ТЕМА №2. КЛАСИ ЗБЕРІГАННЯ ТА КЕРУВАННЯ ПАМ’ЯТТЮ

Лекція №5. Класи зберігання

1 Класи зберігання даних.

2 Область видимості.

3 Одиниці та файли трансляції.

4 Зв’язування.

5 Формальні та неформальні терміни.

6 Тривалість зберігання.

7 Автоматичні змінні.

8 Блоки без фігурних дужок.

9 Ініціалізація автоматичних змінних.

10 Регістрові змінні.

11 Статичні змінні з областю видимості в межах блоку.

12 Статичні змінні з зовнішнім зв’язуванням.

13 Ініціалізація зовнішніх змінних.

14 Використання зовнішньої змінної.

15 Зовнішні імена.

16 Визначення та оголошення.

17 Статичні змінні з внутрішнім зв’язуванням.

18 Використання декількох файлів.

Однією з переваг мови **C** є можливість керування витонченими аспектами програми. Система керування пам'яттю в **C** є ілюстрацією такого керування, дозволяючи визначати, яким функціям відомі ті або інші змінні та наскільки довго змінна існує в програмі. Використання місця зберігання в пам'яті є ще одним елементом проектного рішення, яке покладається в основу програми.

1 Класи зберігання даних

Для зберігання даних у пам'яті мова **C** пропонує п'ять різних моделей, або **класів зберігання**. Щоб зрозуміти доступні варіанти, корисно спочатку вивчити декілька концепцій і термінів.

В кожній програмі дані зберігаються в пам'яті. Для цього існує **апаратний аспект** – будь-яке збережене значення знаходиться в фізичній пам'яті. В літературі по **C** для опису такої ділянки пам'яті застосовується термін **об'єкт**. Об'єкт може зберігати одне або декілька значень. В певний момент об'єкт може поки не містити збережене значення, але він буде мати правильний розмір для розміщення потрібного значення.

Існує також і **програмний аспект** – програмі потрібен будь-який спосіб доступу до об'єкту. Цього можна досягти, наприклад, шляхом оголошення змінної:

```
int entity = 3;
```

Показане оголошення призводить до створення ідентифікатора на ім'я **entity**. Ідентифікатор являє собою ім'я, в даному випадку таке, яке може застосовуватися для позначення вмісту окремого об'єкту. Ідентифікатори відповідають домовленостям про іменування змінних, які були розглянуті раніше. В цьому випадку ідентифікатор **entity** відображає спосіб, яким програмне забезпечення (програма на **C**) вказує об'єкт, що зберігається в апаратній пам'яті. Таке оголошення також надає значення для зберігання в об'єкті.

Ім'я змінної – не єдиний метод позначення об'єкту. Наприклад, погляньте на наступні оголошення:

```
int *pt = &entity;
int ranks[10];
```

В першому випадку **pt** являє собою ідентифікатор. Він позначає об'єкт, який містить адресу. Вираз ***pt** – не ідентифікатор, оскільки він не є іменем. Проте, він вказує на об'єкт, в даній ситуації – на той самий об'єкт, що й **entity**.

В загальному випадку, вираз, який означає об'єкт, називається **l**-значенням. Таким чином, **entity** – це ідентифікатор, що являє собою **l**-значення, а ***pt** – вираз, що є **l**-значенням. При тих самих оголошеннях вираз **ranks+2*entity** – не ідентифікатор (не ім'я) і не **l**-значення (не вказує на вміст комірки пам'яті). Але вираз ***(ranks+2*entity)** є **l**-значенням, тому що воно вказує на значення певної комірки пам'яті (сьомого елемента масиву **ranks**). До речі, оголошення **ranks** призводить до створення об'єкта, який здатен зберігати **10** значень типу **int**, і кожен елемент масиву також являє собою об'єкт.

Якщо **l**-значення можна використовувати для зміни значення всередині об'єкта, ми маємо справу з **l**-значенням, що модифікується. Тепер розглянемо наступне оголошення:

```
const char *pc = "Це рядковий літерал!";
```

Воно призводить до того, що програма зберігає в пам'яті значення рядкового літерала, і цей масив символьних значень є об'єктом. Кожен символ в масиві також являє собою об'єкт, оскільки до нього можна звертатися індивідуально. Оголошення також створює об'єкт, який має ідентифікатор **pc** і зберігає адресу даного рядка. Ідентифікатор **pc** – це **l**-значення, що модифікується, оскільки його можна заново встановлювати для посилання на інші рядки. Ключове слово **const** запобігає зміні вмісту рядка, на який вказує **pc**, але не зміні того, на який рядок він вказує. Таким чином, вираз ***pc**, означає об'єкт даних з символом **'ц'**, є **l**-значенням, але не **l**-значенням, що модифікується. Аналогічно, сам рядковий літерал, вказуючи на об'єкт, який містить символьний рядок, являє собою **l**-значення, що не допускає модифікації.

