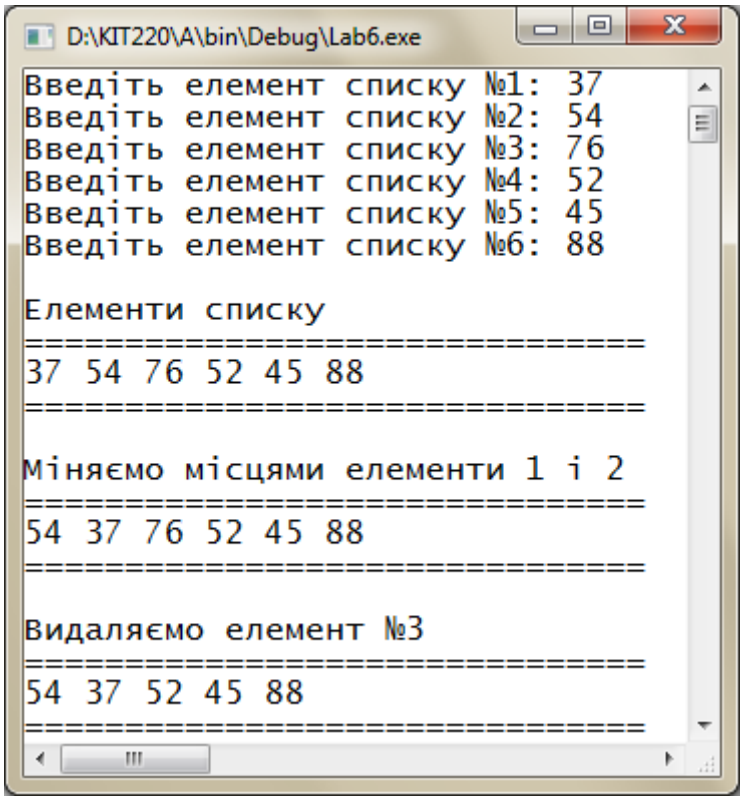


Рисунок 6.7 – Результат виконання програми для роботи з деком

Хід виконання роботи:

1. Опрацюйте матеріал лекції «Дек. Реалізація деку за допомогою списку» і теоретичний матеріал даної лабораторної роботи.

2. Напишіть *С*–програму, яка демонструє роботу деку, реалізованого за допомогою списку і виконує наступні дії:

– заповнює дек випадковими цілими числами в діапазоні від **a** до **b** (табл. 6.1) за принципом: до голови деку заносяться парні числа, а до хвоста деку – непарні (кількість чисел дорівнює **k**). Після занесення чергового числа необхідно виводити на екран вміст деку, щоб контролювати правильність наповнення деку;

– витягає з голови деку **n** чисел і виводить на екран вміст деку;

– витягає з хвоста деку **m** чисел і виводить на екран вміст деку;

Таблиця 6.1 – Дані для виконання завдань

Варіант	**k**	**a**	**b**	**n**	**m**	**n1**	**m1**
1	11	15	37	2	2	1	3
2	20	41	76	7	6	6	7
3	17	23	32	3	8	5	6
4	18	54	79	6	4	5	5
5	17	16	48	4	6	5	5
6	16	37	83	4	5	6	3
7	18	32	75	5	3	4	4
8	15	32	87	3	7	4	6
9	12	28	54	2	3	3	2
10	14	27	75	3	5	4	4
11	17	44	88	6	4	5	5
12	13	72	99	4	3	3	4
13	10	43	65	2	4	3	3
14	19	21	54	4	7	5	6
15	16	30	58	3	5	4	4
16	20	76	98	5	5	3	7
17	11	26	48	2	2	3	1
18	20	25	69	5	6	6	5
19	17	13	76	3	8	5	6
20	18	42	82	6	4	5	5

– виводить на екран суму чисел, які на даний момент знаходяться в деку;

– додає до голови **n1** випадкових чисел і виводить вміст деку на екран;

– додає до хвоста **m1** випадкових чисел і виводить вміст деку на екран;

– виводить на екран голову та хвіст деку, а також всі елементи деку у зворотному порядку;

– міняє місцями елементи **n2** і **m2**, які знаходяться поруч (сусідні) та елементи, між якими є хоча б один елемент. Значення **n2** і **m2** необхідно ввести з клавіатури на свій розсуд в залежності від кількості елементів деку. Після кожного обміну вміст деку треба вивести на екран.

Звіт про виконання лабораторної роботи повинен містити:

1. Титульний аркуш.

2. Мету роботи.

3. Результати виконання роботи:

– текст *C*–програми;

– результат виконання *C*–програми.

4. Висновки по роботі.

Контрольні запитання:

1. Що собою являє двозв'язний лінійний список і чим він відрізняється від однозв'язного лінійного списку?

2. Які основні операції можна робити з деком?

3. Що треба зробити для пошуку необхідного елемента в деку та за допомогою якої функції це можна здійснити?

4. Яка операція використовується найпершою при роботі з деком?

5. Які способи реалізації деку вам відомі?

6. Чим відрізняються функції `listPrint()` і `listPrintRevers()` та для чого вони призначені?

7. Яке призначення має функція `swap()` та що вона приймає в якості параметрів?

Лабораторна робота №7
ДОСЛІДЖЕННЯ ОСНОВНИХ ЗАСОБІВ ПРЕПРОЦЕСОРНОЇ ОБРОБКИ

Мета роботи: ознайомитися з принципами роботи препроцесора мови *C*; засвоїти навички роботи з директивами препроцесора: `#include`, `#define`, `#undef`, `#ifdef`, `#ifndef`, `#if`, `#else`, `#elif`, `#endif`.

Теоретична частина

7.1. Загальні відомості

До інтегрованого середовища *Code::Blocks* в якості обов'язкового компонента входить препроцесор. Основне призначення препроцесора – аналіз та

обробка початкового тексту програми до її компіляції. Можливості препроцесора є досить цікавими, особливо це стосується можливостей макрообробки та умовної компіляції [1–5]. Обробка текстів *C*–програм – це основна задача препроцесора, однак він може також перетворювати будь–які тексти.

На вхід препроцесора надходить текст з відповідними директивами, на виході – маємо перетворений початковий текст без препроцесорних директив.

7.2. Стадії препроцесорної обробки

Препроцесорна обробка тексту складається з декількох стадій, які виконуються в наступному порядку:

– усі позначення, які залежать від системи (наприклад, ознака кінця рядка) перекодовується на стандартні коди;

– кожна пара символів `'\'` і «символу нового рядка» разом з пробілами між ними прибираються, і тим самим наступний рядок початкового тексту приєднується до рядка, в якому знаходилася ця пара символів;

– в тексті кожного окремого рядка розпізнаються директиви та лексеми препроцесора, а кожен коментар замінюється одним символом пробілу;

– виконуються директиви препроцесора та проводяться макропідстановки;

– ескейп–послідовності в символьних константах і символьних рядках, наприклад, `'\n'` або `'\xF2'`, замінюються на їх еквіваленти (числові коди);

– проводиться конкатенація суміжних символьних рядків (рядкових констант), тобто вони об’єднуються в один рядок;

– кожна препроцесорна лексема перетворюється на текст на мові *C*.

До препроцесорних лексем або лексем препроцесора (preprocessing token) належать:

– символьні константи;

– імена файлів, що підключаються;

– ідентифікатори;

– знаки операцій;

– препроцесорні числа;

– розділові знаки;

– рядкові константи (рядки);

– будь–які символи, що не є пробілами.

Розглянемо більш докладно стадію обробки директив препроцесора.

Під час її виконання можливі наступні дії:

– заміна ідентифікаторів (позначень) наперед підготовленими послідовностями символів;

– включення до програми текстів з вказаних файлів;

– виключення з програми окремих частин її тексту (умовна компіляція);

– макропідстановка, тобто заміна позначення параметризованим текстом, що формується препроцесором з урахуванням конкретних параметрів.

Перед символом `'#'` і після нього в директиві дозволені пробіли. Вони також дозволені попереду лексем препроцесора, між ними та після них. Ознакою кінця директиви препроцесора є кінець текстового рядка (за наявності символу `'\'`, що означає перенесення рядка, закінченням препроцесорної директиви буде ознака кінця наступного рядка тексту).

Для препроцесора визначені наступні директиви:

`#define` – визначення ма́кросу або препроцесорного ідентифікатора;

`#include` – включення тексту з файлу;

`#undef` – відміна визначення макросу або ідентифікатора;

`#if` – перевірка умови – виразу;

`#ifdef` – перевірка визначеності ідентифікатора;

`#ifndef` – перевірка невизначеності ідентифікатора;

`#else` – початок альтернативної гілки для `#if`;

`#endif` – закінчення умовної директиви `#if`;

`#elif` – складена директива `#else...#if`;

`#line` – зміна номеру рядка, що розташований нижче;

`#error` – формування тексту повідомлення про помилку трансляції;

`#pragma` – дії, які передбачені реалізацією;

`#` – пуста директива.

Крім препроцесорних директив існують три препроцесорні операції, які докладніше будуть розглядатися разом з командою `#define`:

`defined` – перевірка істинності операнда;

`##` – конкатенація препроцесорних лексем;

`#` – перетворення операнда на рядок символів.

Директива `#define` має декілька модифікацій. Вони передбачають визначення макросів і препроцесорних ідентифікаторів, кожному з яких ставиться у відповідність певна символьна послідовність.

Директива `#include` дозволяє включати до тексту програми фрагмент з зазначеного файлу.

Директива `#undef` відміняє дію директиви `#define`, яка до цього визначила ім'я препроцесорного ідентифікатора.

Директива `#if` та її модифікації `#ifdef` і `#ifndef` спільно з директивами `#else`, **`#endif`** і `#elif` дозволяють організувати умовну обробку тексту програми. При використанні цих засобів компілюється не увесь текст, а тільки ті його частини, які виділені за допомогою перерахованих директив.

Директива `#line` дозволяє керувати нумерацією рядків у файлі з програмою. Ім'я файлу та необхідний початковий номер рядка зазначаються безпосередньо в директиві `#line`.

Директива `#error` дозволяє задати текст діагностичного повідомлення, яке виводиться при виникненні помилок.

Директива `#pragma` спричиняє дії, що залежать від реалізації.

Директива `#` нічого не робить та ігнорується препроцесором.

7.3. Заміни в тексті програми

Директива `#define` використовується для заміни обраного програмістом ідентифікатора заздалегідь підготовленою послідовністю:

```
#define ідентифікатор рядок_заміщення
```

Директива може розміщуватися в будь–якому місці тексту, а її дія зазвичай поширюється від позиції цієї директиви до кінця файлу. В результаті роботи препроцесора входження ідентифікатора, визначеного командою `#define`, в тексті програми заміняються рядком заміщення, закінченням якої є ознака кінця того «фізичного» рядка, де розміщена команда `#define`. Символи пробілів, що розташовані на початку та наприкінці рядка заміщення, при підстановці не використовується.

На рис. 7.1 наведено приклад використання директиви `#define`: ліворуч – текст *C*–програми до препроцесора; праворуч – після препроцесора.

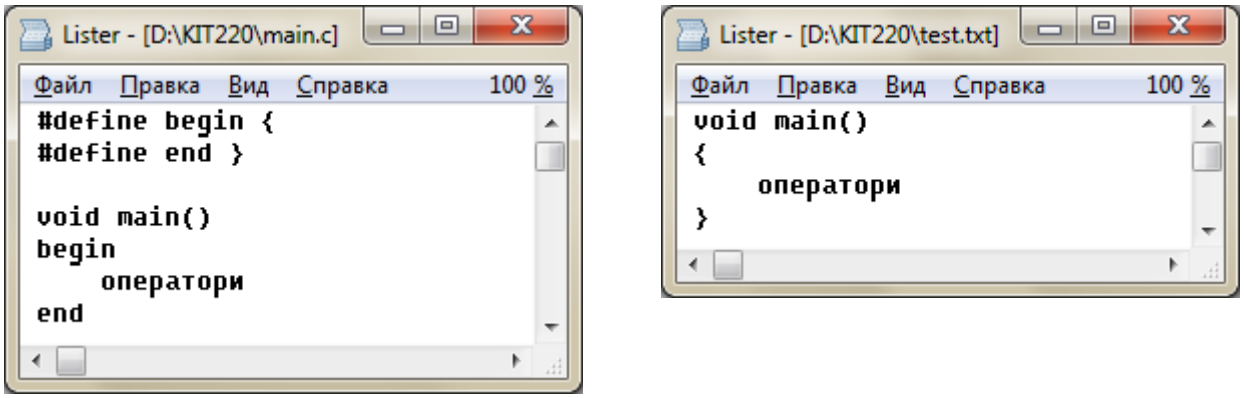


Рисунок 7.1 – Демонстрація результатів роботи препроцесора

Отримати цей результат можна за допомогою рядка

```
gcc –E –P d:\KIT220\main.c –o d:\KIT220\test.txt,
```

в якому необхідно задати повний шлях до файлу з програмою (**`main.c`**), а також шлях до файлу з результатом (**`test.txt`**). З рис. 7.1 видно, що текст може бути будь–яким (препроцесор буде обробляти лише директиви, що починаються з символу `'#'`). Опції для командного рядка компілятора **`gcc`** наведені в додатку 7.

В даному випадку в якості операторних фігурних дужок застосовуються ідентифікатори **`begin`** і **`end`**. Відповідні вказівки програміст дав препроцесору

за допомогою двох директив `#define`. До компіляції препроцесор замінює усі входження цих ідентифікаторів стандартними дужками `'{'` і `'}'`.

7.4. Ланцюг підстановок

Якщо в рядку заміщення директиви `#define` в якості окремої лексеми зустрічається препроцесорний ідентифікатор, який раніше був визначений іншою директивою `#define`, то виконується ланцюг послідовних підстановок. В якості прикладу розглянемо, як можна визначити діапазон `RANGE` можливих значень будь–якої цілої змінної типу **`int`** в наступній програмі:

```
#include <limits.h>
#define RANGE ((INT_MAX) - (INT_MIN) + 1)
// RANGE - діапазон значень для int
int RANGE_T = RANGE / 8;
```

Препроцесор послідовно переглядає текст і, виявивши директиву

```
#include <limits.h>
```

вставляє текст з файлу **`limits.h`**, фрагмент якого наведено на рис. 7.2.

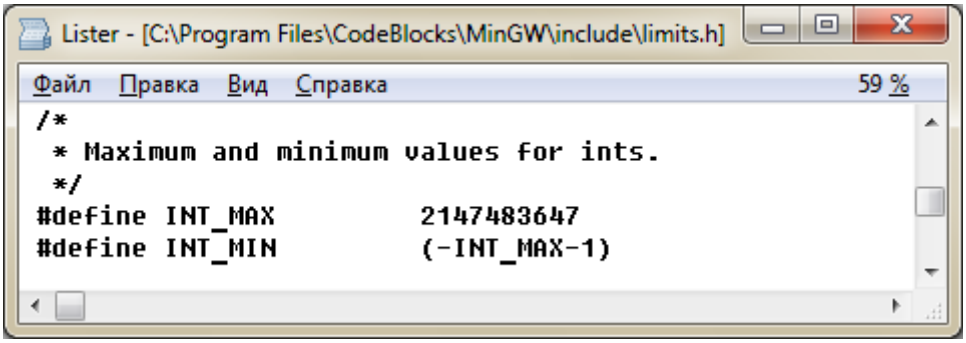


Рисунок 7.2 – Фрагмент файлу `limits.h`

Там визначені константи `INT_MAX` і `INT_MIN` (максимальне та мінімальне значення цілих величин). Програма буде мати такий вигляд (інші рядки файлу **`limits.h`** не будемо розглядати):

```
#define INT_MAX 2147483647
#define INT_MIN (-INT_MAX-1)
#define RANGE ((INT_MAX) - (INT_MIN) + 1)
int RANGE_T = RANGE / 8;
```

Зверніть увагу, що директива `#include` зникла з програми, але її замінив відповідний текст. Зник також і коментар. Виявивши в тексті файлу **`limits.h`** дві директиви `#define`, препроцесор робить відповідні підстановки, і програма приймає такий вигляд:

```
#define RANGE ((2147483647) - ((-2147483647-1)) + 1)
int RANGE_T = RANGE / 8;
```

Підстановки змінили рядок заміщення препроцесорного ідентифікатора `RANGE` в директиві `#define`. Послідовно аналізуючи текст програми, препроцесор зустрічає препроцесорний ідентифікатор `RANGE` і виконує підстановку. Програма набуває наступного вигляду:

```
int RANGE_T = ((2147483647) - ((-2147483647-1)) + 1) / 8;
```

Тепер усі директиви `#define` видалені з тексту. В результаті отримано текст для подальшої компіляції, тобто створена «одиниця трансляції». Підстановка рядка заміщення замість ідентифікатора `RANGE` виконана у виразі `RANGE/8`.

Цей приклад ілюструє виконання «ланцюга» підстановок. Фрагмент реального файлу на вході та виході препроцесора наведено на рис. 7.3: зверху – текст *С*–програми до препроцесора; знизу – після препроцесора.

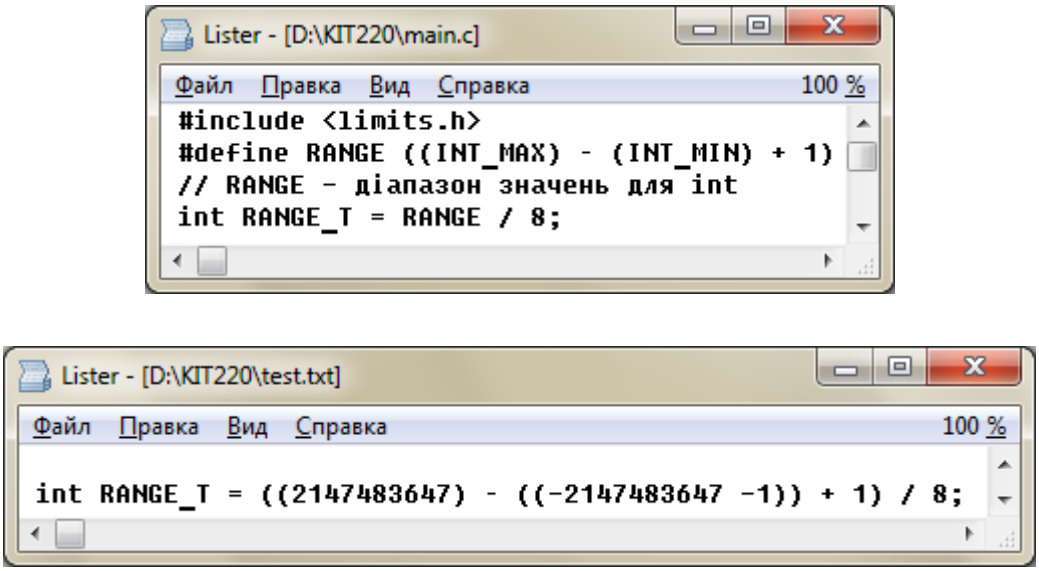


Рисунок 7.3 – Демонстрація результатів роботи препроцесора

Якщо рядок заміщення виявиться надто довгим, то його можна продовжити на наступному рядку тексту. Для цього в кінці рядка поміщається символ `'\'`. В ході однієї зі стадій препроцесорної обробки цей символ разом з наступним символом кінця рядка буде видалений з тексту програми.

7.5. Використання аргументів у директиві `#define`

Розглянемо приклад використання директиви `#define` для створення функціонального макросу. Текст програми для отримання синуса кута, що заданий в градусах, має такий вигляд:

```
#include <stdio.h>
#include <windows.h>
#include <math.h>
#define PI 3.14159265
#define SIN(x) sin(PI * x / 180)
int main(void)
{
    int c;
    SetConsoleOutputCP(1251);
    printf("Введіть кут в градусах: ");
    scanf("%d", &c);
    printf("============================\n");
    printf("sin(%d) = %lf\n", c, SIN(c));
    printf("============================\n\n");
    return 0;
}
```

Результат роботи програми наведено на рис. 7.4.

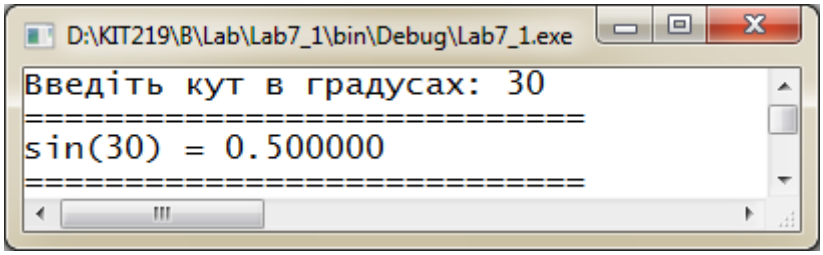


Рисунок 7.4 – Результат роботи програми з директивою препроцесора `#define`, яка використовувалася для створення функціонального макросу

7.6. Застосування операції

Операція ## може застосовуватися в замінній частині функціонального макросу. Додатково вона може використовуватися в замінній частині об'єктного макросу. Операція ## об'єднує дві лексеми в одну. Текст програми, що демонструє застосування операції ## має такий вигляд:

```
#include <stdio.h>
#define SUM(x,y) (a##x + a##y)

int main(void)
{
    int a1 = 5, a2 = 3;
    int a3 = 4, a4 = 7;

    printf("a1 + a2 = %2d\n", SUM(1, 2)); // (a1 + a2)
    printf("a3 + a4 = %2d\n", SUM(3, 4)); // (a3 + a4)
    printf("\n");
    return 0;
}
```

Результат роботи програми наведено на рис. 7.5.

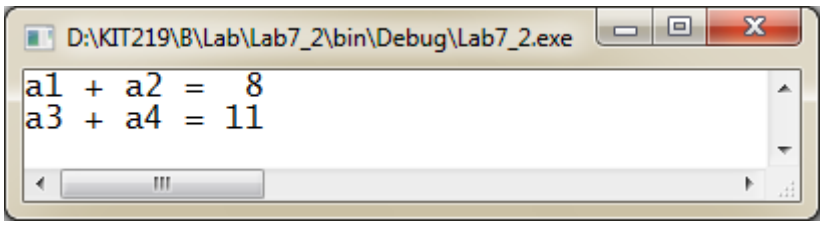


Рисунок 7.5 – Результат застосування директиви препроцесора #define та операції злиття ##

7.7. Умовна компіляція

Директиви умовної компіляції препроцесора наведені на рис. 7.6.

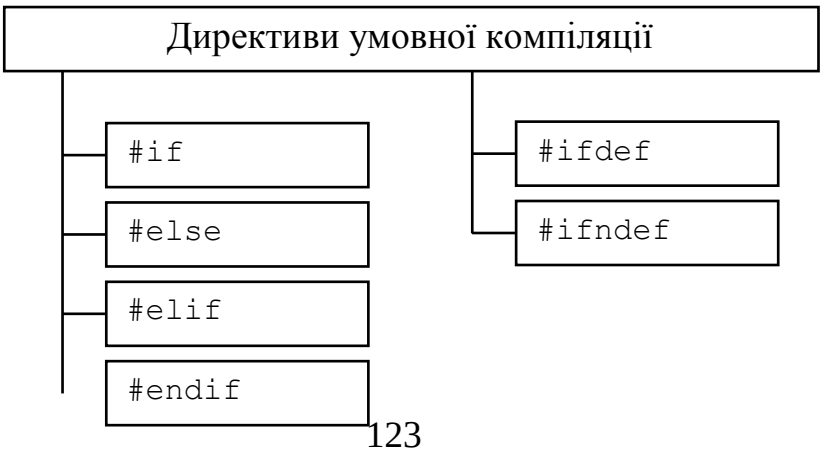


Рисунок 7.6 – Директиви препроцесора для умовної компіляції

7.7.1. Застосування директив `#if`, `#else`, `#elif` і `#endif`

Якщо після директиви `#if` константний вираз приймає істинне значення, то код між `#if` і `#endif` компілюється, в протилежному випадку – пропускається. Директива `#endif` використовується для позначення кінця блоку `#if`.

Стандартний вигляд директиви `#if` такий :

```
#if константний_вираз
    послідовність операторів
#endif
```

Розглянемо приклад використання директиви `#if`:

```
#include <stdio.h>
#include <windows.h>
#define MAX 100

int main(void)
{
    SetConsoleOutputCP(1251);
    #if MAX > 99
        printf("Компіляція проводиться лише для MAX > 99\n");
    #endif
    return 0;
}
```

Результат роботи програми наведено на рис. 7.7.