Об'єкт можна описати в термінах його тривалості зберігання, яка вказує, наскільки довго він залишається в пам'яті. Ідентифікатор, що застосовується для

доступу до цього об’єкту, може бути описаний за допомогою його області видимості та зв’язування, які разом вказують, в якій частині програми цей ідентифікатор дозволено використовувати. Різні класи зберігання пропонують різні поєднання області видимості, зв’язування та тривалості зберігання. Допускається існування ідентифікаторів, які спільно використовуються в декількох файлах вхідного коду, ідентифікаторів, які можуть застосовуватися в будь-яких функціях всередині одного файлу, ідентифікаторів, які використовуються тільки всередині окремої функції, і навіть ідентифікаторів, що застосовуються лише в певному розділі функції. Одні об’єкти можуть існувати протягом часу життя цілої програми, а інші – тільки протягом часу виконання функції, яка їх містить. В паралельному програмуванні можна мати об’єкти, які існують протягом часу виконання окремого потоку. Можна також зберігати дані в пам’яті, яка явно виділяється та звільняється шляхом викликів функцій. Давайте поглянемо, що означають терміни область видимості, зв’язування та тривалість зберігання. Після цього приступимо до вивчення конкретних класів зберігання.

2 Область видимості

Область видимості описує ділянку або ділянки програми, де можна звертатися до ідентифікатора. Змінна в **C** має одну з наступних областей видимості:

- в межах блоку;
- в межах функції;
- в межах прототипу функції;
- в межах файлу.

У прикладах програм, що були розглянуті до сих пір, для змінних використовувалася в основному область видимості на рівні блоку. Як ви пам’ятаєте, блок – це частина коду, що розміщується між фігурними дужками. Наприклад, блоком є усе тіло функції. Будь-який складений оператор всередині функції також вважається блоком. Змінна, що визначається у блоці, має область видимості в межах блоку, і вона є видимою від місця, де вона визначена, до кінця

блоку, що містить це визначення. Крім того, формальні параметри функції, хоча вони з'являються до відкривальної фігурної дужки функції, також мають область видимості в межах блоку та належать блоку, що містить тіло функції. Таким чином, усі локальні змінні, які застосовувалися до сих пір, включаючи формальні параметри функцій, мали область видимості в межах блоку. Відповідно, змінні **cleo** і **patrick** в наведеному нижче коді мають область видимості в межах блоку, що простягається до закривальної фігурної дужки:

```
double blocky(double cleo)
{
     double patrick = 0.0;
     ...
     return patrick;
}
```

Змінні, що оголошені у внутрішньому блоці, отримують область видимості, яка обмежена тільки цим блоком:

```
double blocky(double cleo)
{
     double patrick = 0.0;
     int i;

     for(i = 0; i < 10; i++)
     {
          double q = cleo*i; // початок області видимості для q
          ...
          patrick *= q;
     } // кінець області видимості для q
     ...
     return patrick;
}
```

В цьому прикладі область видимості **q** обмежена внутрішнім блоком, і доступ до **q** може мати тільки код всередині цього блоку.

За традицією змінні з областю видимості в межах блоку повинні оголошуватися на початку блоку. В стандарті **C99** ця вимога була послаблена, і змінні дозволено оголошувати в будь-якому місці блоку. Одна з нових можливостей пов'язана з оголошенням всередині керуючого розділу циклу **for**. Тепер можна зробити таким чином:

```
for(int i = 0; i < 10; i++)
     printf("Можливість C99: i = %d", i);
```

Як частина цієї нової можливості, стандарт **C99** розширив концепцію блоку шляхом включення до неї коду, що керується циклами **for**, **while**, **do...while** або оператором **if**, навіть якщо фігурні дужки при цьому не використовуються. Таким чином, у попередньому циклі **for** змінна **i** вважається частиною блоку циклу **for**. Відповідно, її область видимості обмежена циклом **for**. Після того, як виконання покине цикл **for**, ця змінна **i** більше не буде помітною для програми.