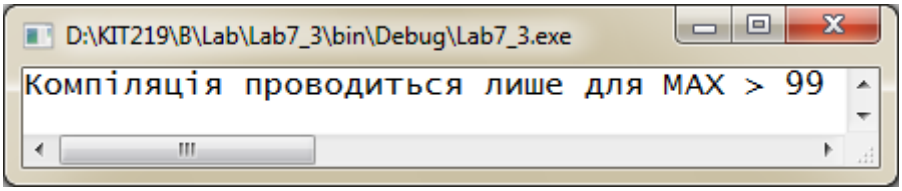


Рисунок 7.7 – Результат застосування директиви `#if`

Дана програма виводить повідомлення на екран, оскільки `MAX` має значення більше ніж 99. Даний приклад демонструє дуже важливий момент: вираз, що йде за директивою `#if`, обчислюється на етапі компіляції, відповідно, він повинен містити наперед визначені ідентифікатори і константи, а не змінні.

Директива `#else` застосовується лише у складі директиви `#if...#else`. Повний вигляд цієї директиви такий :

```
#if константний_вираз
    послідовність операторів_1
#else
    послідовність операторів_2
#endif
```

Робота директиви `#else` дуже схожа на роботу оператора **`else`** – вона надає альтернативний варіант компіляції, якщо `#if` містить хибне значення виразу. Попередній приклад можна модифікувати наступним чином:

```
#include <stdio.h>
#include <windows.h>
#define MAX 10

int main(void)
{
    SetConsoleOutputCP(1251);
    #if MAX > 99
        printf("Компіляція проводиться лише для MAX > 99\n");
    #else
        printf("Компіляція для значень MAX <= 99\n");
    #endif
    return 0;
}
```

Результат роботи програми наведено на рис. 7.8.

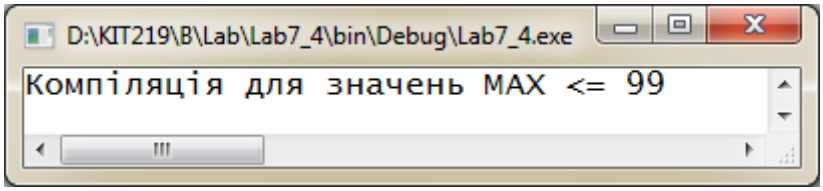


Рисунок 7.8 – Результат застосування директиви `#else`

В даному випадку `MAX` визначається таким чином, щоб значення було меншим ніж 99. В результаті буде компілюватися код, що йде після директиви `#else`. Зверніть увагу, що `#else` використовується для позначення кінця блоку `#if` і початку блоку `#else`. Це робиться тому, що може бути лише одна директива `#endif`, яка пов’язана з `#if`.

Директива `#elif` означає «інакше, якщо» і використовується для побудови конструкції **`if-else-if`** для визначення різних опцій компіляції. За директивою `#elif` повинен йти константний вираз. Якщо він істинний, то блок коду компілюється, решта виразів не перевіряються. В протилежному випадку розглядається наступний блок. Стандартний вигляд `#elif` такий :

```
#if вираз
    послідовність операторів
#elif вираз_1
    послідовність операторів_1
#elif вираз_2
    послідовність операторів_2
#elif вираз_3
    послідовність операторів_3
#elif вираз_4
    послідовність операторів_4
...
#elif вираз_N
    послідовність операторів_N
#endif
```

Наприклад, такий фрагмент використовує значення `ACTIVE_COUNTRY` для визначення валюти:

```
#include <stdio.h>
#include <windows.h>
#define USA 0
#define FRA 1
#define UKR 2
#define ACTIVE_COUNTRY FRA

int main(void)
{
    SetConsoleOutputCP(1251);
    #if ACTIVE_COUNTRY == USA
        char currency[] = "долар";
    #elif ACTIVE_COUNTRY == FRA
        char currency[] = "євро";
    #else
        char currency[] = "гривня";
    #endif
    printf("Валюта країни: %s\n", currency);
    return 0;
}
```

Результат роботи програми наведено на рис. 7.9.

Директиви `#if` і `#elif` можуть бути вкладеними. Тоді кожен `#endif`, `#else` або `#elif` асоціюються з найближчим `#if` або `#elif`. Наприклад:

```
#if MAX > 100
    #if SERIAL_VERSION
        int port = 198;
    #elif
        int port = 200;
    #endif
#else
    char out_buffer[100];
#endif
```

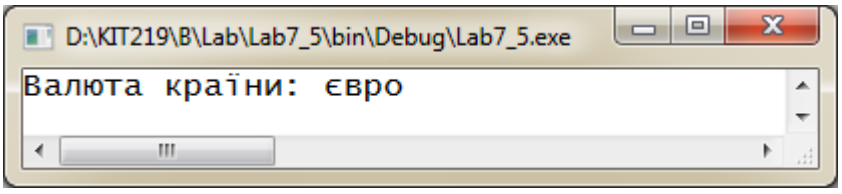


Рисунок 7.9 – Результат застосування директиви `#elif`

7.7.2. Застосування директив `#ifdef` і `#ifndef`

Другий метод умовної компіляції полягає у використанні директив `#ifdef` і `#ifndef`, що відповідно означає «якщо визначено» та «якщо не визначено». Стандартний вигляд директиви `#ifdef`:

```
#ifdef ім'я_макросу
    послідовність операторів
#endif
```

Якщо `ім'я_макросу` визначено раніше в операторі `#define`, то буде компілюватися послідовність операторів, що знаходяться між директивами `#ifdef` і `#endif`. Стандартний вигляд директиви `#ifndef`:

```
#ifndef ім'я_макросу
    послідовність операторів
#endif
```

Якщо `ім'я_макросу` не визначено раніше в операторі `#define`, то буде компілюватися послідовність операторів, що знаходяться між `#ifdef` і `#endif`.

Як `#ifdef`, так і `#ifndef` можуть використовувати директиву `#else`, але не директиву `#elif`.

Наприклад:

```
#include <stdio.h>
#include <windows.h>
#define TED 10

int main(void)
{
    SetConsoleOutputCP(1251);
    #ifdef TED
        printf("Константа TED визначена\n");
    #else
        printf("Константа TED не визначена\n");
    #endif
    #ifndef RALPH
        printf("Константа RALPH не визначена\n");
    #endif
    return 0;
}
```

Результат роботи програми наведено на рис. 7.10.

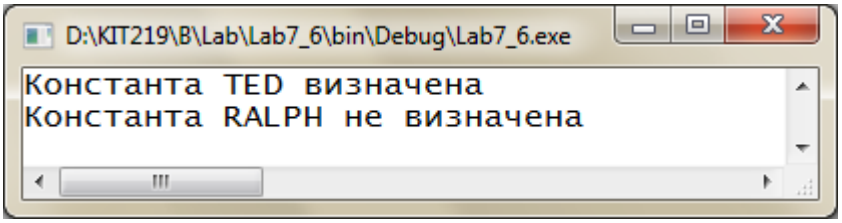


Рисунок 7.10 – Результат застосування директив `#ifdef` та `#ifndef`

Директиви `#ifdef` і `#ifndef` можна також вкладати одна в одну.

7.7.3. Застосування `defined`

Окрім директиви `#ifdef` існує інший спосіб дізнатися, чи визначений макрос, чи ні. Разом з директивою `#if` можна використовувати оператор часу компіляції `defined`, який має такий вигляд:

```
defined ім'я_макросу
```

Якщо `ім'я_макросу` визначено, то вираз стає істинним. У протилежному випадку він стає хибним. Наприклад, для того, щоб дізнатися, чи визначений макрос `MYFILE`, можна скористатися однією з двох команд препроцесора:

```
#if defined MYFILE
```

або

```
#ifdef MYFILE
```

Попереду `defined` можна також помістити символ `'!'` для зміни поточного стану. Наприклад, такий фрагмент компілюється, тільки якщо `DEBUG` не визначений.

```
#if !defined DEBUG
    printf("Final version!\n");
#endif
```

7.8. Застосування директиви `#undef`

Директива `#undef` використовується для зняття визначення макросу. Вона має такий вигляд:

```
#undef ім'я_макросу
```

Приклад використання директиви `#undef`:

```
#define LEN 100
#define WIDTH 100
char array[LEN][WIDTH];
#undef LEN
#undef WIDTH
// на даний момент як LEN, так і WIDTH – не визначені
```

Основне призначення директиви `#undef` полягає в локалізації імен макросів тільки в тих частинах, де вони необхідні.

7.9. Застосування директиви `#error`

Директива `#error` вказує компілятору на припинення компіляції у разі виникнення помилки. Як правило, вона використовується для проведення налагодження програми. Загальний вигляд директиви такий :

```
#error повідомлення_про_помилку
```

Зверніть увагу, що `повідомлення_про_помилку` записується без подвійних лапок. Коли компілятор виявляє цю директиву, він виводить повідомлення в наступному вигляді і завершує компіляцію.

```
Fatal: ім'я_файлу номер_рядка: Error directive: повідомлення_
про_помилку
```

Тут `ім'я_файлу` – це ім'я файлу, де була знайдена директива **`#error`**, `номер_рядка` – це номер рядка директиви, а `повідомлення_про_помилку` – це власне саме повідомлення.

7.10. Наперед визначені макроси

В стандарті *C* є декілька наперед визначених макросів: `__DATE__`, `__FILE__`, `__LINE__`, `__STDC__`, `__STDC_HOSTED__`, `__STDC_VERSION__`, `__TIME__`, які більш докладно розглянуті в лекції.

Хід виконання роботи:

1. Опрацюйте матеріал лекції «Препроцесор мови *C*», а також теоретичний матеріал даної лабораторної роботи.

2. Напишіть *C*–програму, яка демонструє можливості використання аргументів у директиві `#define` при створенні функціональних макросів, можливості створення рядків з аргументів макросу за допомогою операції `#`, а також варіанти використання операції `##`.

3. Напишіть *C*–програму, яка демонструє можливості роботи з директивами препроцесора `#if`, `#else`, `#elif` і `#endif` , а також з директивами `#ifdef` і `#ifndef` для забезпечення умовної компіляції.

4. Напишіть власну *C*–програму, яка демонструє можливість застосування наперед визначених макросів: `__DATE__`, `__FILE__`, `__LINE__`, `__STDC__`, `__STDC_HOSTED__`, `__STDC_VERSION__`, `__TIME__`, а також наперед визначеного ідентифікатора `__func__`.

В кожній з трьох програм будь–яким чином необхідно задіяти ваше прізвище, ім'я, групу (або в якості даних, або як аргумент функції `printf()`).

Звіт про виконання лабораторної роботи повинен містити:

1. Титульний аркуш.
2. Мету роботи.
3. Результати виконання роботи:
   – тексти *C*–програм;
   – результати виконання *C*–програм.
4. Висновки по роботі.

Контрольні запитання:

1. Для чого використовується препроцесор, та які функції він виконує?
2. Разом з якою директивою препроцесора застосовуються операції `##` і `#`. Що вони означають?
3. Які існують варіанти застосування директиви `#define`? Для чого вона призначена?
4. Яким чином використовують директиву `#include`? Які способи застосування цієї директиви ви знаєте?
5. Які директиви забезпечують умовну компіляцію?
6. Які наперед визначені макроси вам відомі?
7. Що дозволяє отримати наперед визначений ідентифікатор `__func__`?
8. Що препроцесор робить з коментарями?
9. Що треба робити у випадку, коли директива препроцесора `#define` не вміщується в один рядок?

Лабораторна робота №8

ДОСЛІДЖЕННЯ ОСНОВНИХ МОЖЛИВОСТЕЙ РОБОТИ З ГРАФІЧНОЮ БІБЛІОТЕКОЮ WINBGIM В CODE::BLOCKS

Мета роботи: адаптувати графічну бібліотеку WinBGIm до середовища *Code::Blocks*; ознайомитися з основними функціями графічної бібліотеки WinBGIm; навчитися застосовувати основні графічні функції в *C*–програмах для візуалізації інформації на екрані комп'ютера.

Теоретична частина

WinBGIm – це графічна бібліотека BGI (Borland Graphics Interface), яка була адаптована для середовища *Code::Blocks* Майклом Мейном (Michael Main) почесним професором Університету Колорадо (США). Оскільки бібліотека несумісна з новими версіями *Code::Blocks*, рекомендується завантажити портативну версію (рис. 8.1) з сайту `http://codeblocks.codecutter.org/`.

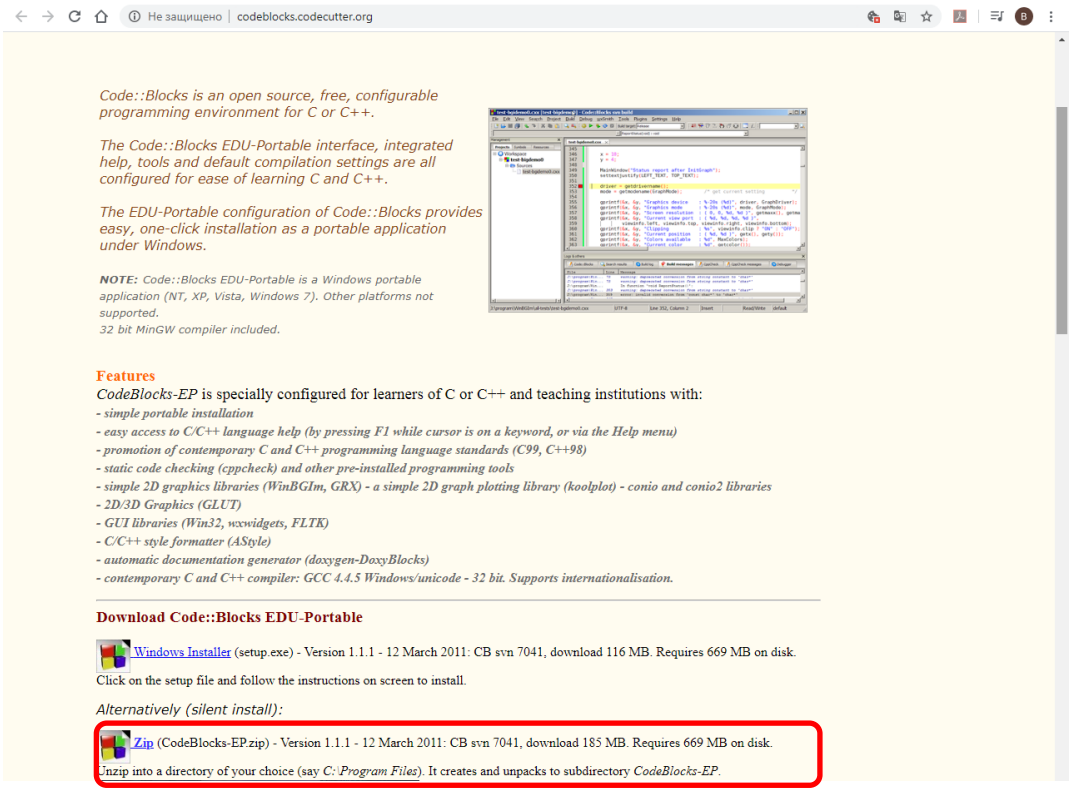


Рисунок 8.1 – Місце розташування файлу `CodeBlocks-EP.zip`

Для того, щоб підключити її до IDE *Code::Blocks*, необхідно виконати наступні дії:

1) Скопіювати файл **`CodeBlocks-EP.zip`** в окрему директорію **`D:\CodeBlocks-EP\`** та розпакувати архів (рис. 8.2).

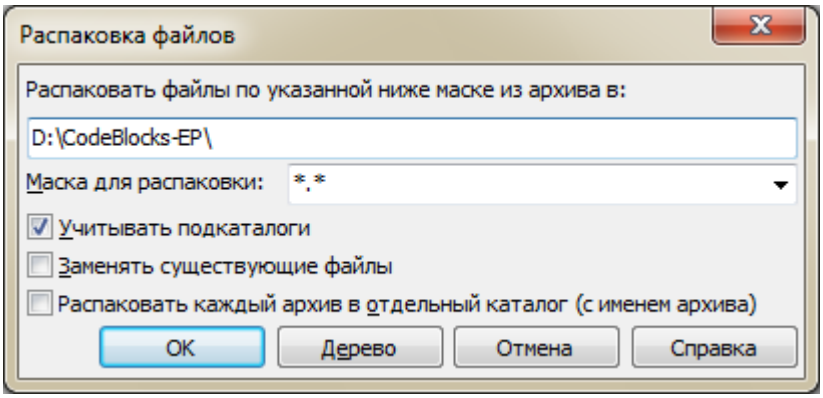


Рисунок 8.2 – Вказуємо директорію для розпаковування архіву

Список файлів портативної версії *Code::Blocks* наведено на рис. 8.3.

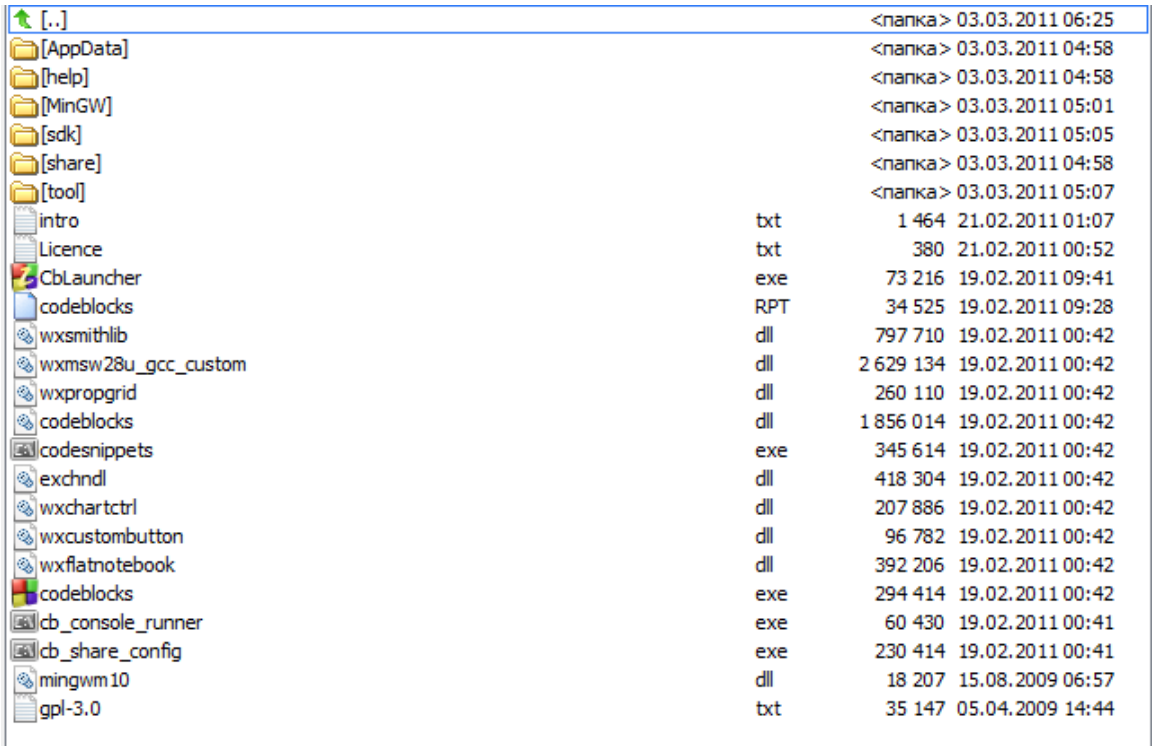


Рисунок 8.3 – Список файлів портативної версії *Code::Blocks*

2) Завантажити за адресою https://www.dropbox.com/s/nnwjx4eb0ioc4r9/graphics.h файл **graphics.h** (рис. 8.4) і за адресою https://www.dropbox.com/s/ihz8tf6dd8hh1pr/winbgim.h файл **winbgim.h** (рис. 8.5) та помістити їх до директорії компілятора **..\CodeBlocks-EP\MinGW\include**.

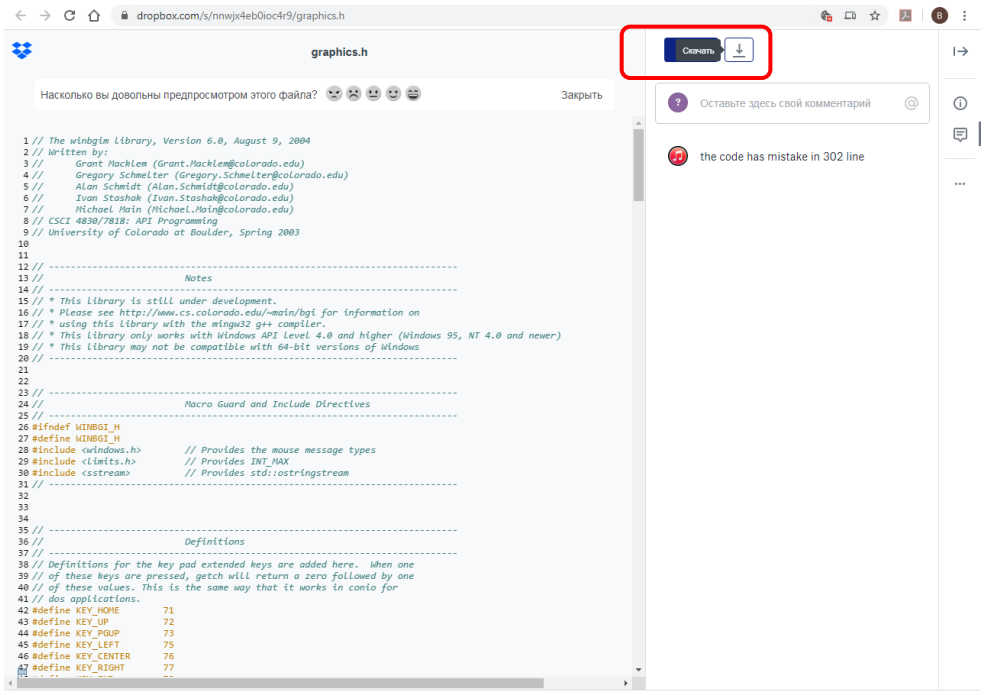


Рисунок 8.4 – Вікно для завантаження файлу **graphics.h**

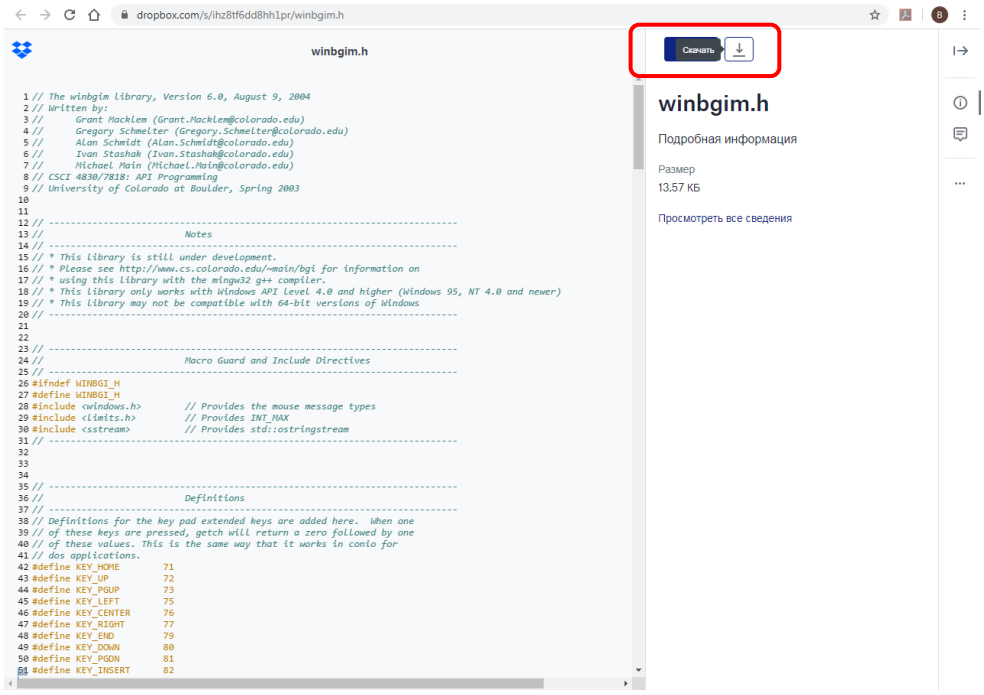


Рисунок 8.5 – Вікно для завантаження файлу `winbgim.h`

3) Завантажити за адресою https://www.dropbox.com/s/t4amk8yqnd7wjb0/libbgi.a файл **`libbgi.a`** (рис. 8.6) та помістити його до директорії **`..\CodeBlocks-EP\MinGW\lib`**.