Область видимості в межах функції застосовуються тільки до міток, що використовуються з операторами **goto**. Це означає, що коли мітка вперше з'являється у внутрішньому блоці функції, її область видимості простирається на всю функцію. Якщо б можна було використовувати одну й ту ж саму мітку всередині двох окремих блоків, виникла б плутанина, тому область видимості в межах функції для міток запобігає виникненню такої ситуації.

Область видимості в межах прототипу функції застосовується до імен змінних, що використовуються в прототипах функцій, як в наступному вигляді:

```
int mighty(int mouse, double large);
```

Область видимості в межах прототипу функції поширюється від місця визначення змінної до кінця оголошення прототипу. Це означає, що при обробці аргументу прототипу функції компілятор цікавить тільки тип аргументу. Якщо вказані імена, то зазвичай вони не грають жодної ролі та не обов'язково повинні збігатися з іменами, які застосовуються у визначенні функції. Імена грають невелику роль у випадку параметрів, що мають типи масивів змінної довжини:

```
void use_a_VLA(int n, int m, int mas[n][m]);
```

При використанні імен в дужках треба пам'ятати, що це повинні бути імена, які були оголошені раніше в прототипі.

Змінна, визначення якої знаходиться за рамками будь-якої функції, має **область видимості в межах файлу**. Змінна, що має область видимості в межах файлу, є видимою від місця її визначення і до кінця файлу, який містить її визначення.

Погляньте на показаний нижче приклад:

```
#include <stdio.h>

int units = 0;              // змінна з областю видимості
                            // в межах файлу
void critic(void);

int main(void)
{
      ...
}

void critic(void)
{
      ...
}
```

Тут змінна **units** має область видимості в межах файлу та може використовуватися і в функції **main()**, і в функції **critic()**. Оскільки змінні з областю видимості в межах файлу можуть використовуватися в більш ніж одній функції, вони ще називаються **глобальними змінними**.

3 Одиниці та файли трансляції

То, що ви розглядаєте як декілька файлів, для компілятора може виглядати як єдиний файл. Припустимо для прикладу, і це трапляється досить часто, що ви включаєте один або декілька файлів заголовку (з розширенням **.h**) до файлу вхідного коду (з розширенням **.c**). В свою чергу, файл заголовку може містити інші файли заголовку. У підсумку може бути задіяно багато фізичних файлів. Однак препроцесор **C** по суті замінює директиву **#include** вмістом файлу заголовку. Таким чином, компілятор бачить єдиний файл, який містить інформацію з вашого файлу вхідного коду та усіх файлів заголовку. Такий файл називається **одиницею трансляції**. Коли ми описуємо змінну як ту, що має область видимості в межах файлу, насправді вона буде видимою цілій одиниці трансляції. Якщо ваша програма складається з декількох файлів вхідного коду, то вона буде нараховувати й декілька одиниць трансляції, кожна з яких відповідає файлу вхідного коду, і файлам, що включаються до нього.

4 Зв'язування

Змінна в **C** має одне з наступних зв'язувань:

- зовнішнє зв'язування;
- внутрішнє зв'язування;
- відсутність зв'язування.

Змінні з областю видимості в межах блоку, функції або прототипу функції не мають зв'язування. Це означає, що вони є закритими для блоку, функції або прототипу, в якому визначені. Змінна з областю видимості в межах файлу може мати або внутрішнє, або зовнішнє зв'язування. Змінна з зовнішнім зв'язуванням може застосовуватися у будь-якому місці багатофайлової програми, а змінна з внутрішнім зв'язуванням – де завгодно в одиниці трансляції.

5 Формальні та неформальні терміни

В стандарті **C** для опису області видимості, що обмежена однією одиницею трансляції (файл вхідного коду плюс файли заголовку, що підключаються до нього), використовується формулювання «область видимості в межах файлу з внутрішнім зв'язуванням», а для опису області видимості, яка (у всякому випадку, потенційно) поширюється на інші одиниці трансляції – формулювання «область видимості в межах файлу з зовнішнім зв'язуванням». Але у програмістів не завжди є час і терпіння застосовувати такі терміни. Поширення отримали скорочення «область видимості в межах файлу» для «області видимості в межах файлу з внутрішнім зв'язуванням» і «глобальна область видимості» або «область видимості в межах програми» для «області видимості в межах файлу з зовнішнім зв'язуванням».