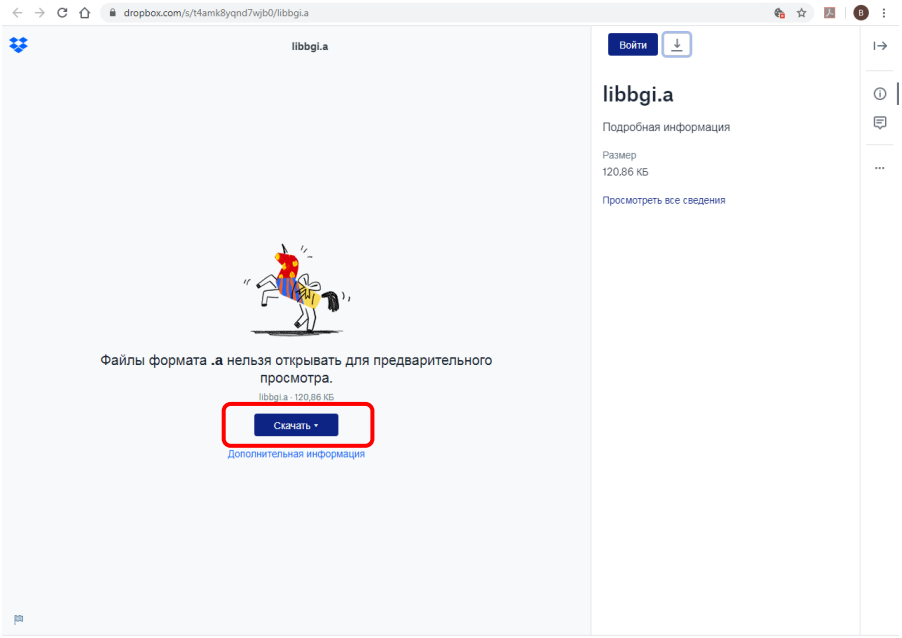


Рисунок 8.6 – Вікно для завантаження файлу `libbgi.a`

4) Запустити програму **`CbLancher.exe`** (!!!!!) та відкрити **`Settings`** → **`Compiler and debugger...`** → **`Linker settings`** (рис. 8.7).

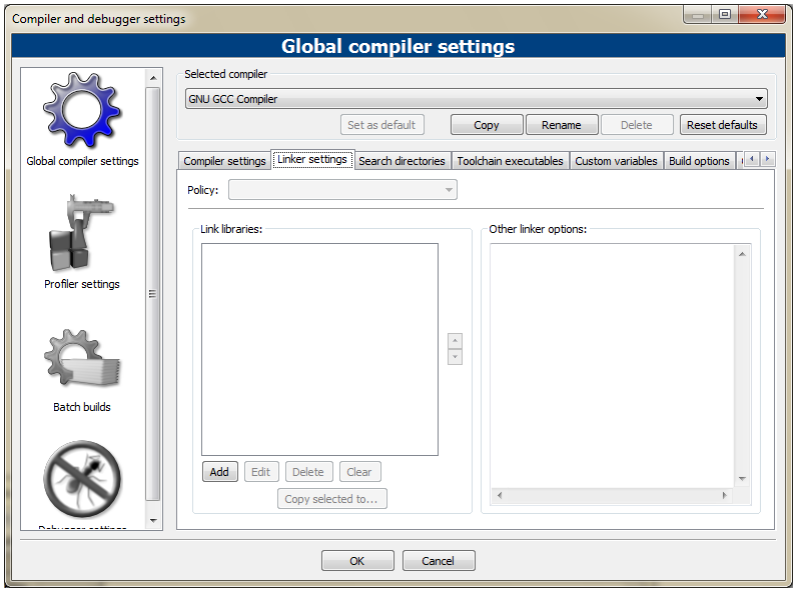


Рисунок 8.7 – Глобальні параметри компілятора (Linker settings)

5) Натиснути на кнопку «Add» та завантажити файл **`libbgi.a`** (рис. 8.8).

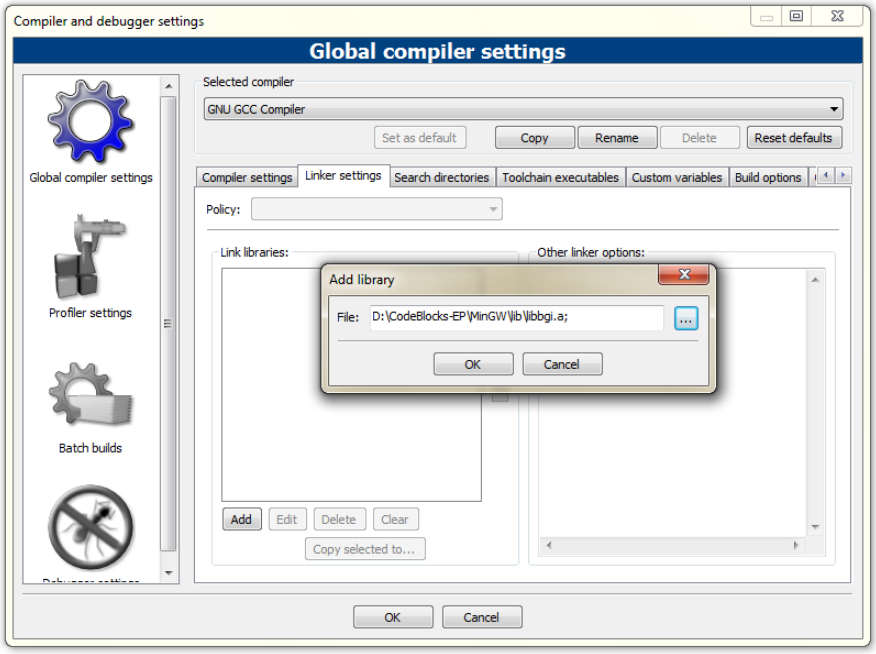


Рисунок 8.8 – Вибір файлу `libbgi.a` після натиснення кнопки «Add»

6) В правій частині (Other linker options:) додати команди:

```
-lbgi -lgdi32 -lcomdlg32 -luuid -loleaut32 -lole32
```

та натиснути кнопку «ОК» (рис. 8.9).

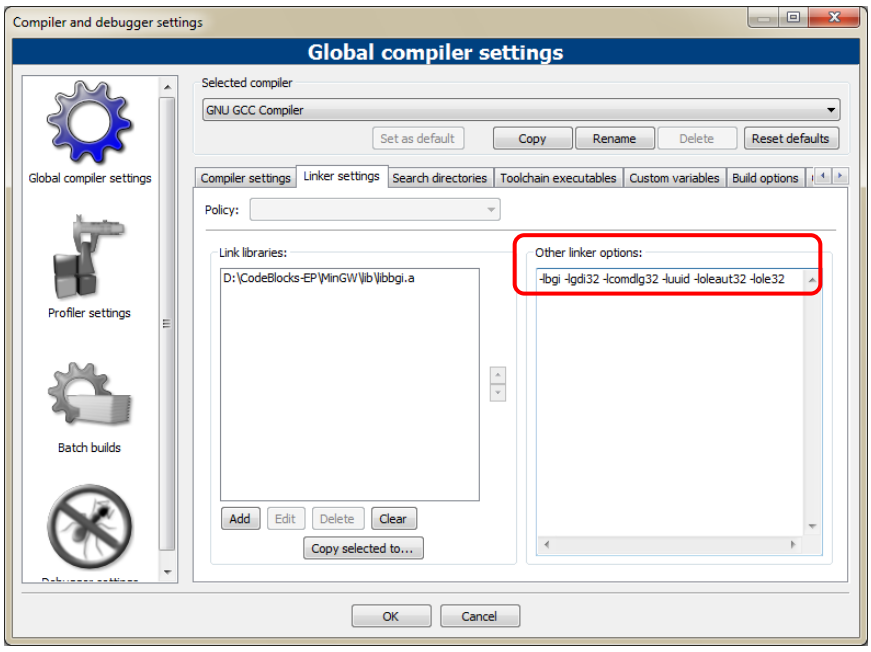


Рисунок 8.9 – Додавання команд до вікна «Other linker options:»

Крім того, необхідно встановити ще деякі глобальні параметри компілятора, які вказані на рис. 8.10 і рис. 8.11.

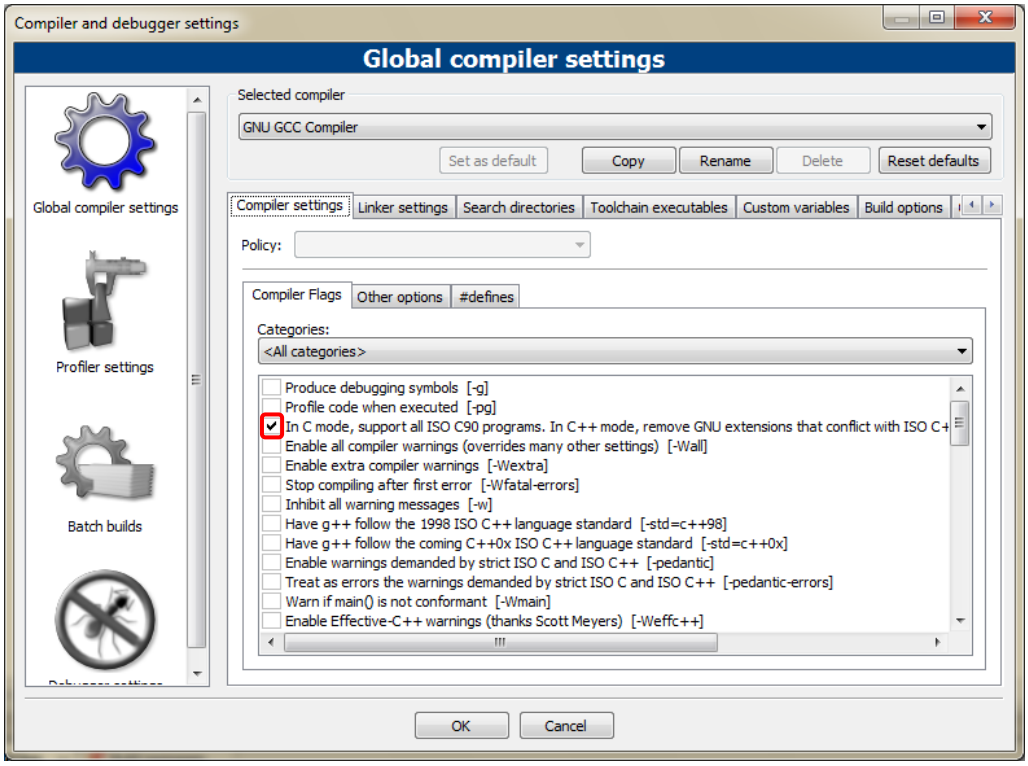


Рисунок 8.10 – Ставимо галочку у вкладці «Compiler settings»

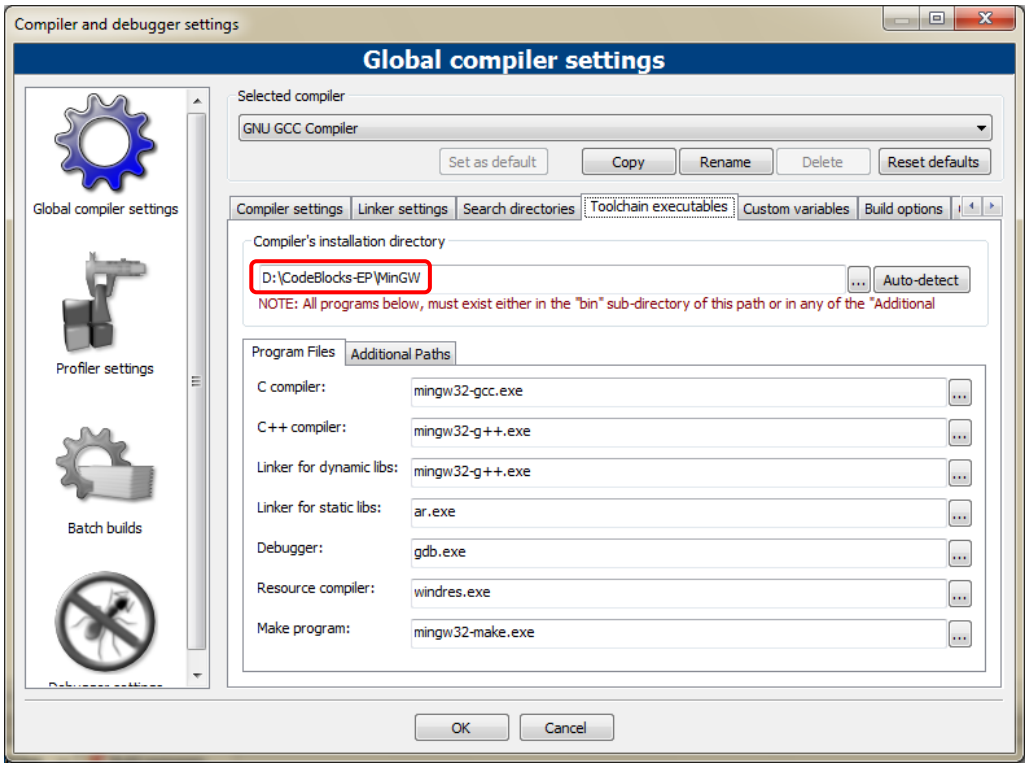


Рисунок 8.11 – Встановлюємо шлях до компілятора

Тепер можна створити проєкт **`WinBGIm project`** та написати програму для демонстрації графіки. Для цього необхідно зробити декілька кроків зі створення проєкту (рис. 8.12 – 8.16).

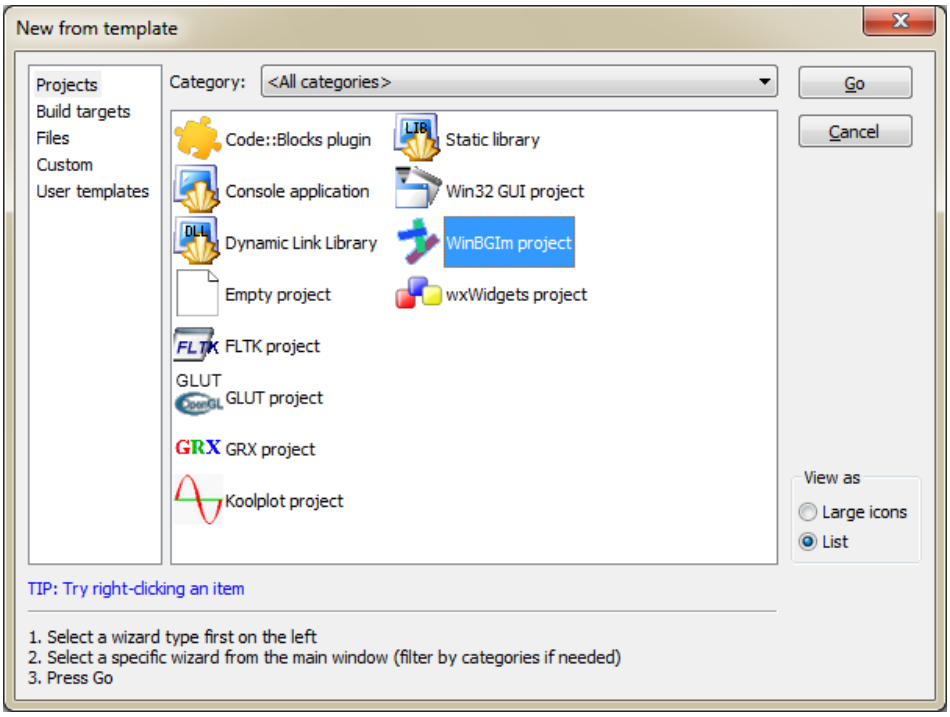


Рисунок 8.12 – Обираємо створення проєкту WinBGIm project

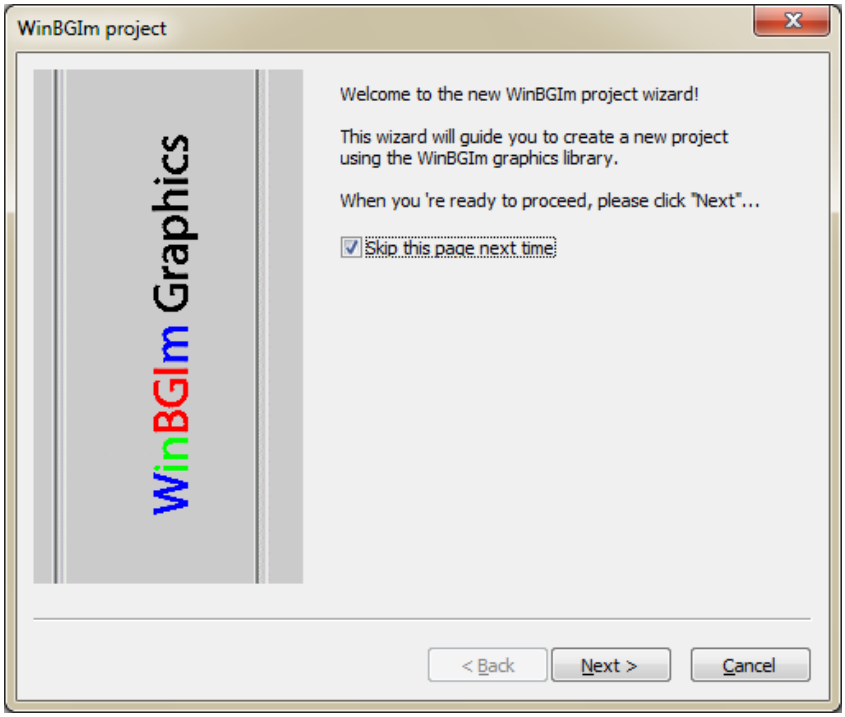


Рисунок 8.13 – Ставимо галочку та натискаємо кнопку «Next»

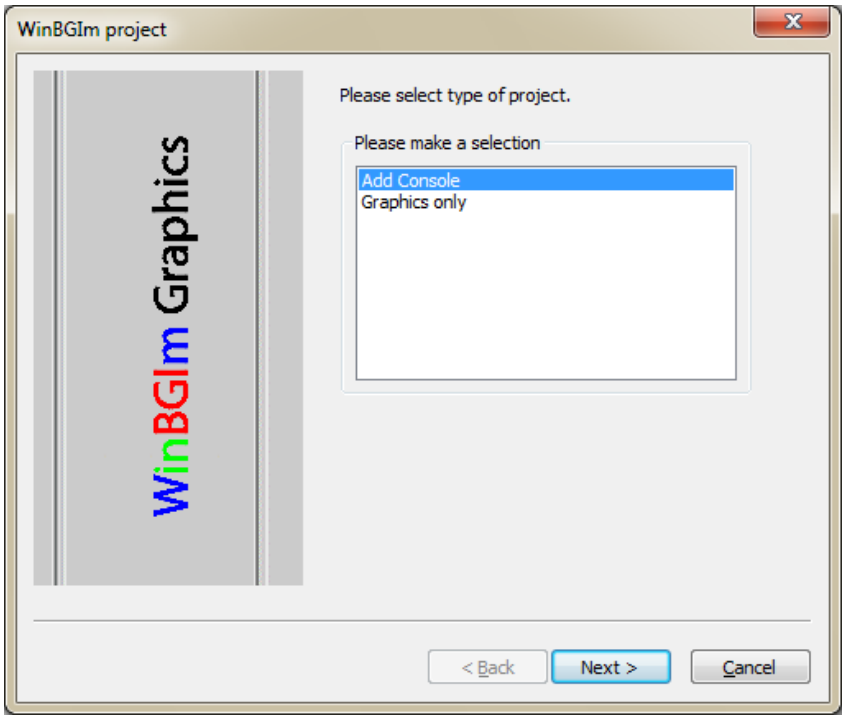


Рисунок 8.14 – Обираємо «Add Console» та натискаємо кнопку «Next»

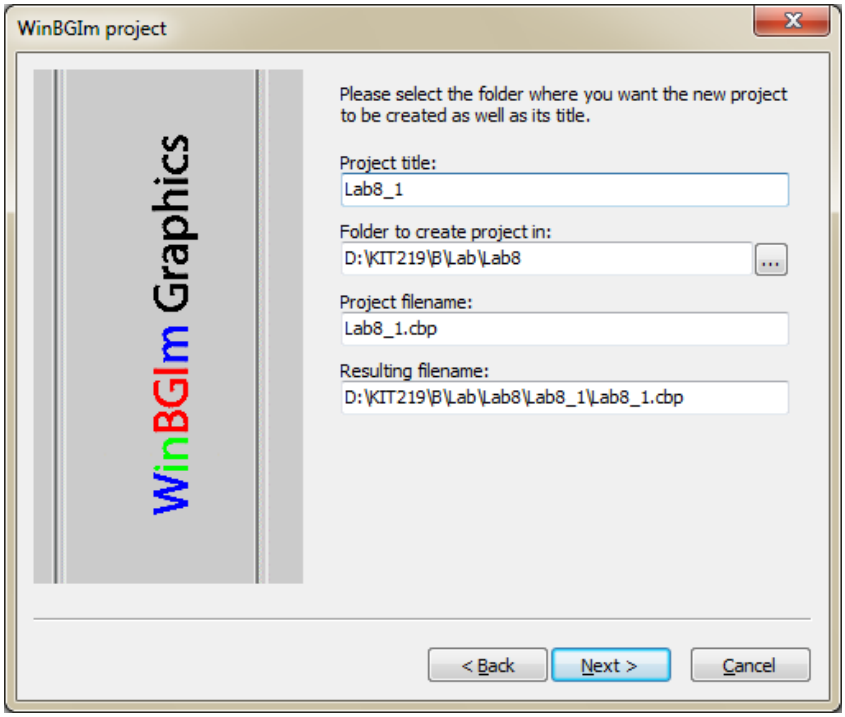


Рисунок 8.15 – Обираємо місце розташування проєкту та його ім’я

З метою створення файлу з програмою для демонстрації графіки необхідно виконати декілька кроків (рис. 8.17 – 8.20).

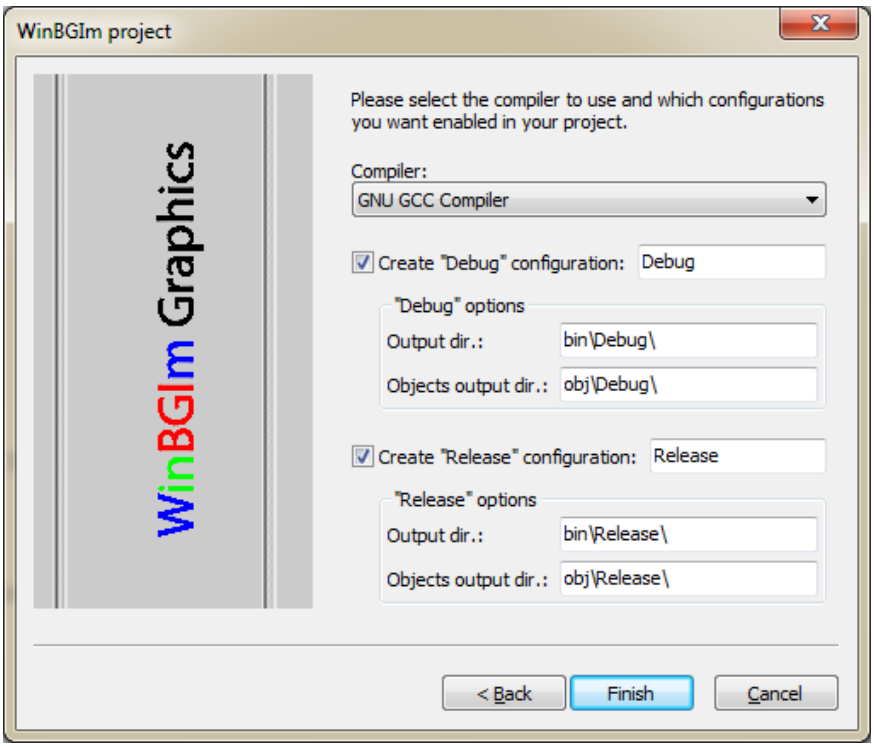


Рисунок 8.16 – Натискаємо кнопку «Finish»

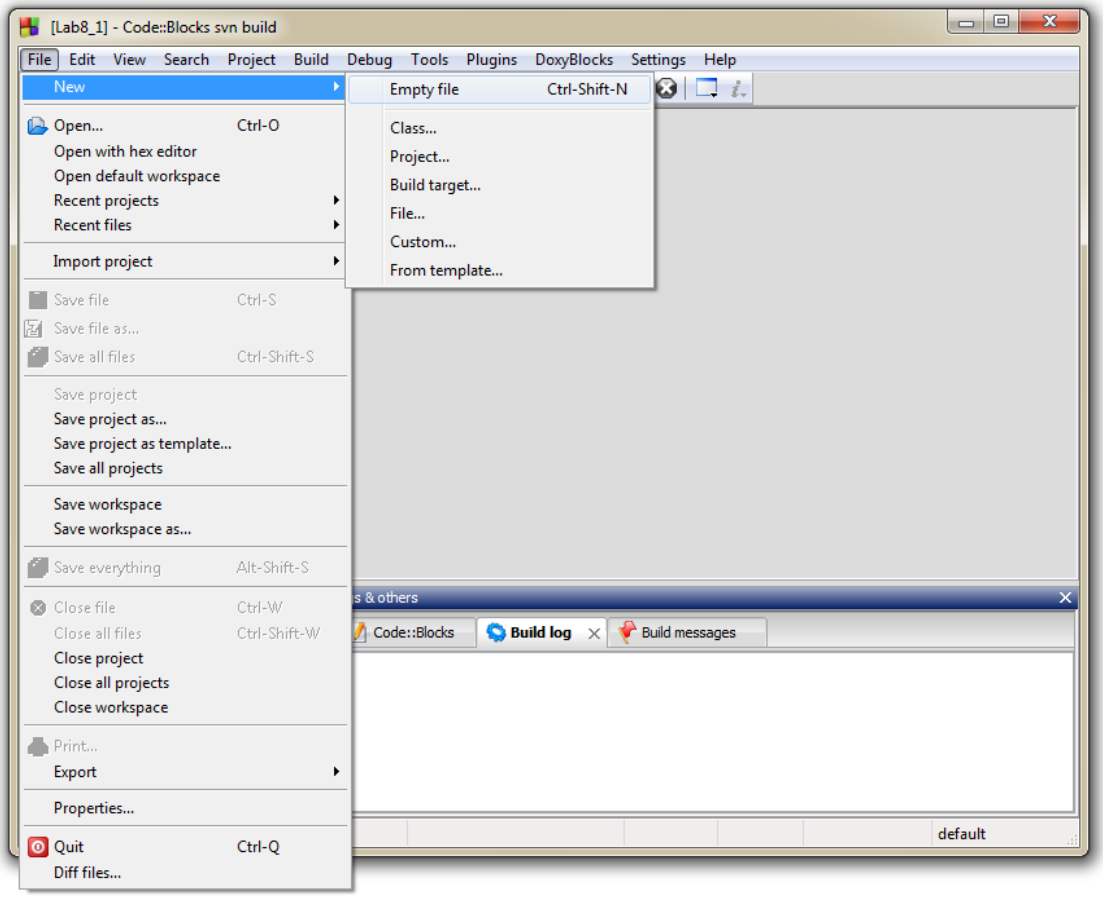


Рисунок 8.17 – Обираємо «Empty File» для створення нового файлу

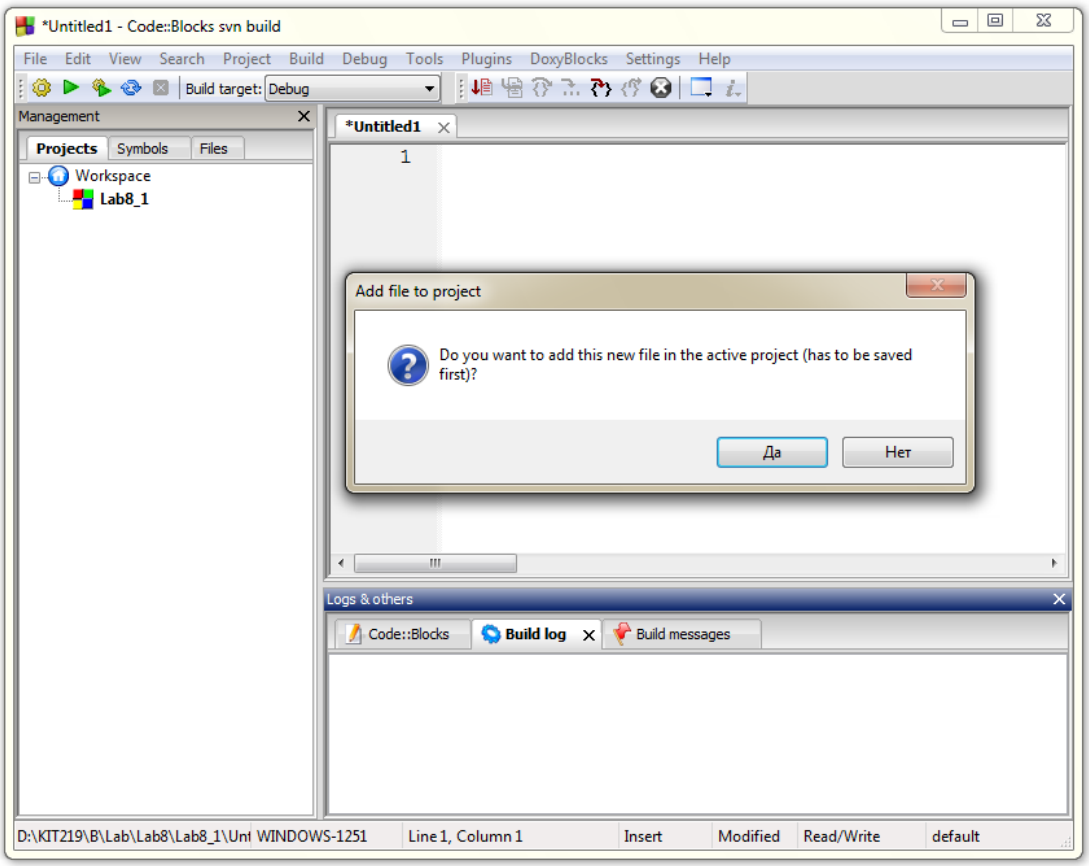


Рисунок 8.18 – Натискаємо кнопку «Так» для додавання файлу до проєкту

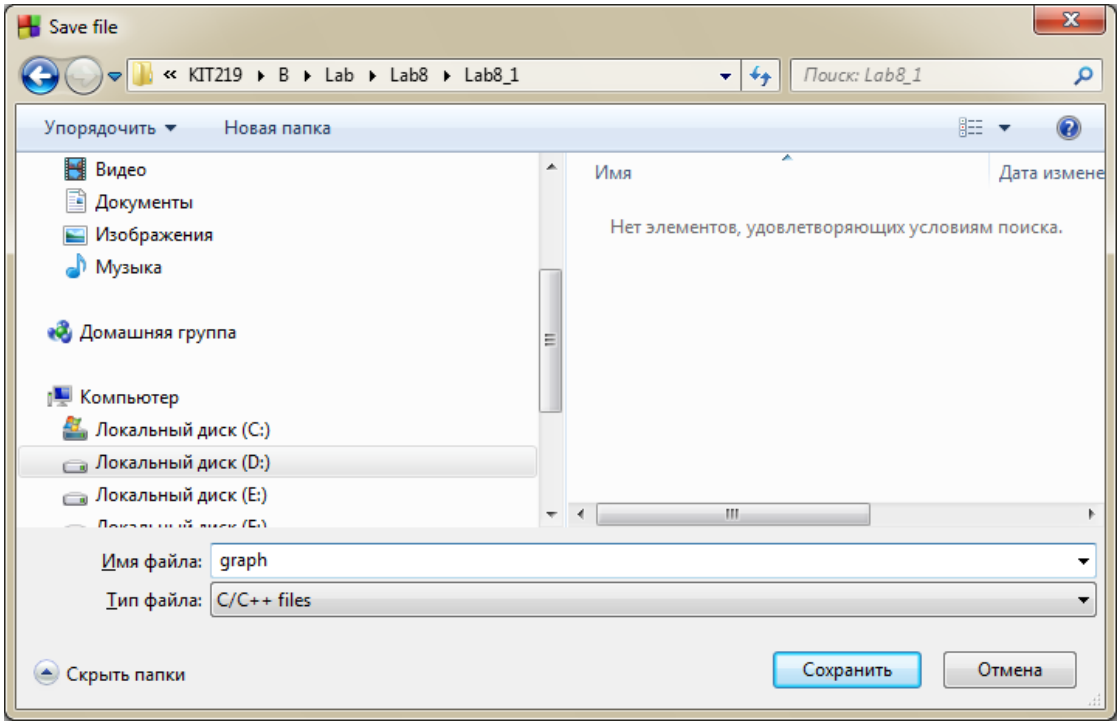


Рисунок 8.19 – Задаємо ім’я для нового файлу

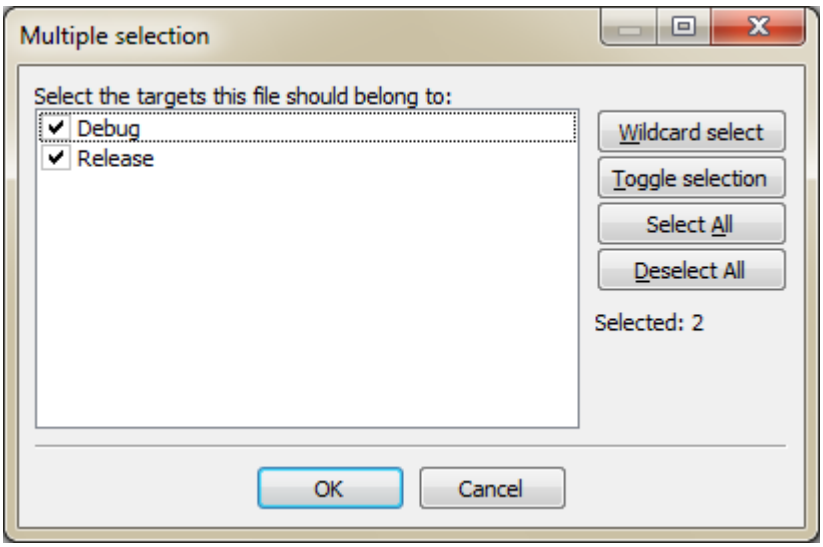


Рисунок 8.20 – Натискаємо кнопку «OK» для завершення

В результаті отримаємо вікно для формування тексту *C*–програми для роботи з графікою (рис. 8.21).

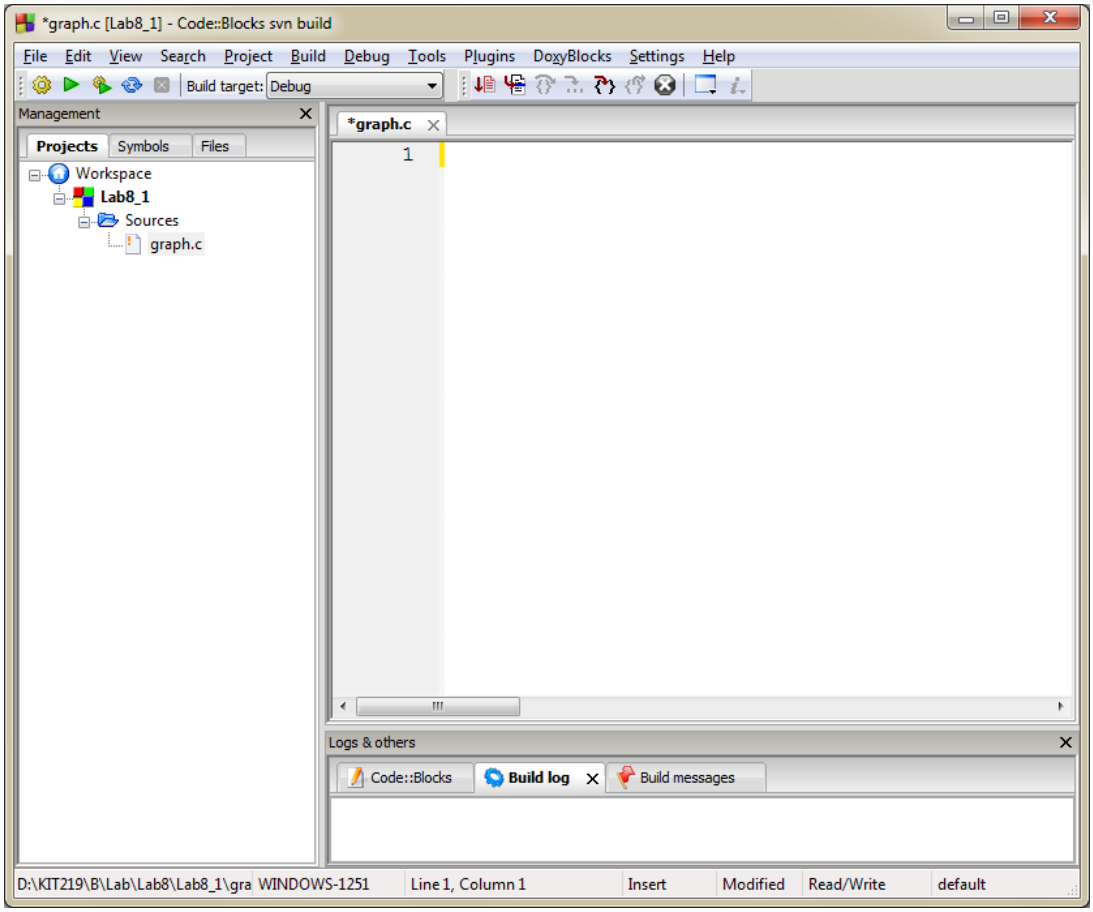


Рисунок 8.21 – Вікно середовища для створення *C*–програми

Текст програми для демонстрації графіки має такий вигляд:

```
#include <graphics.h>
#include <conio.h>
int main(void)
{
    int gdriver = DETECT, gmode;
    initgraph(&gdriver, &gmode, " ");
    setbkcolor(WHITE);
    cleardevice();
    setfillstyle(SOLID_FILL, LIGHTGRAY);
    setcolor(LIGHTGRAY);
    rectangle(50, 50, 50+400, 50+300);
    floodfill(51, 51, LIGHTGRAY);
    setfillstyle(SOLID_FILL, WHITE);
    setcolor(WHITE);
    circle(250, 200, 100);
    floodfill(240, 200, WHITE);
    getch();
    closegraph(ALL_WINDOWS);
    return 0;
}
```

Результат виконання графічної програми наведено на рис. 8.22.

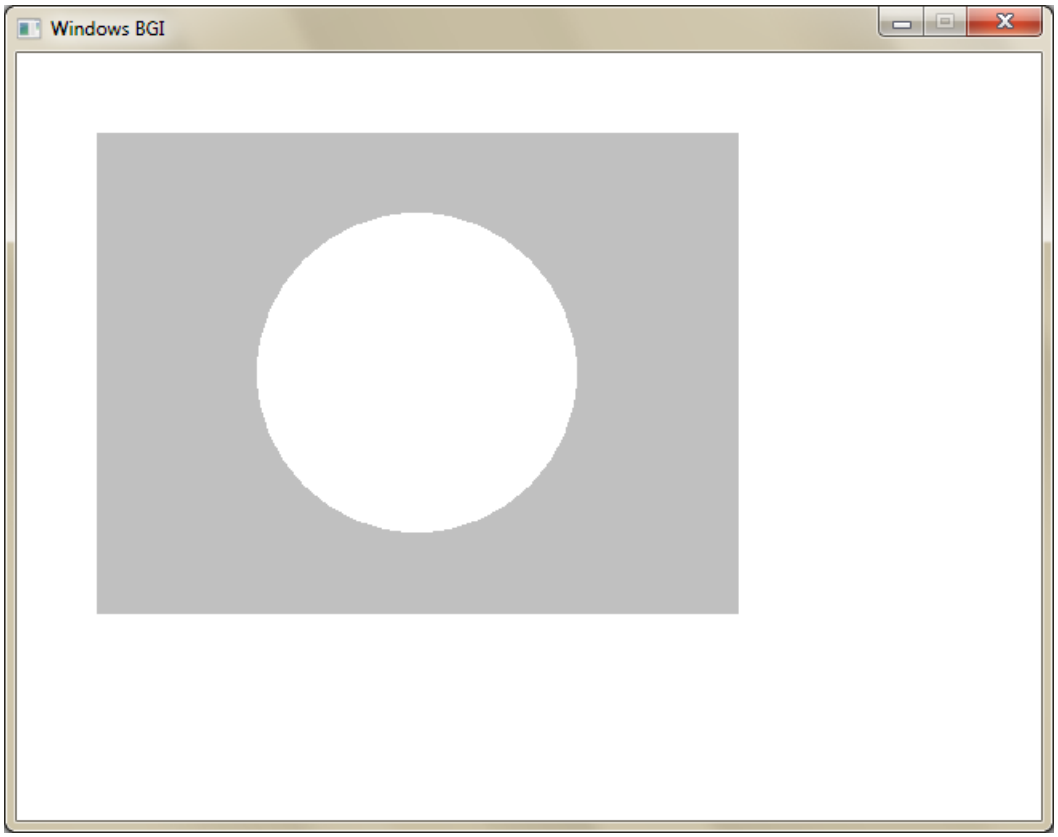


Рисунок 8.22 – Демонстрація можливостей роботи з графікою

Розглянемо ще декілька прикладів застосування графічних функцій, опис яких наведено в Додатку 8.

В попередній програмі розглядався класичний спосіб ініціалізації графіки (з вікном стандартного розміру) за допомогою рядків

```
int gdriver = DETECT, gmode;
initgraph(&gdriver, &gmode, " ");
```

Наступна програма демонструє інший спосіб, який дозволяє створювати вікно заданого розміру.

Приклад №1. Рисування автомобіля.

Текст програми має такий вигляд:

```
#include <graphics.h>
#include <conio.h>

int main(void)
{
    // ініціалізуємо графічне вікно
    initwindow(200, 120);
    setbkcolor(LIGHTGRAY);  // Встановлюємо колір фону
    cleardevice();          // Очищуємо вікно кольором фону
    //================================================
    // рисуємо кузов
    //================================================
    // встановлюємо вид заливки (1 – суцільний),
    // колір заливки (4 – червоний)
    setfillstyle(1, 4);
    // зафарбований прямокутник (нижня частина кузова)
    bar(50, 50, 140, 70);
    // зафарбований прямокутник (верхня частина кузова)
    bar(70, 50, 120, 30);
    //================================================
    // рисуємо колеса
    //================================================
    // встановлюємо білий колір колеса
    setcolor(15);
    // ліве колесо (окружність нижче кузова)
    circle(70, 70, 10);
    // праве колесо (окружність нижче кузова)
    circle(120, 70, 10);
    // встановлюємо вид заливки (1 – суцільний),
    // колір заливки (8 – сірий)
    setfillstyle(1, 8);
    // зафарбовуємо круг лівого колеса до межі кругу
    floodfill(70, 70, 15);
    // зафарбовуємо круг правого колеса до межі кругу
    floodfill(120, 70, 15);
```

```
    // очікуємо натиснення будь-якої клавіші
    getch();
    // вихід з графічного режиму
    closegraph(ALL_WINDOWS);
    return 0;
}
```

Результат виконання графічної програми наведено на рис. 8.23.

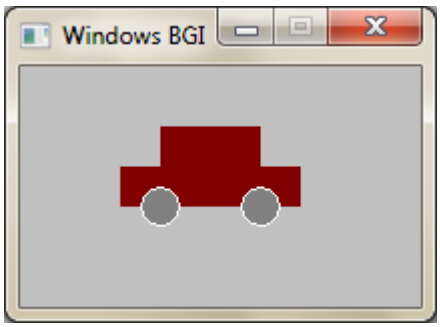


Рисунок 8.23 – Результат роботи програми для рисування автомобіля

Наступний приклад дозволяє розглянути процес рисування в динаміці.

Приклад №2. Побудова фракталу Реслера.

Текст програми має такий вигляд:

```
#include <graphics.h>
#include <conio.h>
#include <math.h>
double x, y, z, dt, a, b, c, x1, y1, z1;
int x2, y2;

int main(void)
{
    // ініціалізуємо графічне вікно
    initwindow(600, 600);
    setbkcolor(WHITE);  // Встановлюємо колір фону
    cleardevice();      // Очищуємо вікно кольором фону
    x  = 3.051522;
    y  = 1.582542;
    z  = 15.62388;
    dt = 0.0001;
    a  = 0.2;
    b  = 0.2;
    c  = 5.7;
    setcolor(getmaxcolor());
    while(!kbhit())
    {
        x1 = x + (-y - z) * dt;
        y1 = y + (x + a * y) * dt;
        z1 = z + (b + z * (x - c)) * dt;
```

```
            x = x1;
            y = y1;
            z = z1;
            putpixel(ceil(19.3 * (y - x * 0.292893) + 320),
                 ceil(-11 * (z + x * 0.292893) + 392), BLACK);
        }
        getch();
        closegraph();
        return 0;
}
```

Результат виконання графічної програми наведено на рис. 8.24.

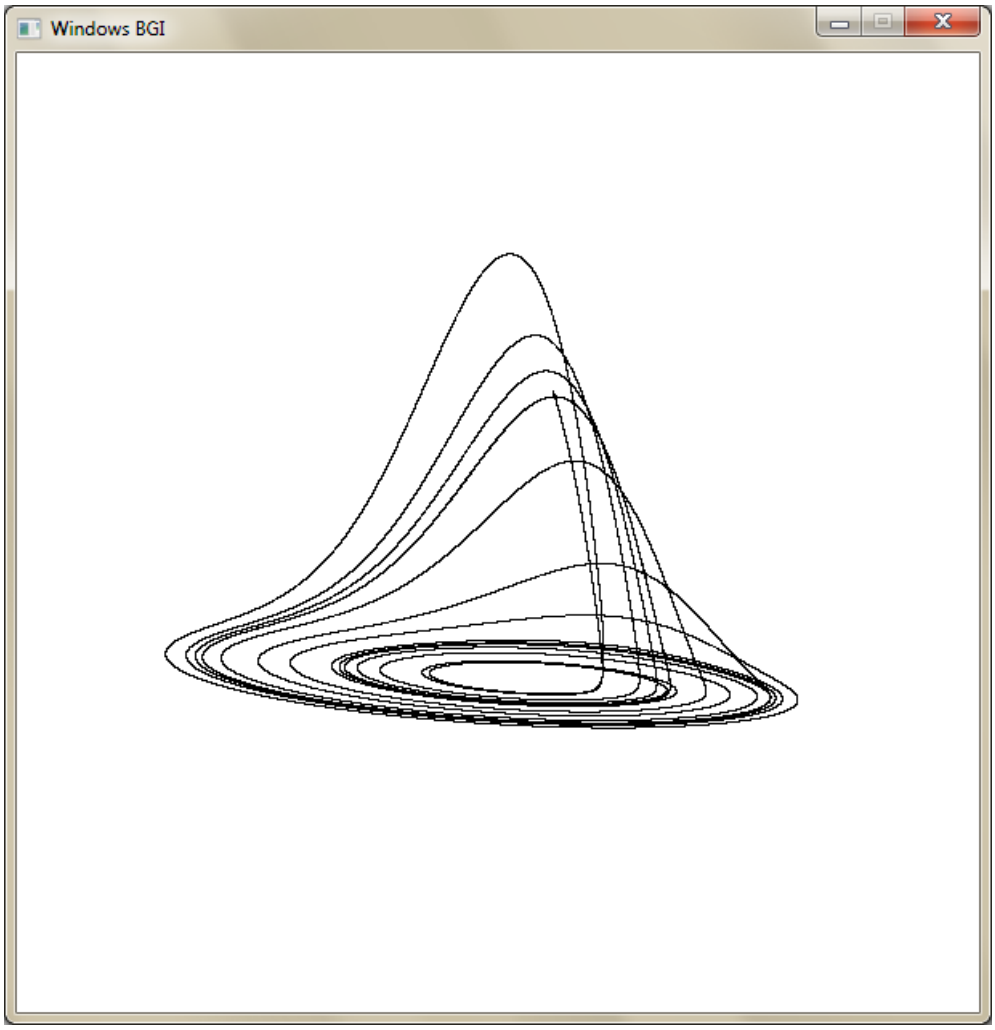


Рисунок 8.24 – Результат роботи програми для фракталу Реслера

Наступний приклад пов’язаний з побудовою фракталу «HenonIFS». Особливість програми полягає в тому, що змінюючи значення **M** в директиві препроцесора можна обирати форму багатокутника, в якій буде формуватися зображення.