Як же тоді з'ясувати, внутрішнє або зовнішнє зв'язування має змінна з областю видимості в межах файлу? Ви повинні подивитися, чи використовується у зовнішньому визначенні специфікатор класу зберігання **static**:

```
int giants =5;              // область видимості в межах файлу,
                            // зовнішнє зв'язування
static int dodgers = 3;     // область видимості в межах файлу,
                            // внутрішнє зв'язування
```

```
int main(void)
{
    ...
}
    ...
```

Змінна **giants** може застосовуватися в інших файлах, які представляють собою складені частини тієї ж самої програми. Змінна **dodgers** є закритою для даного конкретного файлу, але може використовуватися будь-якою функцією у цьому файлі.

6 Тривалість зберігання

Область видимості та зв'язування описують видимість ідентифікаторів. Тривалість зберігання характеризує постійність об'єктів, що є доступними за допомогою цих ідентифікаторів. Об'єкт в **C** має одну з наступних чотирьох тривалостей зберігання:

- статичну;
- потокову;
- автоматичну;
- виділену.

Якщо об'єкт має **статичну тривалість зберігання**, він існує протягом часу виконання програми. Змінні з областю видимості в межах файлу мають статичну тривалість зберігання. Зверніть увагу, що для змінних з областю видимості в межах файлу ключове слово **static** вказує тип зв'язування, а не тривалість зберігання. Змінна з областю видимості в межах файлу, що оголошена з застосуванням **static**, має внутрішнє зв'язування, але усі змінні з областю видимості в межах файлу, що мають внутрішнє або зовнішнє зв'язування, мають статичну тривалість зберігання.

Потокова тривалість зберігання має місце при паралельному програмуванні, коли виконання програми може бути поділено на декілька потоків. Об'єкт з потоковою тривалістю зберігання існує з моменту його оголошення і до завершення потоку. Такий об'єкт створюється, коли оголошення, яке в іншому випадку призвело б до створення об'єкту з областю видимості в

межах файлу, модифіковане за допомогою ключового слова **_Thread_local**. Коли змінна оголошена з таким специфікатором, кожен потік отримує власну закриту копію цієї змінної.

Змінні з областю видимості в межах блоку зазвичай мають автоматичну тривалість зберігання. Пам'ять для цих змінних виділяється, коли потік керування входить до блоку, де вони визначені, та звільняється, коли потік керування покидає цей блок. Ідея полягає в тому, що пам'ять, яка використовується для автоматичних змінних, є робочим простором або тимчасовою пам'яттю, яка може застосовуватися багатократно. Наприклад, після завершення виклику функції, пам'ять, яку функція використовувала для своїх змінних, може бути задіяна під час виклику наступної функції.

Масиви змінної довжини демонструють невеличке виключення в тому, що вони існують від місця свого оголошення і до кінця блоку, а не від початку блоку і до свого кінця.

Локальні змінні, які ми застосовували до сих пір, потрапляють до категорії автоматичної тривалості зберігання. Наприклад, в наступному коді змінні **number** і **index** з'являються кожного разу, коли функція **bore()** викликається, та зникають після її завершення:

```
void bore(int number)
{
     int index;
     for(index = 0; index < number; index++)
          puts("Виконання звичної роботи.\n");
     return 0;
}
```

Проте, змінна може мати область видимості в межах блоку, але статичну тривалість зберігання. Щоб створити таку змінну, треба оголосити її всередині блоку та додати до оголошення ключового слова **static**:

```
void more(int number)
{
     int index;
     static int ct = 0;
     ...
     return 0;
}
```

Тут змінна `ct` зберігається в статичній пам'яті. Вона існує з моменту завантаження програми в пам'ять і аж до завершення виконання програми. Але область видимості `ct` обмежена блоком функції `more()`. Тільки під час виконання цієї функції програма може використовувати змінну `ct` для доступу до об'єкту, який вона позначує.

Область видимості, зв'язування та тривалість зберігання використовується в `C` з метою визначення декількох схем зберігання для змінних.

Існує п'ять класів зберігання:

- автоматичний;
- регістровий;
- статичний з областю видимості в межах блоку;
- статичний з зовнішнім зв'язуванням;
- статичний з внутрішнім зв'язуванням.

Можливі комбінації представлені в табл. 1.1.

Таблиця 1.1 – П'ять класів зберігання

Клас зберігання	Тривалість зберігання	Область видимості	Зв'язування	Оголошення
Автоматичний	Автоматична	В межах блоку	Ні	В блоці
Регістровий	Автоматична	В межах блоку	Ні	В блоці з зазначенням ключового слова `register`
Статичний з зовнішнім зв'язуванням	Статична	В межах файлу	Зовнішнє	За рамками усіх функцій
Статичний з внутрішнім зв'язуванням	Статична	В межах файлу	Внутрішнє	За рамками усіх функцій з зазначенням ключового слова `static`
Статичний без зв'язування	Статична	В межах файлу	Ні	В блоці з зазначенням ключового слова `static`

7 Автоматичні змінні

Змінна, що належить до автоматичного класу зберігання, має автоматичну тривалість зберігання, область видимості в межах блоку і не має зв'язування. За замовчуванням будь-яка змінна, яка оголошена в блоці або в заголовку функції,

належить до автоматичного класу зберігання. Однак ви можете сформулювати свої наміри, явно вказавши ключове слово **auto**:

```
int main(void)
{
     auto int plox;
     ...
}
```

Це можна зробити, скажімо, для документування того факту, що ви навмисно перевизначаєте зовнішню змінну, або для підкреслення важливості того, що клас зберігання змінної не повинен змінюватися. Ключове слово **auto** називається специфікатором класу зберігання. В **C++** ключове слово **auto** призначено для зовсім іншої цілі, тому просто не застосовуйте **auto** в якості специфікатора класу зберігання, щоб домогтися більшої сумісності між **C** і **C++**.

Область видимості в межах блоку та відсутність зв'язування припускає, що доступ до цієї змінної по імені може здійснюватися тільки в блоці, де змінна визначена. Звичайно, за допомогою аргументів значення та адресу змінної можна передати до іншої функції, але це будуть вже непрямі відомості. В іншій функції може використовуватися змінна з тим самим іменем, але це буде незалежна змінна, що зберігається в іншій комірці пам'яті.

Автоматична тривалість зберігання означає, що змінна починає своє існування, коли потік керування входить до блоку, який містить оголошення цієї змінної. Після того як потік керування залишить блок, автоматична змінна зникає. Комірка пам'яті, яку вона займала, може застосовуватися для будь-чого іншого, хоча й необов'язково.

Давайте більш уважно поглянемо на вкладені блоки. Змінна відома тільки в блоці, в якому вона оголошена, і в будь-яких блоках, що розміщені всередині цього блока:

```
int loop(int n)
{
     int m;      // m знаходиться в області видимості
     scanf("%d", &m);
     {
          int i;     // m та i знаходяться в області видимості
          for(i = m; i < n; i++)
               puts("i є локальною змінною в субблоці \n");
```

```
        }
        return m;       // m в області видимості,
                        // і зникла
    }
```

В даному коді змінна **i** є видимою тільки у внутрішніх дужках. Якщо ви спробуєте скористатися цією змінною до або після внутрішнього блоку, компілятор повідомить про помилку. Зазвичай такий прийом при проектуванні програм не застосовується. Проте, часом зручно визначати змінну в субблоці, якщо вона не використовується в інших місцях. В цьому випадку можна документувати призначення змінної близько до місця її застосування.

Крім того, змінна не буде залишатися невикористаною, дарма займаючи місце, коли вона більше не потрібна. Оскільки змінні **n** і **m** визначені в заголовку функції та у зовнішньому блоці, вони находяться в області видимості всієї функції та існують аж до її завершення.

Що станеться, коли ви оголосите у внутрішньому блоці змінну, яка має таке ж саме ім’я, як і змінна у зовнішньому блоці? Тоді ім’я, що визначено всередині блоку, відповідає змінній, яка застосовується в цьому блоці. Ми говоримо, що ім’я приховує зовнішнє визначення. Однак коли потік керування залишає внутрішній блок, зовнішня змінна повертається до області видимості. Ці та інші аспекти проілюстровані в наступній програмі. Текст програми має наступний вид:

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

int main(void)
{
    int x = 30;         // перша змінна x

    SetConsoleOutputCP(1251);

    printf("x у зовнішньому блоці:  %4d за адресою %p\n", x, &x);
    {
        int x = 77;     // друга змінна x, що приховує першу x
        printf("x у внутрішньому блоці: %4d за адресою %p\n",
                x, &x);
    }
    printf("x у зовнішньому блоці:  %4d за адресою %p\n", x, &x);
    while(x++ < 33)     // перша змінна x
    {
        int x = 100;    // третя змінна x, що зв'язує першу x
```