Приклад №3. Побудова фракталу «HenonIFS».

Текст програми має такий вигляд:

```
#include <graphics.h>
#include <conio.h>        // для функцій kbhit(), getch()
#include <math.h>
#include <stdlib.h>       // для функції rand()
#define M_PI 3.14159265358979323846
#define M 3  // M-кутник (задаємо кількість кутів > 2)
int main(void)
{
    int i, l;
    float x1, y1, x, y;
    float a[M], b[M];
    initwindow(640, 483);
    setbkcolor(WHITE);
    cleardevice();
    x = 0; y = 0;           // координати точки A (x,y)
    for(i = 0; i < M; i++)
    {
        a[i] = cos(2 * M_PI * i / M);
        b[i] = sin(2 * M_PI * i / M);
    }
    while(!kbhit())         // Цикл виконується, поки не
                            // натиснута клавіша
    {
        l = rand() % M;     // присвоюємо l випадкове значення
                            // (0 < l < M-1)
        if(rand() % M < 2)  // 1-й випадок
        {
            x1 = x / 2 + a[l];
            y1 = y / 2 + b[l];
        }
        else                // 2-й випадок
        {
            x1 = (x * a[l] + y * b[l] + x * x * b[l]) / 6;
            y1 = (y * a[l] - x * b[l] + x * x * a[l]) / 6;
        }
        x = x1;             // присвоюємо значення
        y = y1;             // новим змінним
        // рисуємо точку чорним кольором (BLACK == 0)
        putpixel(320 + ceil(x * 140), 240 + ceil(y * 140), 0);
    }
    getch();                // чекаємо натиснення клавіші
    closegraph(ALL_WINDOWS);
    return 0;
}
```

Результати роботи програми для M = 3 і M = 5 наведені на рис. 8.25 і рис. 8.26.

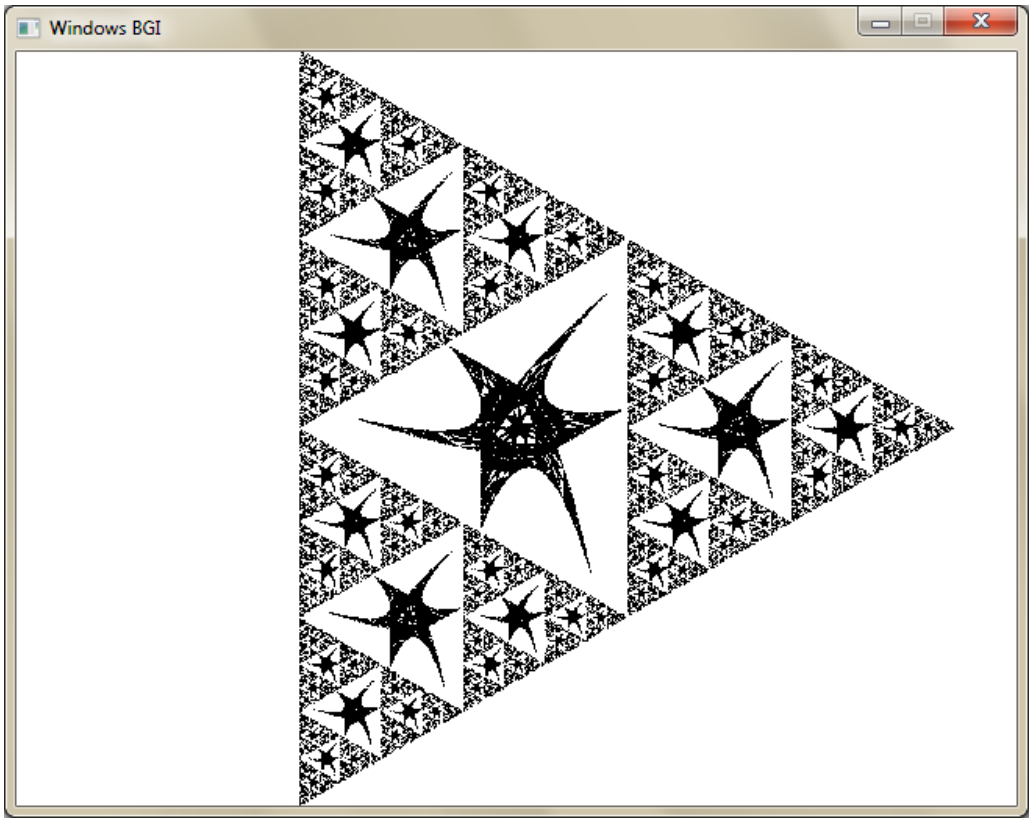


Рисунок 8.25 – Результат роботи програми для фракталу «HenonIFS» (M = 3)

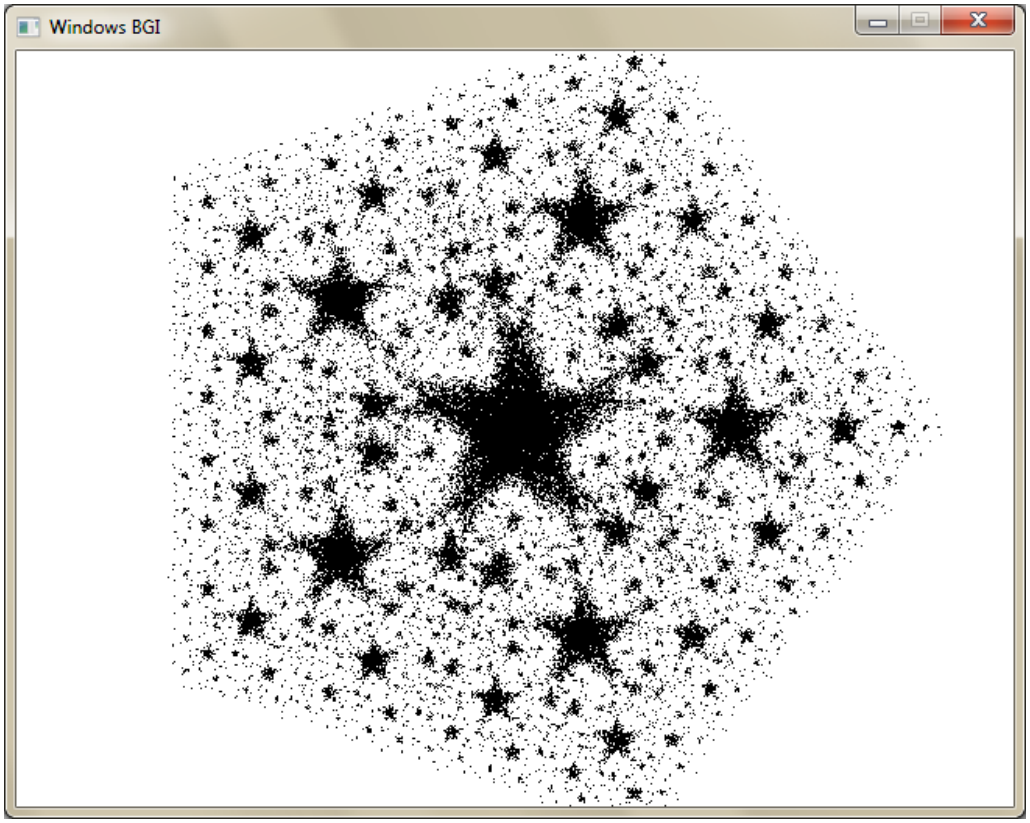


Рисунок 8.26 – Результат роботи програми для фракталу «HenonIFS» (M = 5)

Приклад №4. Необхідно нарисувати двійкове дерево пошуку, що наведено на рис. 8.27. Всі вузли дерева, які підлягають видаленню, необхідно зафарбувати в рожевий колір (вузли 10, 7, 4); вузол 14, для якого треба визначити батьківський вузол – зафарбувати в блакитний колір; інші вузли – зафарбувати в жовтий колір. Також необхідно позначити мінімальний і максимальний вузли та корінь дерева.

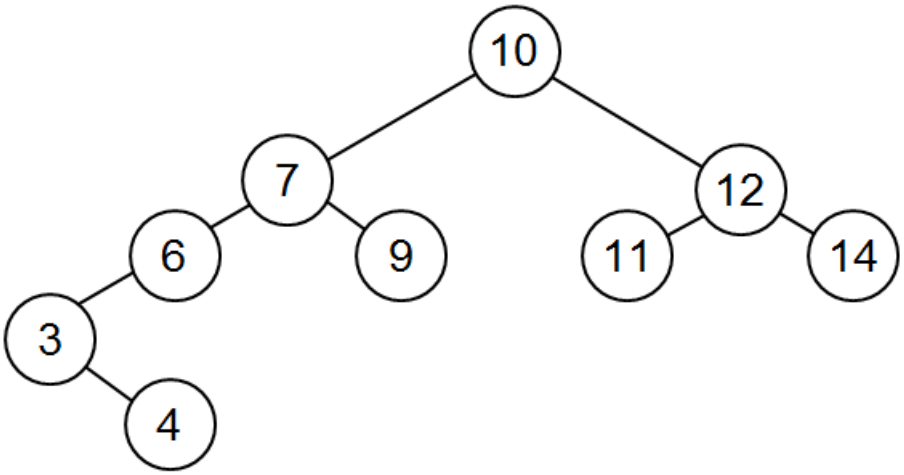


Рисунок 8.27 – Двійкове дерево пошуку

Розглянемо алгоритм побудови двійкового дерева пошуку, яке наведено на рис. 8.27.

1) Спочатку необхідно на аркуші паперу побудувати каркас дерева, який складається зі зв’язків між вузлами та точок, що позначають місця розташування майбутніх вузлів (рис. 8.28).

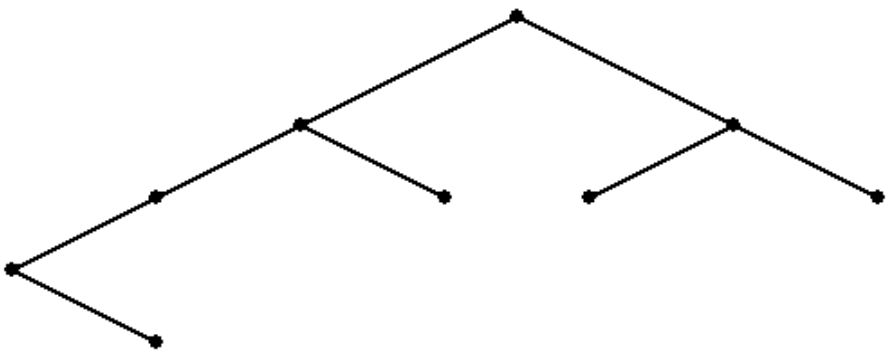


Рисунок 8.28 – Каркас майбутнього двійкового дерева пошуку

2) Далі над кожною точкою треба написати значення вузлів, а під кожною точкою в круглих дужках – номери, що визначають порядок рисування вузлів (рис. 8.29). Вузли, що підлягають видаленню, слід позначити червоним кружечком (значення 10, 7, 4), а вузол, для якого треба визначити батьківський – синім кружечком (значення 14).

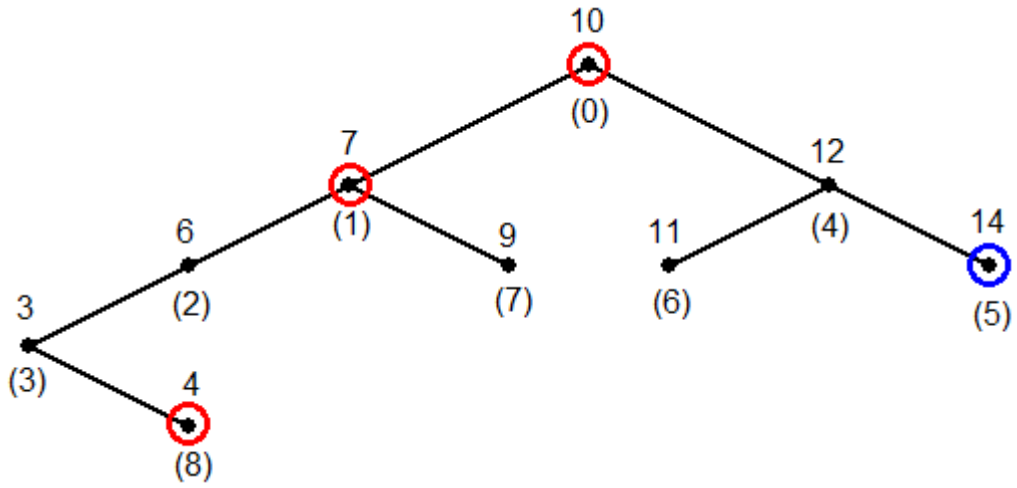


Рисунок 8.29 – Каркас дерева з додатковими позначеннями

3) Потім необхідно переходити до реалізації програми на *C*.

Текст програми, який дозволяє нарисувати двійкове дерево пошуку в динаміці та відповідає розглянутому алгоритму, має такий вигляд:

```
#include <graphics.h>
#include <stdio.h>
#define KOL      9   // Кількість вузлів дерева
#define DELAY   10   // Пауза для рисування лінії
#define DELAY1 500   // Пауза для рисування вузла
#define ROOT    60   // Довжина лінії, що йде від кореня
#define NODE    40   // Довжина лінії, що йде від інших вузлів

void node_data(int x, int y, int value, int color, int n);
void draw_point(int x, int y);

int uzel_x[KOL];
int uzel_y[KOL];
int uzel_zn[KOL];
int uzel_color[KOL];

int main(void)
{
    char s[3];
```

```
// ініціалізуємо графічне вікно
initwindow(640, 300);
// Встановлюємо білий колір для фону вікна
setbkcolor(WHITE);
// Очищуємо вікно кольором фону
cleardevice();
// Встановлюємо чорний колір для лінії
setcolor(BLACK);
// Встановлюємо координати для кореня дерева
int X0 = 360;
int Y0 = 50;
// Заповнюємо інформацію про корінь (вузол 10)
node_data(X0, Y0, 10, RED, 0);
// Позначаємо точкою позицію майбутнього оператора
draw_point(X0, Y0);
// Встановлюємо стиль лінії
setlinestyle(0, 0, 1);
// Встановлюємо точку в початкову позицію
moveto(X0, Y0);
// Рисуємо лінію в динаміці
for(int i = 0; i < ROOT; i++)
{
    lineto(X0 - (2 * i), Y0 + i);
    delay(DELAY);  // Пауза для створення ефекту
                   // анімації
}
// Фіксуємо координати наступного вузла
X0 = X0 - 2 * ROOT;
Y0 = Y0 + ROOT;
// Заповнюємо інформацію про вузол 7
node_data(X0, Y0, 7, RED, 1);
draw_point(X0, Y0);
setlinestyle(0, 0, 1);
for(int i = 0; i < NODE; i++)
{
    lineto(X0 - (2 * i), Y0 + i);
    delay(DELAY);
}
X0 = X0 - 2 * NODE;
Y0 = Y0 + NODE;
// Заповнюємо інформацію про вузол 6
node_data(X0, Y0, 6, YELLOW, 2);
draw_point(X0, Y0);
setlinestyle(0, 0, 1);
for(int i = 0; i < NODE; i++)
{
    lineto(X0 - (2 * i), Y0 + i);
    delay(DELAY);
}
X0 = X0 - 2 * NODE;
Y0 = Y0 + NODE;
```

```
// Заповнюємо інформацію про вузол 3
node_data(X0, Y0, 3, YELLOW, 3);
draw_point(X0, Y0);
// Встановлюємо позицію в точку з координатами кореня
X0 = uzel_x[0];
Y0 = uzel_y[0];
moveto(X0, Y0);
setlinestyle(0, 0, 1);
for(int i = 0; i < ROOT; i++)
{
    lineto(X0 + (2 * i), Y0 + i);
    delay(DELAY);
}
X0 = X0 + 2 * ROOT;
Y0 = Y0 + ROOT;
// Заповнюємо інформацію про вузол 12
node_data(X0, Y0, 12, YELLOW, 4);
draw_point(X0, Y0);
setlinestyle(0, 0, 1);
for(int i = 0; i < NODE; i++)
{
    lineto(X0 + (2 * i), Y0 + i);
    delay(DELAY);
}
X0 = X0 + 2 * NODE;
Y0 = Y0 + NODE;
// Заповнюємо інформацію про вузол 14
node_data(X0, Y0, 14, BLUE, 5);
draw_point(X0, Y0);
// Встановлюємо позицію в точку з координатами вузла 12
X0 = uzel_x[4];
Y0 = uzel_y[4];
moveto(X0, Y0);
setlinestyle(0, 0, 1);
for(int i = 0; i < NODE; i++)
{
    lineto(X0 - (2 * i), Y0 + i);
    delay(DELAY);
}
X0 = X0 - 2 * NODE;
Y0 = Y0 + NODE;
// Заповнюємо інформацію про вузол 11
node_data(X0, Y0, 11, YELLOW, 6);
draw_point(X0, Y0);
// Встановлюємо позицію в точку з координатами вузла 7
X0 = uzel_x[1];
Y0 = uzel_y[1];
moveto(X0, Y0);
setlinestyle(0, 0, 1);
for(int i = 0; i < NODE; i++)
{
    lineto(X0 + (2 * i), Y0 + i);
```

```
    delay(DELAY);
}
X0 = X0 + 2 * NODE;
Y0 = Y0 + NODE;
// Заповнюємо інформацію про вузол 9
node_data(X0, Y0, 9, YELLOW, 7);
draw_point(X0, Y0);
// Встановлюємо позицію в точку з координатами вузла 3
X0 = uzel_x[3];
Y0 = uzel_y[3];
moveto(X0, Y0);
setlinestyle(0, 0, 1);
for(int i = 0; i < NODE; i++)
{
    lineto(X0 + (2 * i), Y0 + i);
    delay(DELAY);
}
X0 = X0 + 2 * NODE;
Y0 = Y0 + NODE;
// Заповнюємо інформацію про вузол 4
node_data(X0, Y0, 4, RED, 8);
draw_point(X0, Y0);
//=====================================================
//  Каркас побудовано. Тепер рисуємо вузли дерева
//=====================================================
setcolor(0);
setlinestyle(0, 0, 1);
for(int i = 0; i < KOL; i++)
{
    X0 = uzel_x[i];
    Y0 = uzel_y[i];
    // Встановлюємо точку в позицію поточного вузла
    moveto(X0, Y0);
    switch(uzel_color[i])  // Задаємо колір зафарбовування
    {                      // вузла та фону для номеру
        case 1: setfillstyle(1, LIGHTBLUE);
                setbkcolor(LIGHTBLUE);
                break;
        case 4: setfillstyle(1, LIGHTRED);
                setbkcolor(LIGHTRED);
                break;
       case 14: setfillstyle(1, YELLOW);
                setbkcolor(YELLOW);
                break;
       default: break;
    }
    // Рисуємо вузол
    fillellipse(X0, Y0, 20, 20);
    // Записуємо номер вузла (його значення)
    sprintf(s, "%d", uzel_zn[i]);
    int t = strlen(s);
    if(t == 1)  // Номер з однієї цифри (наприклад, 8)
```

```
            outtextxy(uzel_x[i] - 4, uzel_y[i] - 7, s);
        else        // Номер з двома цифрами (наприклад, 12)
            outtextxy(uzel_x[i] - 8, uzel_y[i] - 7, s);
        delay(DELAY1);
    }
    //=====================================================
    //  Підписуємо max, min і корінь
    //=====================================================
    setbkcolor(WHITE);
    setcolor(LIGHTRED);
    outtextxy(uzel_x[3] - 12, uzel_y[3] + 24, "min");
    delay(DELAY1);
    outtextxy(uzel_x[5] - 12, uzel_y[5] + 24, "max");
    setcolor(LIGHTBLUE);
    delay(DELAY1);
    outtextxy(uzel_x[0] - 20, uzel_y[0] - 40, "корінь");
    getch();
    return 0;
}

// Функція для заповнення інформації про вузол
void node_data(int x, int y, int value, int color, int n)
{
    uzel_x[n]     = x;       // Координата x
    uzel_y[n]     = y;       // Координата y
    uzel_zn[n]    = value;   // Значення для кореня
    uzel_color[n] = color;   // Колір
}

// Функція для рисування точки
void draw_point(int x, int y)
{
    // Рисуємо зафарбований круг радіусом 4 пікселя
    fillellipse(x, y, 4, 4);
    setfillstyle(1, BLACK);
    floodfill(x, y, BLACK);
}
```

Результат роботи програми наведено на рис. 8.30.

Хід виконання роботи:

1. Опрацюйте матеріал лекції «Графіка на *C*», а також теоретичний матеріал даної лабораторної роботи. Налаштуйте портативну версію *Code::Blocks* для роботи з графічною бібліотекою WinBGIm.

2. Напишіть *C*–програму, яка будує графік функції відповідно до вашого варіанта (додаток 8). Приклад побудови графіка функції наведено в лекції.

3. Напишіть *C*–програму, яка рисує двійкове дерево пошуку з практичної роботи №7 відповідно до вашого варіанта. В дереві необхідно позначити: три вузли, що підлягають видаленню; вузол, для якого треба знайти батьківський вузол; мінімальний і максимальний елементи, а також корінь дерева. Прототипи функцій для роботи з графікою наведені в додатку 9.

4. Напишіть *C*–програму, яка створює рисунок відповідно до вашого варіанта (додаток 10). Рисунок необхідно зафарбувати на свій розсуд. Позначте в правому нижньому куті графічного вікна свою групу та прізвище з ініціалами.

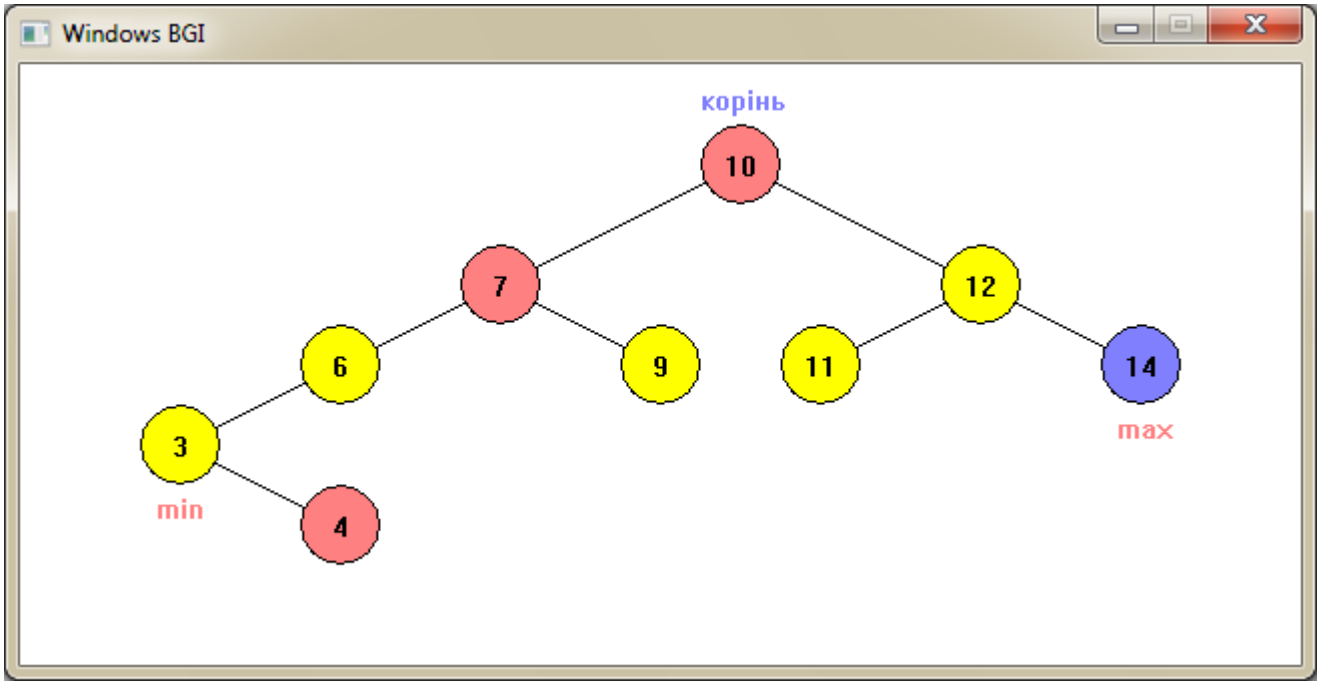


Рисунок 8.30 – Результат рисування двійкового дерева пошуку

Звіт про виконання лабораторної роботи повинен містити:

1. Титульний аркуш.
2. Мету роботи.
3. Результати виконання роботи:
   – тексти *C*–програм;
   – результати виконання *C*–програм.
4. Висновки по роботі.