```
        x++;
        printf("x у циклі while:         %4d за адресою %p\n",
                x, &x);
    }
    printf("x у зовнішньому блоці:  %4d за адресою %p\n", x, &x);
    return 0;
}
```

Результат роботи програми наведено на рис. 1.1.

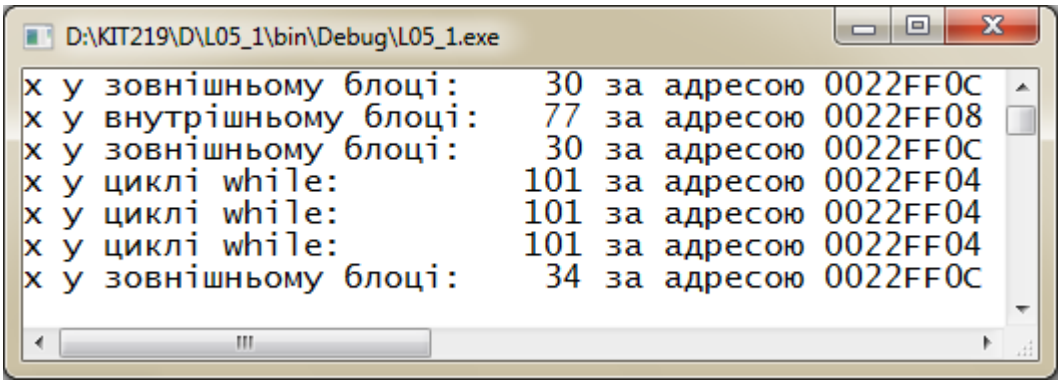

Рисунок 1.1 – Результат роботи програми для відстеження змінних у блоках

Спочатку програма створює змінну **x** зі значенням **30**, як показує перший оператор **printf()**. Потім вона визначає нову змінну **x** зі значенням **77**, про що повідомляє другий оператор **printf()**. Це нова змінна, що приховує першу змінну **x**, значення та адреса якої знову виводяться третім оператором **printf()**. Даний оператор знаходиться після першого внутрішнього блоку та відображає початкове значення **x**, демонструючи тим самим, що ця змінна нікуди не зникла й не змінювалася.

Імовірно, найбільш цікавою частиною програми є цикл **while**. В умові перевірки циклу **while** задіяна початкова змінна **x**:

```
while(x++ < 33)
```

Однак всередині циклу програма бачить третю змінну **x**, тобто ту, яка визначена в рамках блока циклу **while**. Таким чином, коли в тілі циклу використовується вираз **x++**, в ньому беруть участь нова змінна **x**, значення якої інкрементується до **101** і потім відображається. По завершенні кожної ітерації циклу ця нова змінна **x** зникає.

Далі в умові перевірки циклу застосовується та інкрементується початкова змінна **x**, знову відбувається вхід до блоку циклу, і знову створюється нова змінна **x**. В цьому прикладі змінна **x** створюється та знищується три рази.

Зверніть увагу, що для припинення виконання цикл повинен інкрементувати **x** в умові перевірки, тобто інкрементування **x** в тілі циклу призводить до збільшення значення іншої змінної **x**, а не тієї, яка задіяна в умові перевірки.

Хоча конкретний компілятор не використовує повторно комірку пам'яті змінної **x** внутрішнього блоку для версії **x** з циклу **while**, деякі компілятори це роблять.

Призначення цього прикладу зовсім не в тому, щоб заохочувати написання коду в такому стилі. Він служить лише ілюстрацією того, що відбувається, коли ви визначаєте змінні всередині блоку.

8 Блоки без фігурних дужок

Згадана раніше можливість стандарту **C99** полягає в тому, що оператори, які є частиною циклу або оператора **if**, кваліфікуються як блок, навіть якщо фігурні дужки **({})** не вказані. Висловлюючись точніше, повний цикл – це субблок блоку, в якому він розміщений, а тіло циклу – субблок блоку повного циклу. Аналогічно, оператор **if** являє собою блок, а зв'язаний з ним оператор – субблок оператора **if**. Описані правила впливають на те, де ви можете оголошувати змінну, і на область видимості цієї змінної.

В наступній програмі показано, як це працює в циклі **for**. Текст програми має наступний вигляд:

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

int main(void)
{
    int n = 8;

    SetConsoleOutputCP(1251);

    printf("Спочатку        n = %d за адресою %p\n", n, &n);
```

```
    for(int n = 1; n < 3; n++)
        printf("      цикл 1:   n = %d за адресою %p\n", n, &n);
    printf("  Після циклу 1 n = %d за адресою %p\n", n, &n);
    for(int n = 1; n < 3; n++)
    {
        printf(" індекс циклу 2 n = %d за адресою %p\n", n, &n);
        int n = 6;
        printf("      цикл 2:   n = %d за адресою %p\n", n, &n);
        n++;
    }
    printf("  Після циклу 2 n = %d за адресою %p\n", n, &n);
    return 0;
}
```

Результат роботи програми наведено на рис. 1.2.

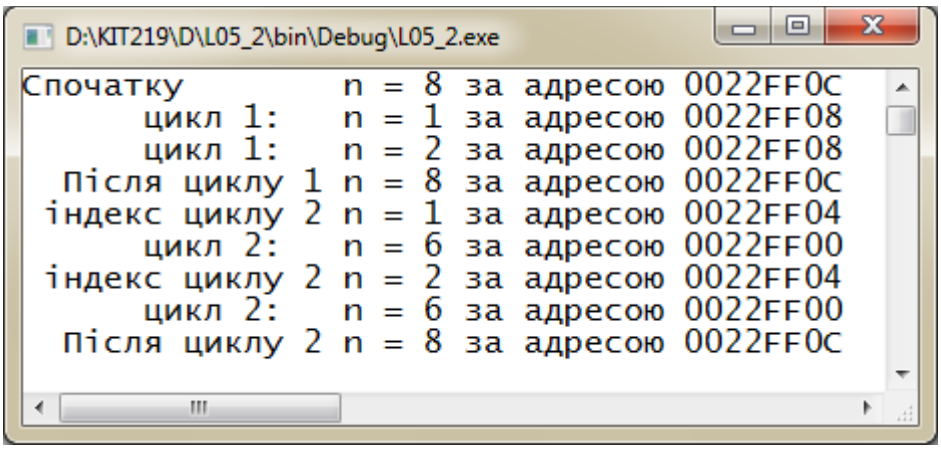

Рисунок 1.2 – Результат роботи програми, яка демонструє використання змінних в блоках

Змінна **n**, яка оголошена в круглих дужка першого циклу **for**, має область видимості до кінця циклу і приховує початкову змінну **n**. Але після того, як керування залишає цикл, початкова змінна **n** повертається до області видимості.

В другому циклі **for** змінна **n**, оголошена як індекс циклу, приховує початкову змінну **n**. Потім змінна **n**, яка оголошена всередині тіла циклу, приховує індекс циклу **n**. Як тільки програма завершить виконання тіла, змінна **n**, що оголошена в тілі, зникає, а в перевірці циклу бере участь індекс **n**. Коли завершиться виконання всього циклу, в області видимості з'являється початкова змінна **n**.

І знову відзначимо, що немає жодної потреби багатократно застосовувати, одне й те ж саме ім’я для змінної, але ви повинні знати, що відбудеться, коли ви все ж таки приймете рішення зробити таким чином.

9 Ініціалізація автоматичних змінних

Автоматичні змінні не ініціалізуються до тих пір, поки ви не зробите це явно. Погляньте на наступні оголошення:

```
int main(void)
{
      int repid;
      int tents = 5;
      ...
}
```

Змінна **tents** ініціалізується значенням **5**, але **repid** отримує значення, яке раніше знаходилось в області пам’яті, що виділяється під цю змінну. Неможна розраховувати, на те, що цим значенням буде **0**. Ви можете ініціалізувати автоматичну змінну неконстантним виразом за умови, що усі задіяні в ньому змінні були визначені раніше:

```
int main(void)
{
      int ruth = 1;
      int rance = 8 * ruth; // використовується раніше визначена
                            // змінна
      ...
}
```

10 Регістрові змінні

Змінні зазвичай зберігаються в пам’яті комп’ютера. За сприятливим збігом обставин регістрові змінні зберігаються в регістрах центрального процесора, або, в загальному випадку, в найшвидшій пам’яті, що забезпечує доступ і маніпулювання ними з меншими витратами часу, ніж для звичайних змінних. Оскільки регістрова змінна може знаходитися в регістрі, або в пам’яті, отримати адресу такої змінної не вдасться. В більшості інших випадків регістрові змінні нічим не відрізняються від автоматичних змінних. Тобто вони мають область

видимості в межах блоку, не мають зв'язування та мають автоматичну тривалість зберігання. Змінна оголошується з використанням специфікатора класу зберігання **register**:

```
int main(void)
{
    register int quick;
    ...
}
```

Ми говоримо «за сприятливим збігом обставин», тому що оголошення змінної як регістрової є скоріш запитом, ніж прямою вказівкою. Компілятор повинен зіставити ваші вимоги з кількістю доступних регістрів або об'єму швидкодіючої пам'яті, або ж він може просто проігнорувати запит, і ваше побажання не буде задовільнене. В такому випадку змінна стає звичайною автоматичною змінною. Проте, застосовувати до неї операцію взяття адреси так само неможна. Ви маєте змогу зробити запит, щоб формальні параметри були регістровими змінними. Для цього просто скористайтеся ключовим словом **register** в заголовку функції:

```
void macho(register int n)
```

Типи, які допускається оголошувати як **register**, можуть виявитися обмеженими. Наприклад, регістри в процесорі можуть бути недостатньо великими, щоб вмістити тип **double**.