Контрольні запитання:

1. Який заголовний файл необхідно підключити для роботи з графікою?

2. Що собою являє бібліотека WinBGIm? Як її налаштувати для роботи в середовищі *Code::Blocks*?

3. Які функції застосовуються для ініціалізації графічного режиму та закінчення роботи з графікою?

4. Яким чином можна створити графічне вікно для виводу графічної інформації?

5. За допомогою яких функцій можна отримати максимально можливі координати графічного вікна?

6. За допомогою яких функцій можна визначити поточні координати вказівника (графічного курсору)?

7. За допомогою яких функцій можна здійснювати переміщення графічного курсору без прорисовування на екрані?

8. Які функції забезпечують рисування точок і ліній у графічному вікні? Які додаткові функції при цьому слід застосовувати?

9. Які функції забезпечують рисування прямокутників у графічному вікні?

10. Які функції забезпечують рисування окружностей та еліпсів у графічному вікні?

11. Як вивести текст у графічному режимі? Що для цього треба зробити?

12. Яким чином можна забезпечити ефект анімації при рисуванні у графічному вікні?

СПИСОК ДЖЕРЕЛ ІНФОРМАЦІЇ

1. Дейтел П., Дейтел Х. С для программистов с введением в С11 – М.: ДМК Пресс, 2014. – 544 с.

2. Дейтел П., Дейтел Х. Как программировать на С, 7–е издание, 2014. – 1000 с.

3. Керниган Б., Ритчи Д. Язык программирования С – М.: Диалектика–Вильямс, 2015. – 288 с.

4. Прата С. Язык программирования С. Лекции и упражнения – М.: Диалектика–Вильямс, 2015. – 928 с.

5. Шилдт Г. С: полное руководство, классическое издание – М.: "Диалектика–Вильямс", 2017. – 704 с.

6. Купа (пам’ять). Матеріал з Вікіпедії – вільної енциклопедії. URL: https://uk.wikipedia.org/wiki/%D0%9A%D1%83%D0%BF%D0%B0_(%D0%BF%D0%B0%D0%BC%27%D1%8F%D1%82%D1%8C) (дата звернення: 07.02.2020).

7. Односвязный список. URL: https://learnc.info/adt/linked_list.html (дата обращения: 07.12.2018).

8. Седжвик, Роберт. Алгоритмы на С++. : Пер. с англ. – М. : ООО "И.Д. Вильямс", 2016. – 1056 с. : ил. – Парал. тит. англ.

9. Двусвязный линейный список. URL: https://prog–cpp.ru/data–dls/ (дата обращения: 11.02.2019).

ДОДАТКИ

Додаток 1

Задачі до лабораторної роботи №1

Для структури даних, яка наведена на рис. Д 1.1, необхідно створити масив записів для бази даних «Товар» та визначити:

1) фірму, товар якої надходить частіше за всі інші;

2) фірму, товар якої надходить рідше за всі інші;

3) суму вартості всіх товарів, поставлених протягом 2020–2021 років;

4) всіх постачальників товарів, країною походження яких є Україна, рік постачання – 2020, вартість товару – більше ніж 3000 грн;

5) найбільш розповсюджену країну постачальника за 2020–2021 роки;

6) фірми постачальники, у найменуванні товарів яких є слово «монітор» та вартість яких не перевищує 1800 грн, а дата постачання – квітень 2021 року;

7) суму вартості товарів кожної країни за 2020–2021 роки;

8) фірму постачальника з максимальною сумою вартості товарів, що були поставлені у 2020 році;

9) постачальників з максимальною сумою товарів у кожному році;

10) постачальників з України, які продавали комп’ютери у 2021 році.

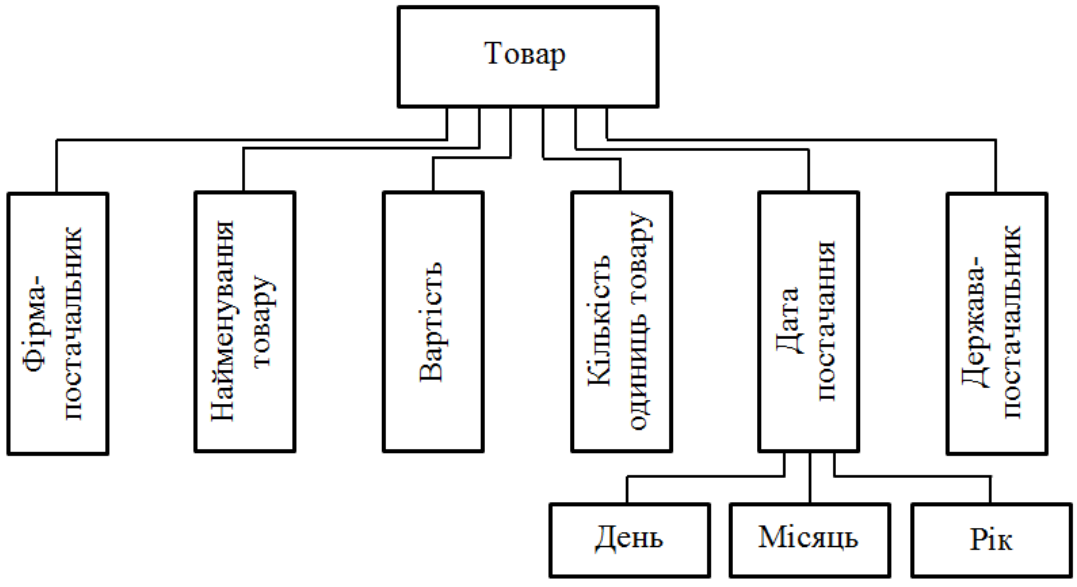


Рисунок Д 1.1 – Схема бази даних «Товар»

Продовження додатка 1

Для структури даних, яка наведена на рис. Д 1.2, необхідно створити масив записів для бази даних «Будова» та визначити:

11) клієнтів, у яких сума банківського вкладу менш ніж вартість будови;

12) клієнта, який повністю розрахувався за будову (невиплачена сума дорівнює нулю);

13) тип будови, яку клієнти замовляли більш ніж 2 рази;

14) тип будови з максимальною вартістю;

15) невиплачену суму всіх клієнтів;

16) клієнта з максимальною невиплаченою сумою;

17) клієнтів, у яких сума вкладу менш ніж невиплачена сума за кредит;

18) клієнтів з максимальним та мінімальним відсотками за кредит;

19) фірму підрядника, у якої клієнт має мінімальну невиплачену суму;

20) тип будови з максимальною вартістю, якщо відсоток за кредит клієнта перевищує 10%.

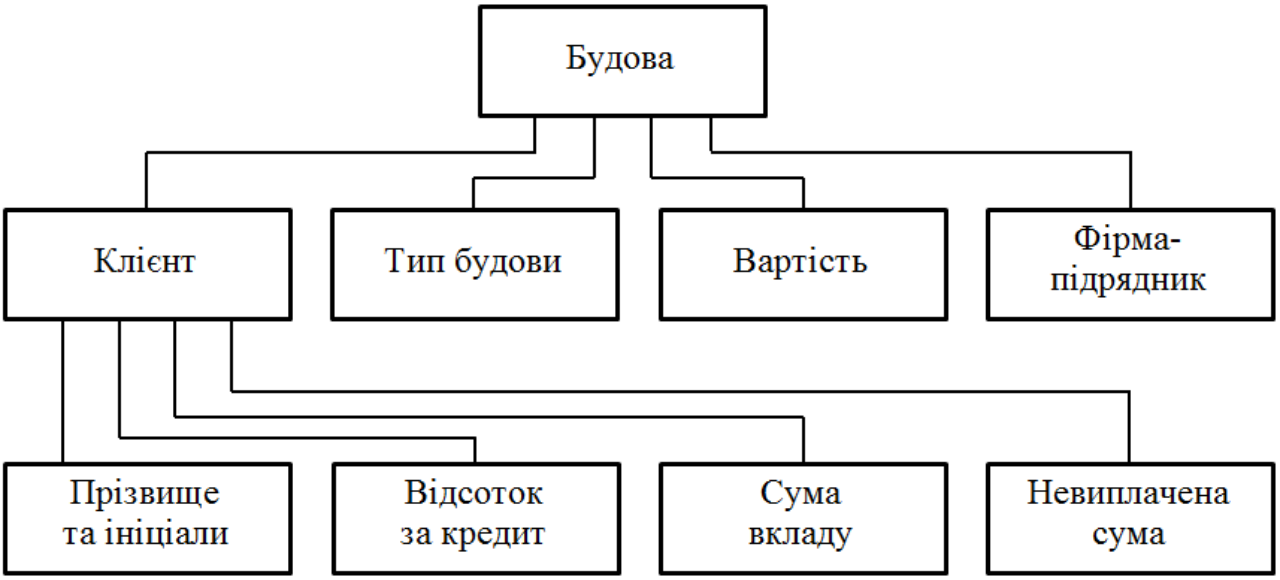


Рисунок Д 1.2 – Схема бази даних «Будова»

Для структури даних, яка наведена на рис. Д 1.3, необхідно створити масив записів для бази даних «Господарча діяльність» та визначити:

21) загальну собівартість реалізованого товару;

22) найбільш рентабельний вид діяльності, виходячи з показників чистого прибутку;

23) фірми, у яких чистий прибуток вище його середнього значення за всіма фірмами;

24) найнерентабельніший вид діяльності, виходячи з показників чистого прибутку;

25) фірми, у яких вид діяльності «Послуги», а об’єм продажу більш ніж 10000 одиниць;

26) товари з мінімальними видатками, у назві яких є слово «телефон».

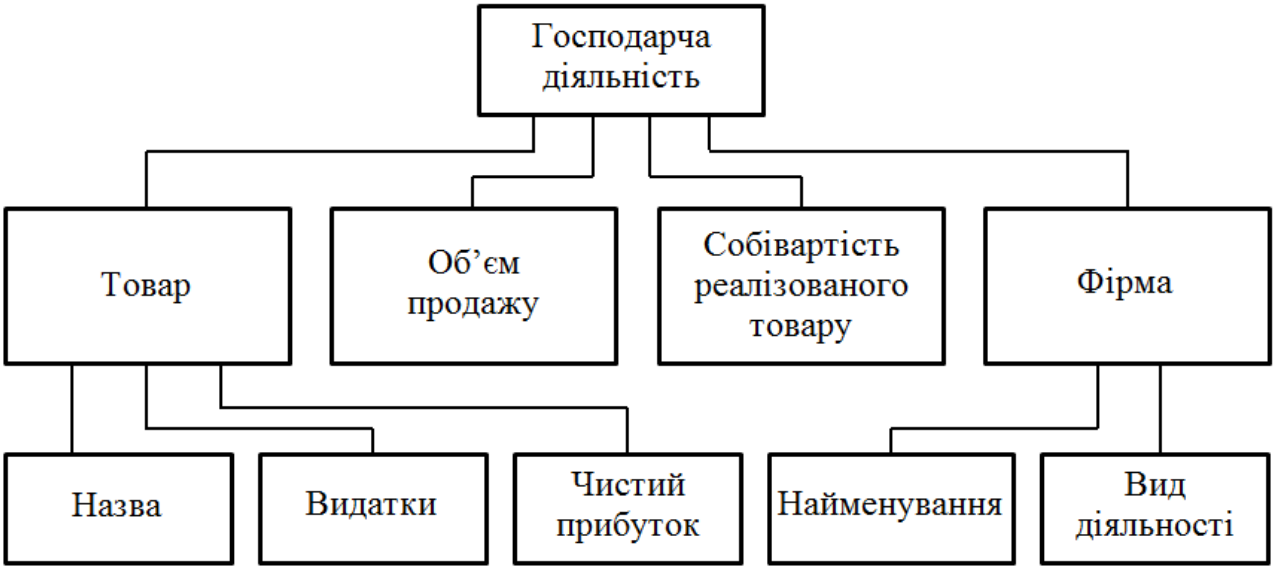


Рисунок Д 1.3 – Схема бази даних «Господарча діяльність»

Додаток 2

Приклад виконання завдання 2 лабораторної роботи №2

Для виконання завдання 2 необхідно в наведеній нижче програмі замінити відповідні значення на значення з табл. 2.1 для вашого варіанта та пояснити отримані результати.

Продовження додатка 2

```
#include <stdio.h>
#include <windows.h>
union Number                // визначення об’єднання Number
{
    unsigned short  s;
    unsigned int    i;
    unsigned long   l;
    float           f;
    double          d;
    unsigned char   c;
    unsigned char   ch[8];
};

int main(void)
{
    union Number value;     // визначаємо змінну типу "об’єднання"
    SetConsoleOutputCP(1251);
    value.d = 0.000000;     // очищуємо вміст об'єднання
    value.s = 23234;        // треба замінити на значення з таблиці
    printf("value.s = %hu;\n", value.s);
    printf("=====================================\n");
    printf("Розмір типу short в байтах:   %d\n", sizeof(short));
    printf("=====================================\n");
    printf("%c short:     %hu\n",  169, value.s);
    printf("  int:       %u\n",   value.i);
    printf("  long:      %lu\n",  value.l);
    printf("  float:     %f\n",   value.f);
    printf("  double:    %f\n",   value.d);
    printf("  char:      '%c'\n",   value.c);
    printf("  char[8]:   .%-8s.\n", value.ch);
    printf("=====================================\n");
    printf("Результат:  ");
    printf("|%02X|%02X|%02X|%02X|%02X|%02X|%02X|%02X|",
           value.ch[7], value.ch[6], value.ch[5], value.ch[4],
           value.ch[3], value.ch[2], value.ch[1], value.ch[0]);
    printf("\n=====================================\n\n");
    value.d = 0.000000;     // очищуємо вміст об'єднання
    value.i = 3136586480;  // треба замінити на значення з таблиці
    printf("value.i = %u;\n", value.i);
    printf("=====================================\n");
    printf("Розмір типу int в байтах:     %d\n", sizeof(int));
    printf("=====================================\n");
    printf("  short:     %hu\n",  value.s);
    printf("%c int:       %u\n",   169, value.i);
    printf("  long:      %lu\n",  value.l);
    printf("  float:     %f\n",   value.f);
    printf("  double:    %f\n",   value.d);
    printf("  char:      '%c'\n",   value.c);
    printf("  char[8]:   .%-8s.\n", value.ch);
    printf("=====================================\n");
```

```
printf("Результат:  ");
printf("|%02X|%02X|%02X|%02X|%02X|%02X|%02X|%02X|",
       value.ch[7], value.ch[6], value.ch[5], value.ch[4],
       value.ch[3], value.ch[2], value.ch[1], value.ch[0]);
printf("\n=====================================\n\n");
value.d = 0.000000;   // очищуємо вміст об'єднання
value.l = 400000057L; // треба замінити на значення з таблиці
printf("value.l = %lu;\n", value.l);
printf("=====================================\n");
printf("Розмір типу long в байтах:    %d\n", sizeof(long));
printf("=====================================\n");
printf("  short:     %hu\n",  value.s);
printf("  int:       %u\n",   value.i);
printf("%c long:      %lu\n",  169, value.l);
printf("  float:     %f\n",   value.f);
printf("  double:    %f\n",   value.d);
printf("  char:      '%c'\n",   value.c);
printf("  char[8]:   .%-8s.\n", value.ch);
printf("=====================================\n");
printf("Результат:  ");
printf("|%02X|%02X|%02X|%02X|%02X|%02X|%02X|%02X|",
       value.ch[7], value.ch[6], value.ch[5], value.ch[4],
       value.ch[3], value.ch[2], value.ch[1], value.ch[0]);
printf("\n=====================================\n\n");

value.d = 0.000000;  // очищуємо вміст об'єднання
value.f = 34.87;     // треба замінити на значення з таблиці
printf("value.f = %5.2f;\n", value.f);
printf("=====================================\n");
printf("Розмір типу float в байтах:   %d\n", sizeof(float));
printf("=====================================\n");
printf("  short:     %hu\n",   value.s);
printf("  int:       %u\n",    value.i);
printf("  long:      %lu\n",   value.l);
printf("%c float:     %.2f\n", 169, value.f);
printf("  double:    %f\n",    value.d);
printf("  char:      '%c'\n",    value.c);
printf("  char[8]:   .%-8s.\n", value.ch);
printf("=====================================\n");
printf("Результат:  ");
printf("|%02X|%02X|%02X|%02X|%02X|%02X|%02X|%02X|",
       value.ch[7], value.ch[6], value.ch[5], value.ch[4],
       value.ch[3], value.ch[2], value.ch[1], value.ch[0]);
printf("\n=====================================\n\n");
value.d = 5.12;     // треба замінити на значення з таблиці
printf("value.d = %5.2f;\n", value.d);
printf("=====================================\n");
printf("Розмір типу double в байтах:  %d\n", sizeof(double));
printf("=====================================\n");
printf("  short:     %hu\n",   value.s);
printf("  int:       %u\n",    value.i);
```

Продовження додатка 2

```
printf("  long:      %lu\n",   value.l);
printf("  float:     %f\n",    value.f);
printf("%c double:    %.2f\n", 169, value.d);
printf("  char:      '%c'\n",    value.c);
printf("  char[8]:   .%-8s.\n", value.ch);
printf("=====================================\n");
printf("Результат:  ");
printf("|%02X|%02X|%02X|%02X|%02X|%02X|%02X|%02X|",
       value.ch[7], value.ch[6], value.ch[5], value.ch[4],
       value.ch[3], value.ch[2], value.ch[1], value.ch[0]);
printf("\n=====================================\n\n");
value.d = 0.000000;  // очищуємо вміст об'єднання
value.c = 'k';       // треба замінити на значення з таблиці
printf("value.c = '%c';\n", value.c);
printf("=====================================\n");
printf("Розмір типу char в байтах:    %d\n", sizeof(char));
printf("=====================================\n");
printf("  short:     %hu\n",  value.s);
printf("  int:       %u\n",   value.i);
printf("  long:      %lu\n",  value.l);
printf("  float:     %f\n",   value.f);
printf("  double:    %f\n",   value.d);
printf("%c char:      '%c'\n",   169, value.c);
printf("  char[8]:   .%-8s.\n", value.ch);
printf("=====================================\n");
printf("Результат:  ");
printf("|%02X|%02X|%02X|%02X|%02X|%02X|%02X|%02X|",
       value.ch[7], value.ch[6], value.ch[5], value.ch[4],
       value.ch[3], value.ch[2], value.ch[1], value.ch[0]);
printf("\n=====================================\n\n");
value.d = 0.000000;  // очищуємо вміст об'єднання
// треба замінити на своє прізвище (не більше 7 букв)
value.ch[0] = 'd';   value.ch[1] = 'e';    value.ch[2] = 'f';
value.ch[3] = 'a';   value.ch[4] = 'u';    value.ch[5] = 'l';
value.ch[6] = 't';   value.ch[7] = '\x0';
printf("value.ch[0] = '%c'; value.ch[1] = '%c';\n",
       value.ch[0], value.ch[1]);
printf("value.ch[2] = '%c'; value.ch[3] = '%c';\n",
       value.ch[2], value.ch[3]);
printf("value.ch[4] = '%c'; value.ch[5] = '%c';\n",
       value.ch[4], value.ch[5]);
printf("value.ch[6] = '%c'; value.ch[7] = \'\\x0\'\n",
       value.ch[6]);
printf("=====================================\n");
printf("Розмір масиву ch[8] в байтах: %d\n",
       sizeof(value.ch));
printf("=====================================\n");
printf("  short:     %hu\n",  value.s);
printf("  int:       %u\n",   value.i);
printf("  long:      %lu\n",  value.l);
```

Продовження додатка 2

```
    printf("  float:     %.2f\n",   value.f);
    printf("  double:    %f\n",   value.d);
    printf("  char:      '%c'\n",   value.c);
    printf("%c char[8]:   .%-8s.\n", 169, value.ch);
    printf("=====================================\n");
    printf("Результат:  ");
    printf("|%02X|%02X|%02X|%02X|%02X|%02X|%02X|%02X|",
           value.ch[7], value.ch[6], value.ch[5], value.ch[4],
           value.ch[3], value.ch[2], value.ch[1], value.ch[0]);
    printf("\n=====================================\n\n");
    printf("=====================================\n");
    printf("Розмір об'єднання value:      %d\n", sizeof(value));
    printf("=====================================\n\n");
    return 0;
}
```

Результат роботи програми наведено на рис. Д 2.1 – Д 2.3.

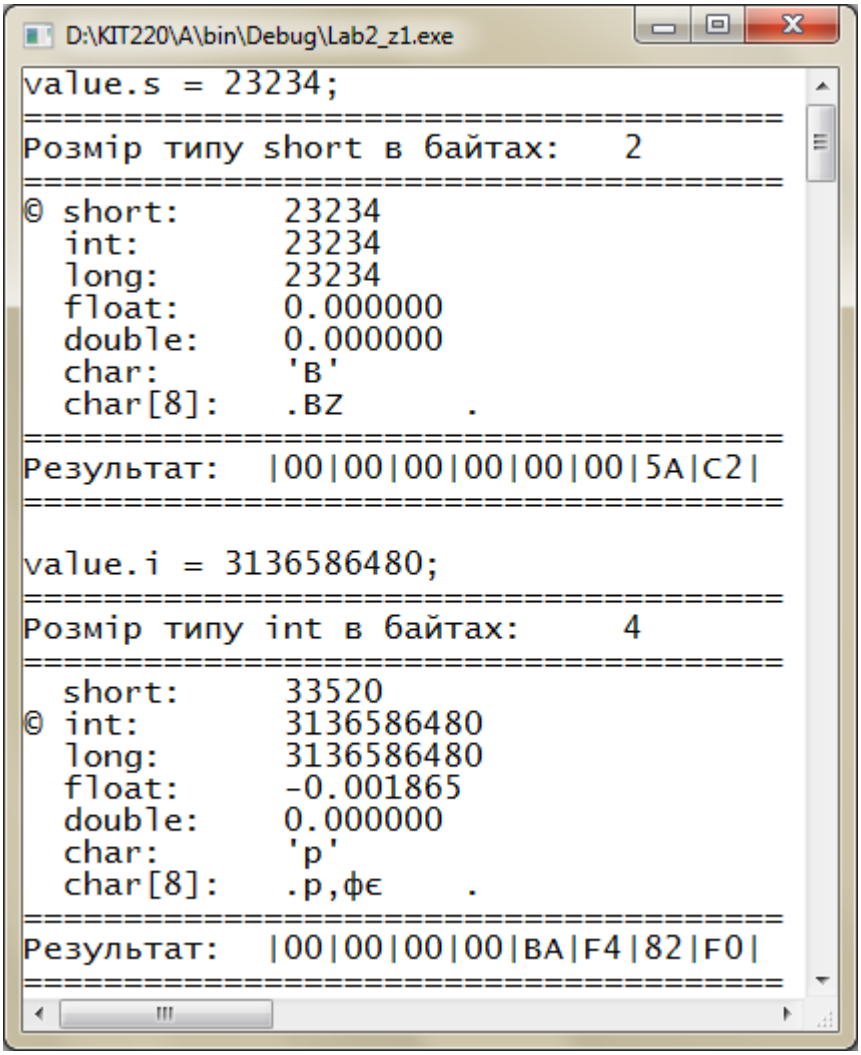


Рисунок Д 2.1 – Результат роботи програми для типів **short** та **int**

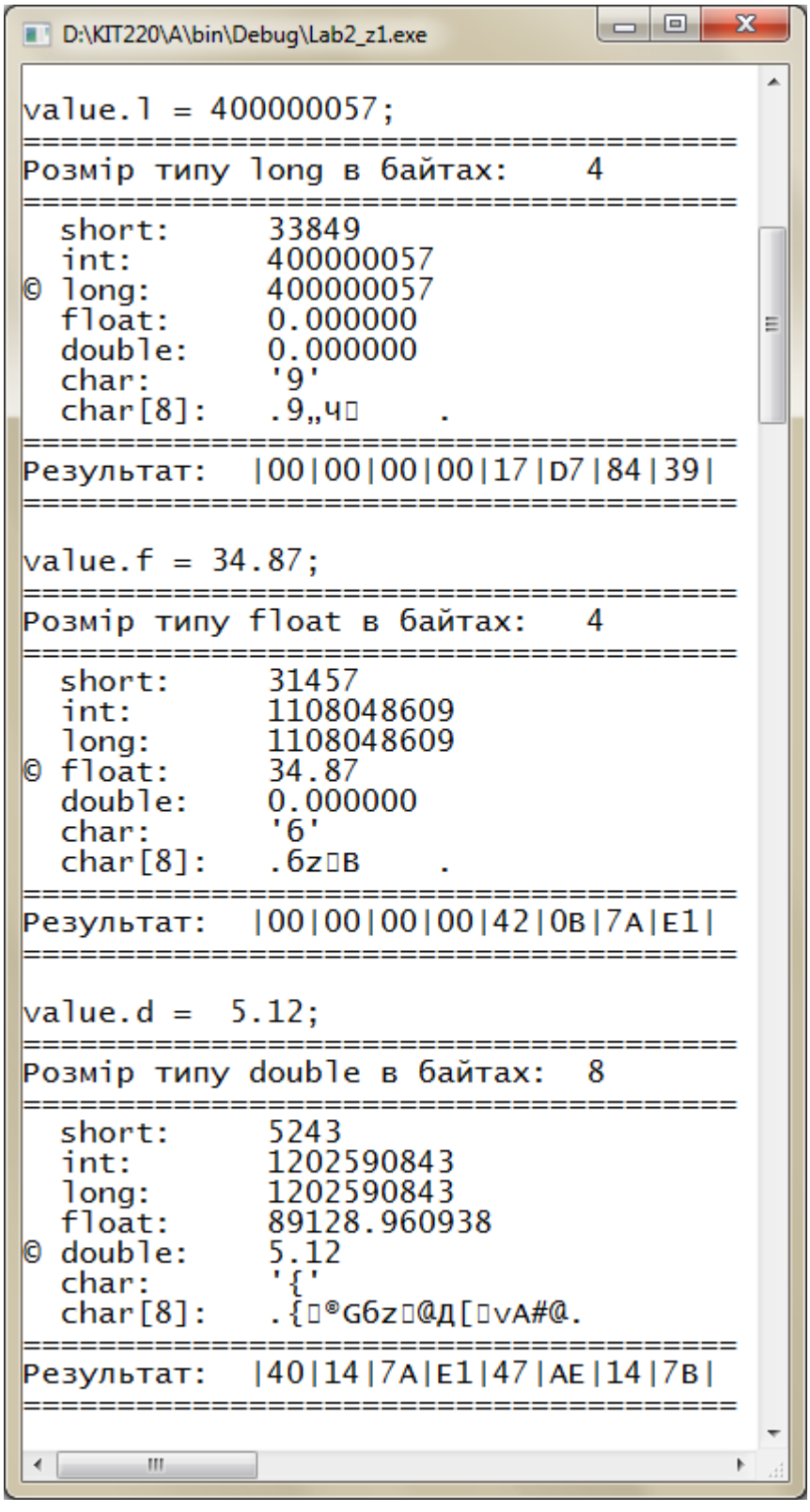


Рисунок Д 2.2 – Результат роботи програми для типів **`long`**, **`float`** і **`double`**

Продовження додатка 2

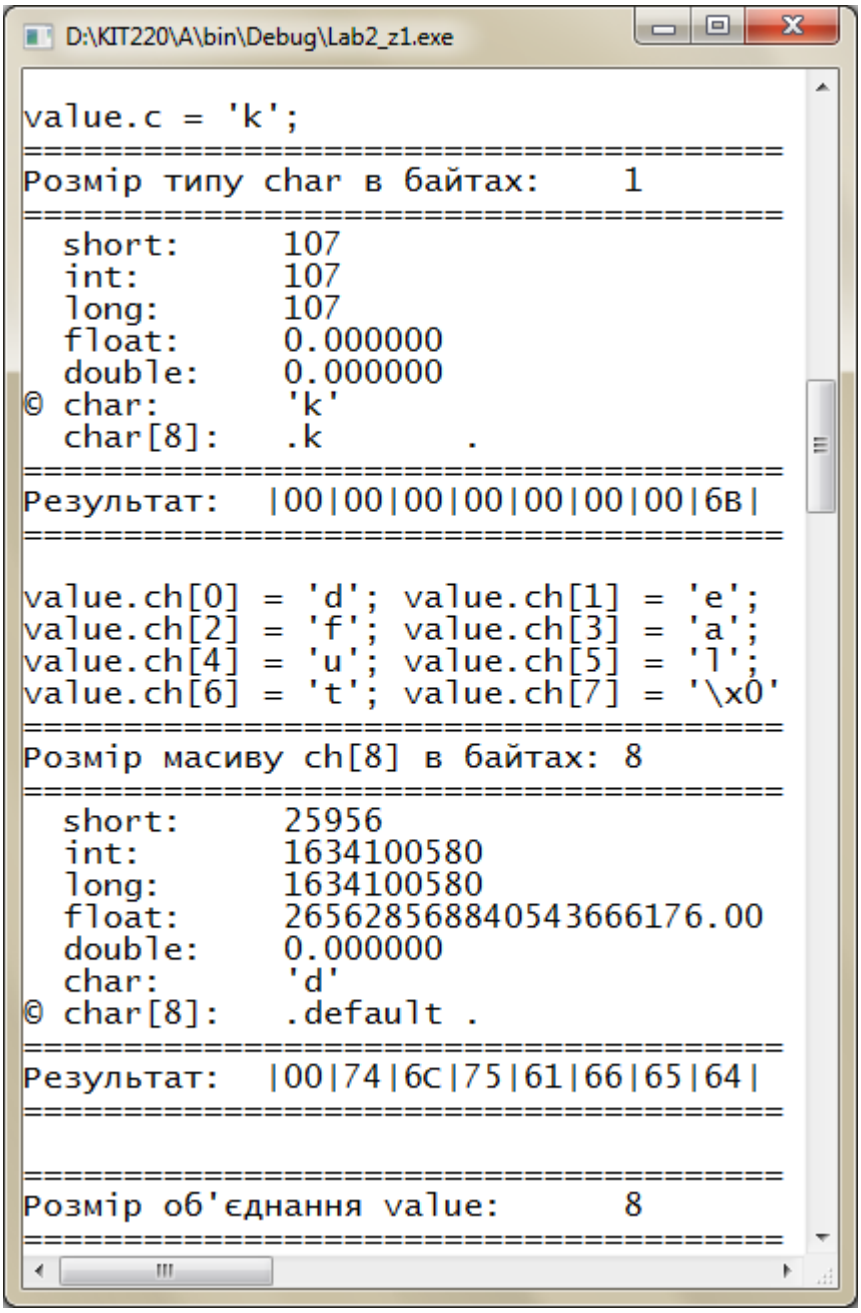


Рисунок Д 2.3 – Результат роботи програми для типу **`char`** і масиву типу **`char`**

Додаток 3

Приклад виконання завдання 3 лабораторної роботи №2

Для виконання завдання 3 необхідно в наведеній нижче програмі замість існуючого створити свій власний перелічуваний тип та вивести значення для його констант за замовчуванням.

Продовження додатка 3

```
#include <stdio.h>
#include <windows.h>
//=============================================================
// Перелічення констант, що являють собою номери годин
//=============================================================
enum Hours  // визначення переліченння Hours
{
    ZERO, ONE,   TWO,   THREE, FOUR, FIVE,
    SIX,  SEVEN, EIGHT, NINE,  TEN,  ELEVEN
};
int main(void)
{
    SetConsoleOutputCP(1251);
    printf("================================================\n");
    printf("Значення констант переліченння за замовчуванням \n");
    printf("================================================\n");
    printf("   ZERO   = %2d",   ZERO);
    printf("   ONE    = %2d",   ONE);
    printf("   TWO    = %2d\n", TWO);
    printf("   THREE  = %2d",   THREE);
    printf("   FOUR   = %2d",   FOUR);
    printf("   FIVE   = %2d\n", FIVE);
    printf("   SIX    = %2d",   SIX);
    printf("   SEVEN  = %2d",   SEVEN);
    printf("   EIGHT  = %2d\n", EIGHT);
    printf("   NINE   = %2d",   NINE);
    printf("   TEN    = %2d",   TEN);
    printf("   ELEVEN = %2d\n", ELEVEN);
    printf("================================================\n\n");
    enum Hours  // визначення переліченння Hours
    {
        ZERO, ONE = 7, TWO,   THREE, FOUR,     FIVE,
        SIX,  SEVEN,   EIGHT, NINE,  TEN = 4,  ELEVEN
    };
    printf("================================================\n");
    printf("Змінюємо значення ( ONE = 7, TEN = 4 )          \n");
    printf("================================================\n");
    printf("   ZERO   = %2d",   ZERO);
    printf("   ONE    = %2d",   ONE);
    printf("   TWO    = %2d\n", TWO);
    printf("   THREE  = %2d",   THREE);
    printf("   FOUR   = %2d",   FOUR);
    printf("   FIVE   = %2d\n", FIVE);
    printf("   SIX    = %2d",   SIX);
    printf("   SEVEN  = %2d",   SEVEN);
    printf("   EIGHT  = %2d\n", EIGHT);
    printf("   NINE   = %2d",   NINE);
    printf("   TEN    = %2d",   TEN);
    printf("   ELEVEN = %2d\n", ELEVEN);
    printf("================================================\n\n");
```

```
    enum Hours1  // визначення перелічення Hours1
    {
        ZERO1, ONE1 = 2, TWO1,         THREE1, FOUR1,  FIVE1,
        SIX1,  SEVEN1,   EIGHT1 = -1, NINE1,  TEN1,   ELEVEN1
    };
    printf("================================================\n");
    printf("Змінюємо значення ( ONE = 2, EIGHT = -1 )       \n");
    printf("================================================\n");
    printf("   ZERO   = %2d",   ZERO1);
    printf("   ONE    = %2d",   ONE1);
    printf("   TWO    = %2d\n", TWO1);
    printf("   THREE  = %2d",   THREE1);
    printf("   FOUR   = %2d",   FOUR1);
    printf("   FIVE   = %2d\n", FIVE1);
    printf("   SIX    = %2d",   SIX1);
    printf("   SEVEN  = %2d",   SEVEN1);
    printf("   EIGHT  = %2d\n", EIGHT1);
    printf("   NINE   = %2d",   NINE1);
    printf("   TEN    = %2d",   TEN1);
    printf("   ELEVEN = %2d\n", ELEVEN1);
    printf("================================================\n\n");
    return 0;
}
```

Результат роботи програми наведено на рис. Д 3.1.

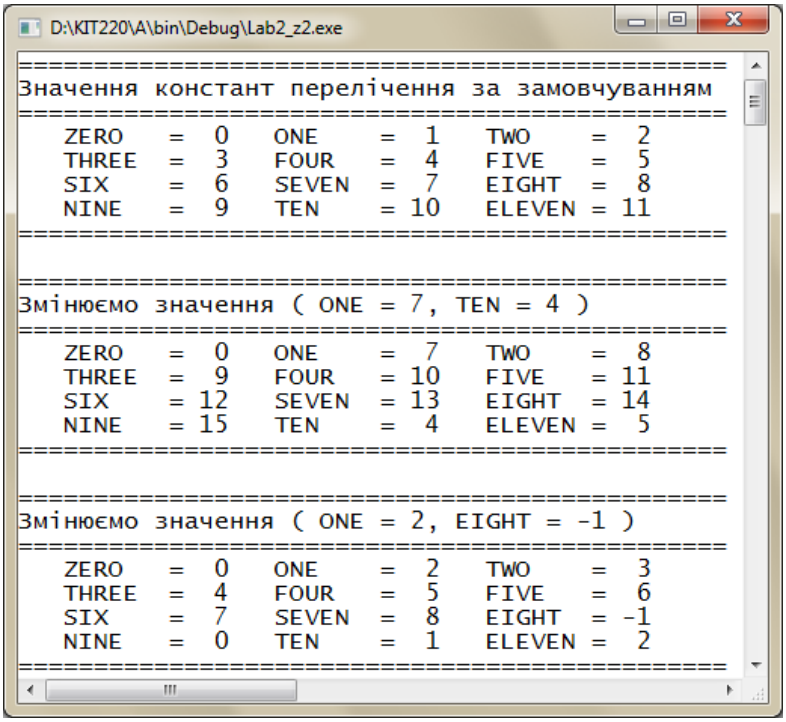


Рисунок Д3.1 – Результат виконання програми для демонстрації можливостей роботи з переліченнями

Додаток 4

Приклад виконання завдання 5 лабораторної роботи №2

Для виконання завдання 5 необхідно в наведеній нижче програмі визначити власну структуру та значення для кількості біт в кожному з полів структури. Далі необхідно присвоїти довільні значення для кожного з полів. У першому випадку значення повинні вміщуватися у відведену кількість бітів, а в другому – перевищувати цю кількість. Отримані результати необхідно пояснити.

```
#include <stdio.h>
#include <windows.h>
#include <limits.h>
#define KOL 8
#define N   5

// Визначення структури з бітовими полями
struct
{
    unsigned int field1 : 2;
    unsigned int field2 : 4;
    unsigned int field3 : 5;
} bstr;

void show_bstr(const char *str);
char *itobs(int n, char *ps);

int main(void)
{
    char bin_str[CHAR_BIT * sizeof(unsigned int) + 1];

    SetConsoleOutputCP(1251);

    printf("sizeof(bstr) = %d\n", sizeof(bstr));

    printf("=============================================\n");
    printf("Визначення бітових полів                     \n");
    printf("=============================================\n");
    printf("    unsigned int field1 : 2;\n"
           "    unsigned int field2 : 4;\n"
           "    unsigned int field3 : 5;\n");
    printf("=============================================\n\n");
    printf("=============================================\n");
    printf("Присвоєння значень бітовим полям             \n");
    printf("=============================================\n");
    bstr.field1 = 2;
    bstr.field2 = 7;
    bstr.field3 = 15;
    printf("    bstr.field1 = %5d;\n"
```

```
       "    bstr.field2 = %5d;\n"
       "    bstr.field3 = %5d;\n", 2, 7, 15);
printf("=============================================\n\n");
printf("bstr = %u\n", bstr);

unsigned int *rab = &bstr;
unsigned int s = *rab;
unsigned int k;
int j = CHAR_BIT * sizeof(bstr) - 1;

for(int i = 0; i < CHAR_BIT * sizeof(bstr); i++)
{
    k = (s >> i) & 1;
    bin_str[j] = (k == 0) ? '0' : '1';
    j--;
}
bin_str[32] = '\x0';
printf("=============================================\n");
for(int i = 1; i < 33; i++)
{
    printf("%c", bin_str[i-1]);
    if(i % 4 == 0) printf("%c", ' ');
}
printf("\n");
printf("=============================================\n");

itobs(bstr.field1, bin_str);
show_bstr(bin_str);
printf("%5d\n", bstr.field1);

itobs(bstr.field2, bin_str);
show_bstr(bin_str);
printf("%5d\n", bstr.field2);

itobs(bstr.field3, bin_str);
show_bstr(bin_str);
printf("%5d\n", bstr.field3);
printf("=============================================\n\n");
printf("=============================================\n");
printf("Присвоєння значень бітовим полям             \n");
printf("=============================================\n");
bstr.field1 = 9;
bstr.field2 = 488;
bstr.field3 = 5602;
printf("    bstr.field1 = %5d;\n"
       "    bstr.field2 = %5d;\n"
       "    bstr.field3 = %5d;\n", 9, 488, 5602);
printf("=============================================\n\n");
printf("bstr = %u\n", bstr);

unsigned int *rab1 = &bstr;
s = *rab1;
```

```
    j = CHAR_BIT * sizeof(bstr) - 1;
    for(int i = 0; i < CHAR_BIT * sizeof(bstr); i++)
    {
        k = (s >> i) & 1;
        bin_str[j] = (k == 0) ? '0' : '1';
        j--;
    }
    bin_str[32] = '\x0';
    printf("=============================================\n");
    for(int i = 1; i < 33; i++)
    {
        printf("%c", bin_str[i-1]);
        if(i % 4 == 0) printf("%c", ' ');
    }
    printf("\n");
    printf("=============================================\n");
    itobs(9, bin_str);
    show_bstr(bin_str);
    printf("%5d\n", bstr.field1);
    itobs(488, bin_str);
    show_bstr(bin_str);
    printf("%5d\n", bstr.field2);
    itobs(5602, bin_str);
    show_bstr(bin_str);
    printf("%5d\n", bstr.field3);
    printf("=============================================\n");
    return 0;
}
char *itobs(int n, char *ps)
{
    int i;
    const static int size = CHAR_BIT * sizeof(unsigned int);
    // передбачається кодування ASCII або схоже
    for(i = size - 1; i >= 0; i--, n >>= 1)
        ps[i] = (01 & n) + '0';
    ps[size] = '\0';
    return ps;
}
// відображення двійкового рядка блоками по 4
void show_bstr(const char *str)
{
    int i = 0;
    while(str[i])        // поки не буде отриманий нульовий символ
    {
        putchar(str[i]);
        if(++i % 4 == 0 && str[i])
            putchar(' ');
    }
}
```

Продовження додатка 4

Результат виконання програми для роботи з бітовими полями структури наведено на рис. Д 4.1.

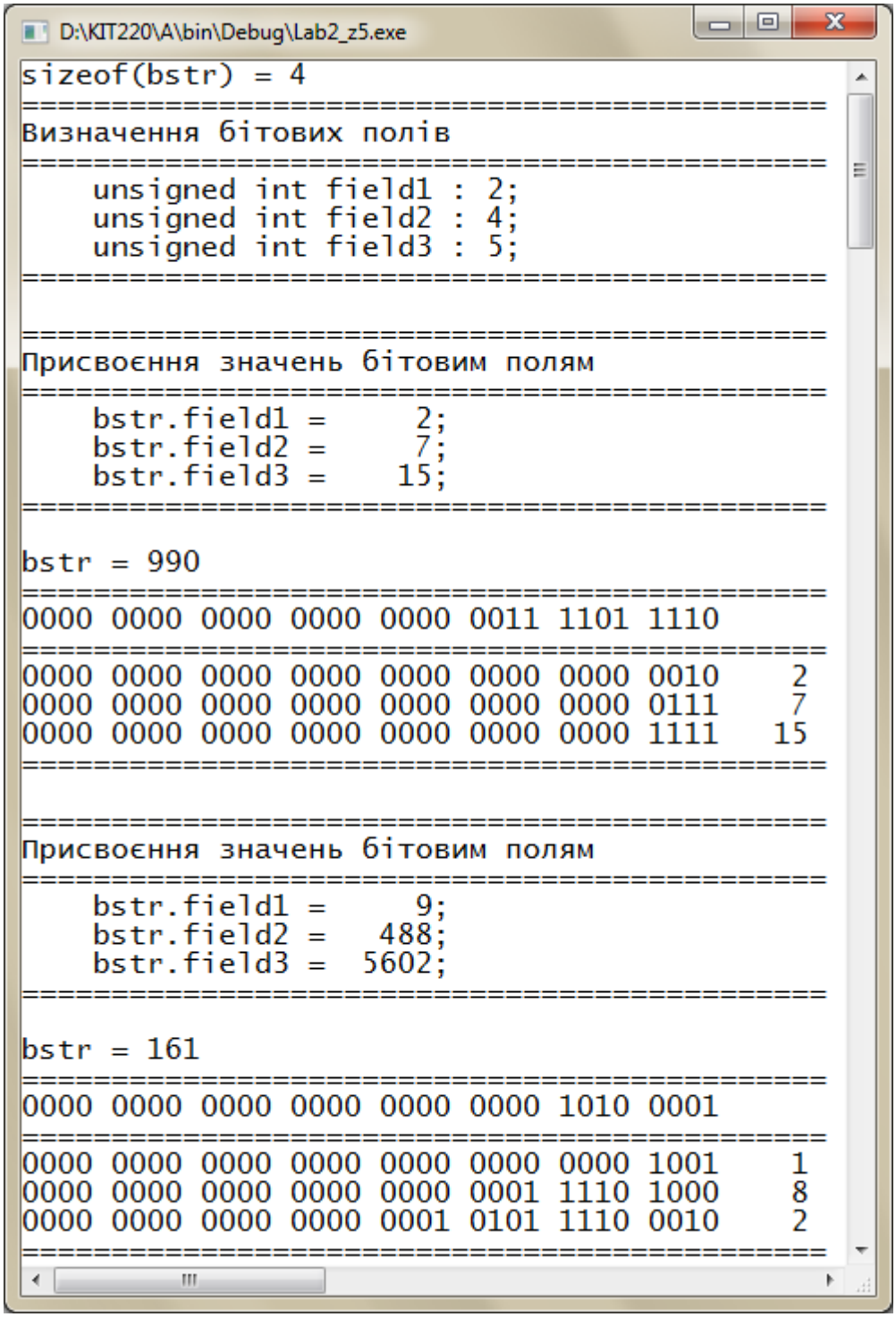


Рисунок Д 4.1 – Результат виконання програми для демонстрації роботи з бітовими полями структури

Додаток 5

Завдання до лабораторної роботи №3

Таблиця Д5.1 – Варіанти завдань

№ вар.	Операнд **a**	Операнд **b**	Встановити в 0 біти	Встановити в 1 біти	Виділити біти
1	0111001011000111	0000000011000101	8÷15	7÷11	3÷12
2	1101001101110010	0011011011101001	3÷9	4÷7	6÷9
3	0011001011010101	0000010011000101	6÷14	7÷11	8÷11
4	0011111011000101	0001000011000100	7÷11	3÷13	1÷10
5	0101011010010101	0010001111000101	4÷13	6÷12	4÷12
6	0011101011001101	0001001010100101	7÷11	8÷14	6÷14
7	0001011011001100	0001001011000101	6÷12	1÷7	7÷12
8	0101001011000101	0000001011010101	3÷12	5÷11	7÷11
9	0010010011000100	0001001011010101	1÷10	5÷11	6÷9
10	0001001011000101	0000011011000100	7÷10	9÷14	5÷11
11	0110001011010100	0001011111010101	8÷11	4÷13	3÷8
12	0011001010000111	0001101011010101	4÷12	2÷10	1÷9
13	0101001011110101	0001001011010101	5÷14	6÷11	4÷11
14	0100101010100101	0000101011010011	8÷11	8÷15	8÷14
15	0111001011110101	0001111011000101	6÷9	4÷12	8÷13
16	0110011011010100	0000101011000111	8÷13	5÷14	5÷14
17	0111011011010101	0001010011000101	5÷10	4÷9	6÷11
18	1101001101110010	0011011011101001	4÷8	7÷12	10÷13
19	0011001011010101	0000010011000101	2÷7	8÷14	5÷9
20	0011111011000101	0001000011000100	6÷11	7÷12	4÷7

Примітка: В тексті *C*–програми операнди треба задавати в десятковій системі числення.

В *C*–програмі	В табл. Д 5.1
`unsigned short a = 44758;`	`// a = 1010111011010110`
`unsigned short b = 46291;`	`// b = 1011010011010011`

Додаток 6

Результати виконання програми лабораторної роботи №3

Результат роботи програми для завдань №1–4 (рис. Д 6.1).

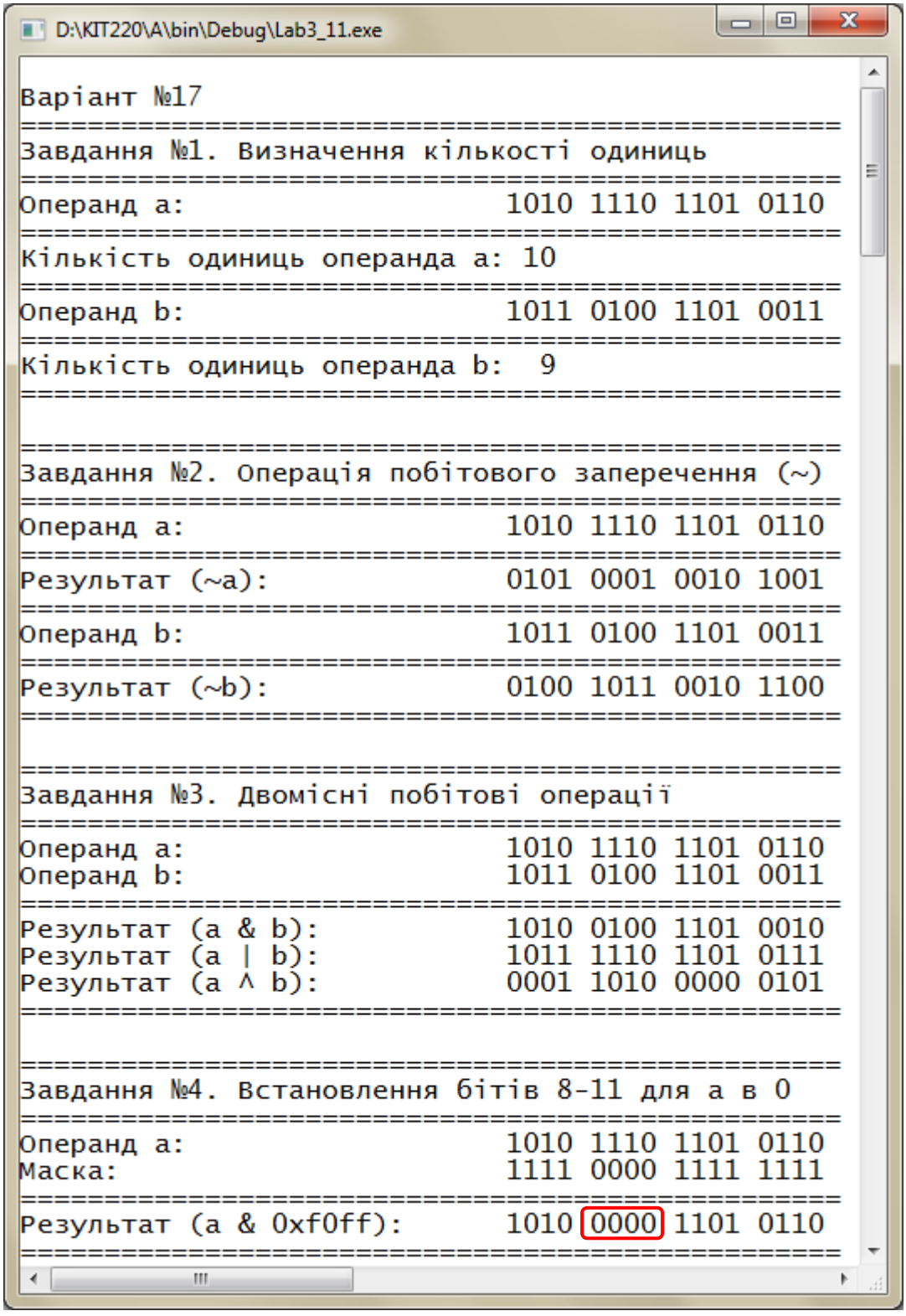


Рисунок Д 6.1 – Результат роботи програми для завдань №1–4

Результат роботи програми для завдань №5–7 (рис. Д 6.2).

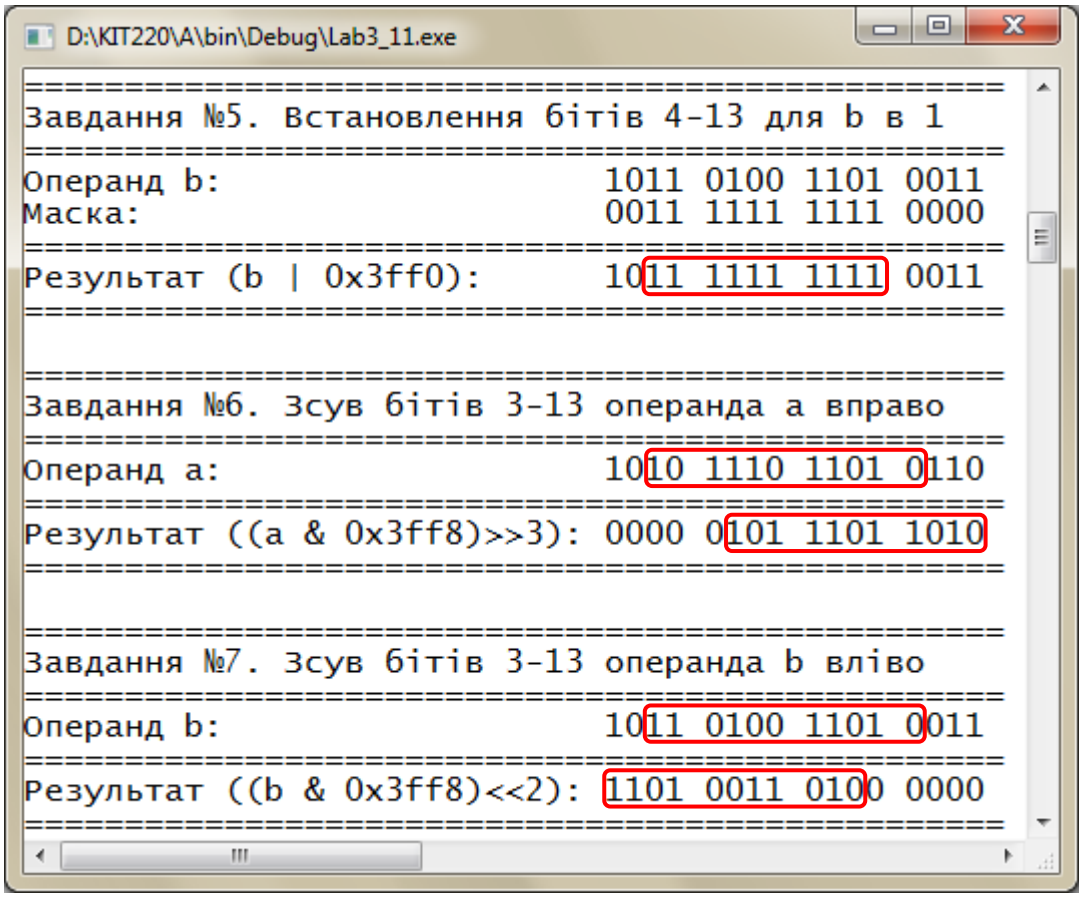


Рисунок Д 6.2 – Результат роботи програми для завдань №5–7

Додаток 7

Опції командного рядка компілятора gcc

На рис. Д 7.1 наведені опції командного рядка компілятора gcc.

Рисунок Д 7.1 – Опції для командного рядка компілятора *gcc*

Додаток 8

Варіанти завдання 2 лабораторної роботи №8

Визначення функцій та інтервалів, які необхідно позначити на графіку наведені в табл. Д 9.1.

Таблиця Д 9.1 – Визначення функцій та інтервалів

Номер варіанта	Функція	Інтервал $[a, b]$	Номер варіанта	Функція	Інтервал $[a, b]$
1	$f(x) = e^x$	[–1,1]	11	$f(x) = \ln(\|x\|)$	[–2,2]
2	$f(x) = e^{-x}$	[–2,2]	12	$f(x) = \|\sin(x)\|$	[–π/2,π/2]
3	$f(x) = \sin(x)$	[–π/2,π/2]	13	$f(x) = \ln(x^2 + 1)$	[–2,2]
4	$f(x) = \cos(x)$	[0,4π]	14	$f(x) = e^{-\|x\|}$	[–1,1]
5	$f(x) = tg(x)$	[–π/4,π/4]	15	$f(x) = \sqrt{x^3}$	[0,3]
6	$f(x) = 1/x$	[1,3]	16	$f(x) = \cos(x) - \sin(x)$	[–4,4]
7	$f(x) = 1/x^2$	[1,3]	17	$f(x) = e^x$	[–2,2]
8	$f(x) = \sqrt{x}$	[0,5]	18	$f(x) = e^{-x}$	[–3,3]
9	$f(x) = 1/(1 - x^2)$	[2,5]	19	$f(x) = \sin(x)$	[–2π,2π]
10	$f(x) = x^2 \ln(x)$	[1,5]	20	$f(x) = \cos(x)$	[–4π,4π]

Додаток 9

Прототипи функцій для роботи з графікою

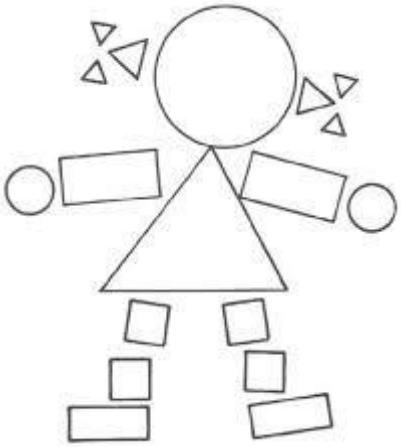
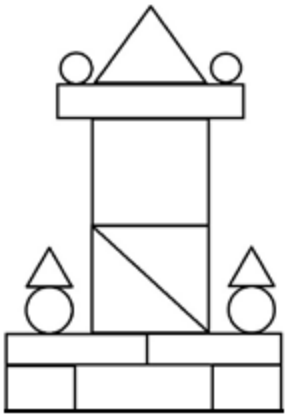
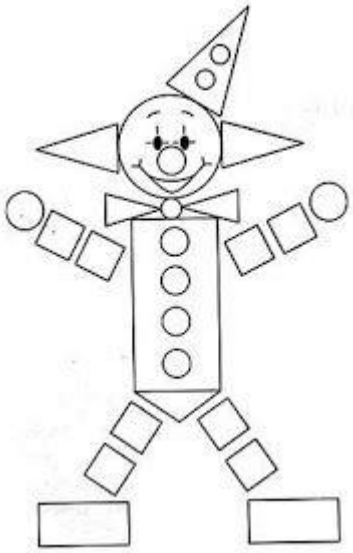
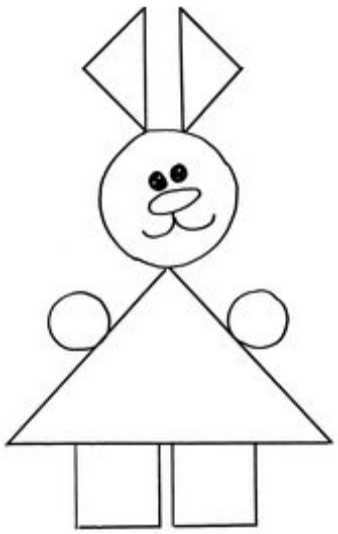
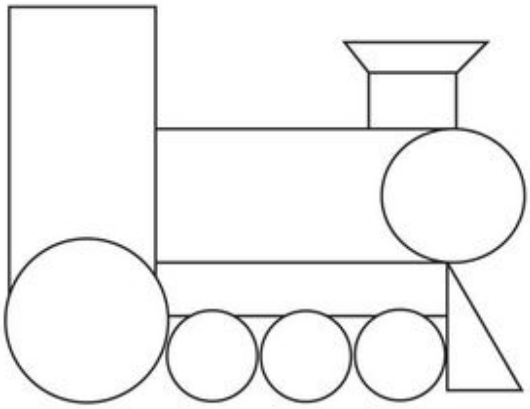
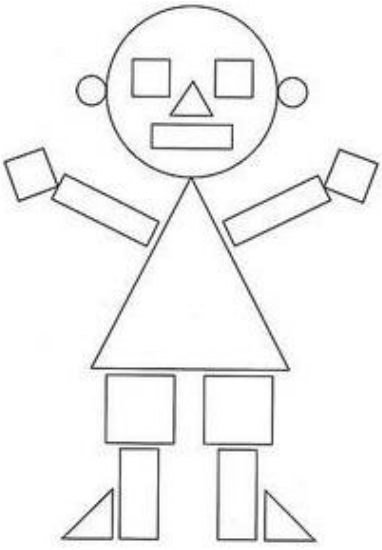
```
// дуга
void arc(int x, int y, int stangle, int endangle, int radius);
// зафарбований прямокутник
void bar(int left, int top, int right, int bottom);
// паралелепіпед
void bar3d(int left, int top, int right, int bottom,
           int depth, int topflag);
// круг
void circle(int x, int y, int radius);
// очистити екран
void cleardevice();
// багатокутник
void drawpoly(int n_points, int *points);
// еліпс
void ellipse(int x, int y, int stangle, int endangle,
             int xradius, int yradius);
```

Продовження додатка 9

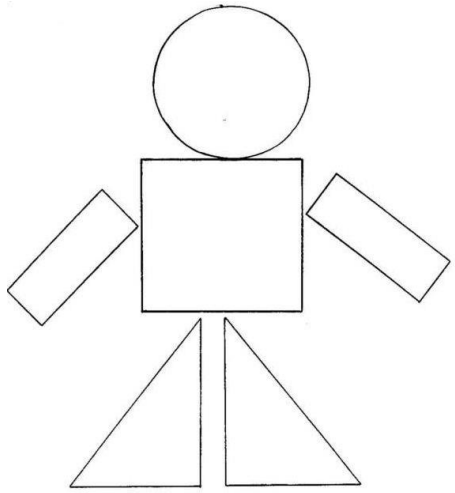
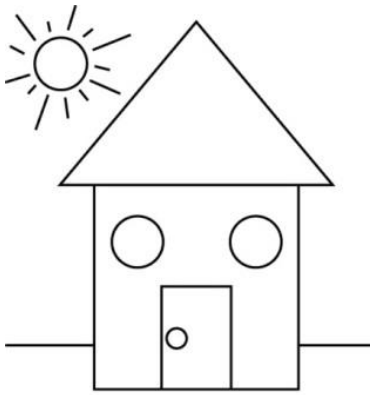
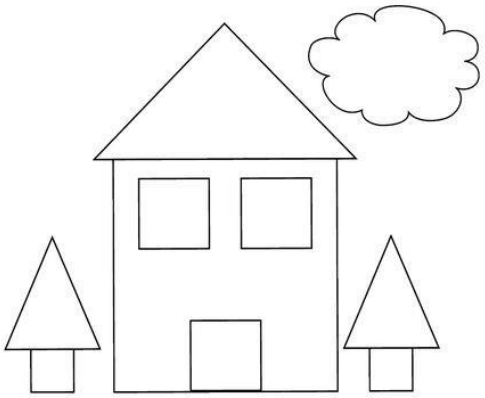
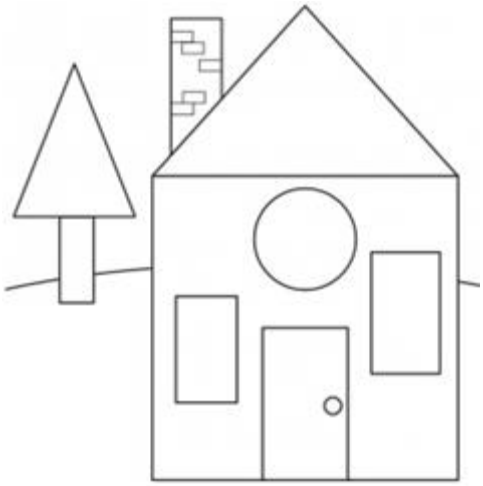
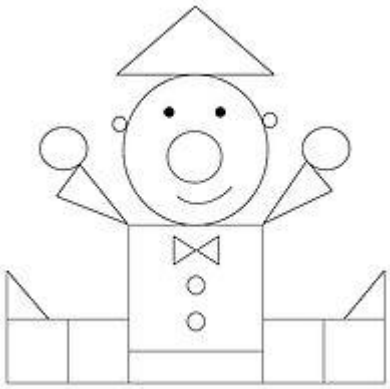
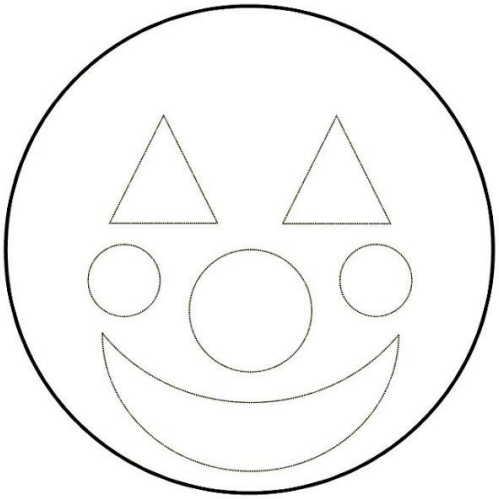
```
// зафарбований еліпс
void fillellipse(int x, int y, int xradius, int yradius);
// зафарбований багатокутник
void fillpoly(int n_points, int *points);
// зафарбувати область, що містить точку (x,y)
void floodfill(int x, int y, int border);
// рисувати лінію
void line(int x1, int y1, int x2, int y2);
// рисувати лінію від поточної точки до заданої точки
// на відстані dx по осі x і dy по осі y
void linerel(int dx, int dy);
// рисувати лінію від поточної точки до точки
// з координатами (x, y)
void lineto(int x, int y);
// рисувати сектор кругової діаграми з визначеним радіусом,
// що покриває кут від startangle до endangle
void pieslice(int x, int y, int startangle, int endangle,
              int radius);
// нарисувати точку
void putpixel(int x, int y, int color);
// прямокутник
void rectangle(int left, int top, int right, int bottom);
// сектор
void sector(int x, int y, int stangle, int endangle,
            int xradius, int yradius);
// переміщає поточну позицію на величини dx і dy
void moverel(int dx, int dy);
// переміщає поточну точку в точку з координатами (x, y)
void moveto(int x, int y);
// змінює колір фону на color
void setbkcolor(int color);
// встановлює поточний колір рисування
void setcolor(int color);
// встановлює визначений шаблон зафарбовування кольором color
void setfillpattern(char *upattern, int color);
// встановлює стиль та колір заливки
void setfillstyle(int pattern, int color);
// визначає зовнішній вигляд лінії
void setlinestyle(int linestyle, unsigned pattern, int thickness);
// створює нову область перегляду
void setviewport(int left, int top, int right, int bottom,
                 int clip);
// визначає зовнішній вигляд даних, що виводяться функціями
// line(), linerel(), lineto(), rectangle() і drawpoly()
void setwritemode(int mode);
```

Додаток 10

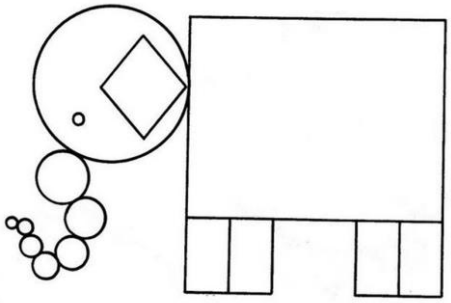
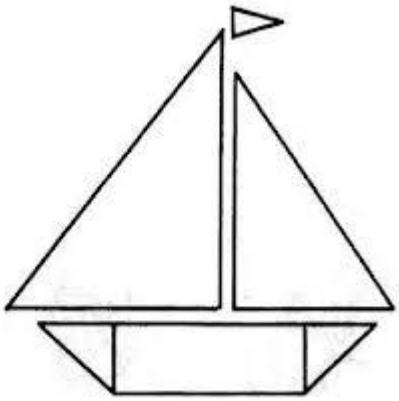
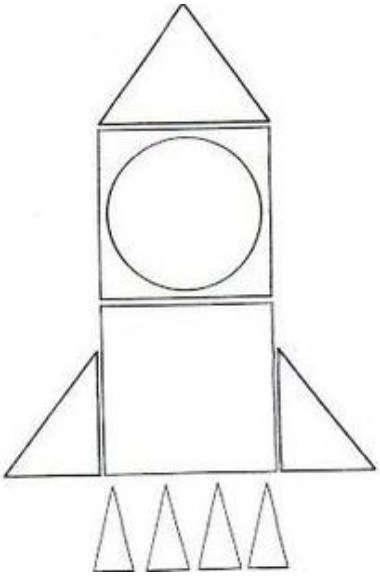
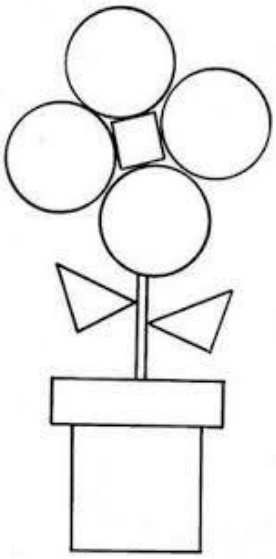
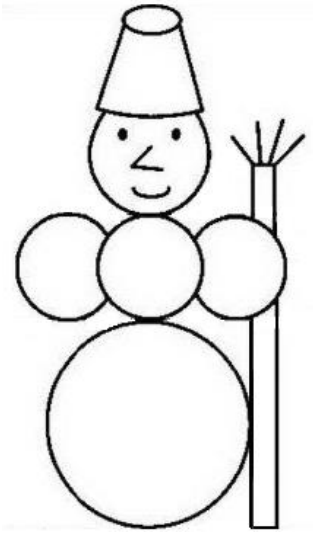
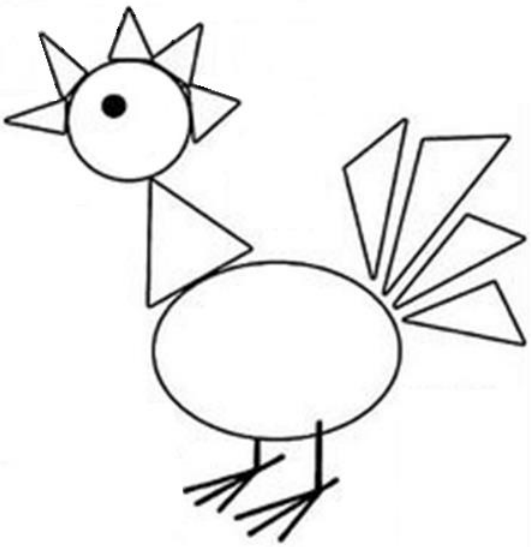
Варіанти завдання 4 лабораторної роботи №8

№	Зображення	№	Зображення
1		11	
2		12	
3		13	

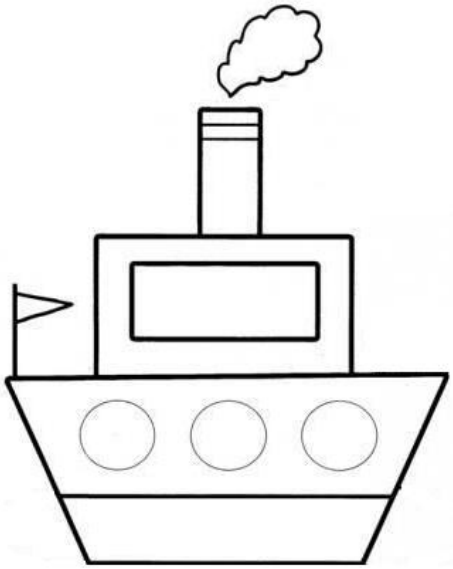
Продовження додатка 10

№	Зображення	№	Зображення
4		14	
5		15	
6		16	

Продовження додатка 10

№	Зображення	№	Зображення
7		17	
8		18	
9		19	

Продовження додатка 10

№	Зображення	№	Зображення
10		20	

Навчальне видання

Методичні вказівки
до виконання лабораторних робіт з дисципліни
«Програмування»
для студентів денної та заочної форм навчання за спеціальністю
123 «Комп’ютерна інженерія»
Частина 2

Укладачі: ПОРОШИН Сергій Михайлович
СТАТКУС Андрій Віталійович
КОРОЛЬОВА Яна Юріївна
ОНИЩЕНКО Валерій Валентинович

Відповідальний за випуск проф. Порошин С. М.
Роботу рекомендував до друку проф. Заполовський М. Й.

В авторській редакції

План 2021 р., поз. 260

Підписано до друку 6.12.2021 р. Формат 60 × 84 1/16. Папір офсетний.
RISO-друк. Гарнітура Тайм
Ум. друк. арк. 10.7
Наклад 50 прим. Замовлення № . Ціна договірна

Видавничий центр НТУ «ХПІ».
Свідоцтво про державну реєстрацію ДК № 5478 від 21.08.2017 р.
61002, Харків, вул. Кирпичова, 2

Видавництво та друк ФО-П Єфименко С.А.
61166, Україна, м. Харків, вул. Коломенська, 27
Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру видавців – виготовлювачів і розповсюджувачів видавничої продукції
ДК № 6869 від 08.08.2019 р.