11 Статичні змінні з областю видимості в межах блоку

Назва «статична змінна» звучить як взаємовиключне поняття, наче «змінна, яка не може бути змінена». В дійсності характеристика «статична» означає, що змінна залишається розміщеною в пам'яті, а не обов'язково відноситься до значення. Змінні з областю видимості в межах файлу автоматично (і обов'язково) мають статичну тривалість зберігання. Як згадувалося раніше, можна також створювати локальні змінні, що мають область видимості в межах блоку, але статистичну тривалість зберігання. Такі змінні мають таку ж саму область видимості, як автоматичні змінні, однак вони не зникають, коли функція, що їх

містить, завершує свою роботу. Іншими словами, такі змінні мають область видимості в межах блоку, не мають зв'язування, але мають статичну тривалість зберігання. Комп'ютер пам'ятає їх значення від одного виклику функції до наступного. Такі змінні створюються в результаті оголошення в блоці (що забезпечує область видимості в межах блоку і відсутність зв'язування) зі специфікатором класу зберігання **static** (що надає статичну тривалість зберігання).

Наступна програма ілюструє цей прийом. Текст програми має наступний вигляд:

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

void trystat(void);

int main(void)
{
    int count;

    SetConsoleOutputCP(1251);

    for(count = 1; count <= 3; count++)
    {
        printf("Ітерація %d:\n", count);
        trystat();
    }
    return 0;
}

void trystat(void)
{
    int fade = 1;
    static int stay = 1;

    printf("=======================\n");
    printf("    fade = %d; stay = %d\n", fade++, stay++);
    printf("=======================\n\n");
}
```

Зверніть увагу, що **trystat()** інкрементує кожну змінну після виводу її значення. Результат роботи програми наведено на рис. 1.3.

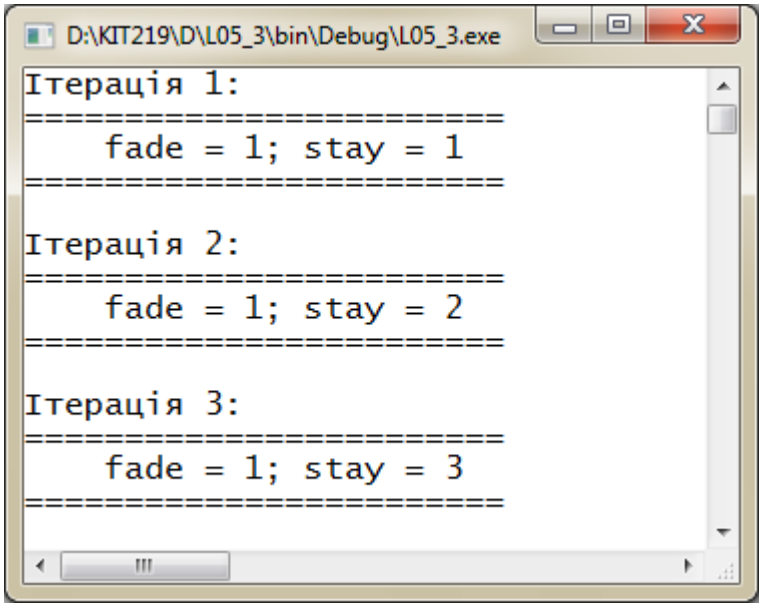

Рисунок 1.3 – Результат роботи програми, яка демонструє статичні змінні з областю видимості в межах блоку

Статична змінна **stay** запам'ятовує, що її значення було збільшено на **1**, але змінна **fade** кожного разу починається заново. Це вказує на відмінність в ініціалізації: **fade** ініціалізується при кожному виклику **trystat()**, а **stay** – тільки одного разу, коли функція **trystat()** компілюється. Статичні змінні ініціалізуються нулем, якщо вони не були явно ініціалізовані іншим значенням.

Два наступних оголошення виглядають схожими:

```
int fade = 1;
static int stay = 1;
```

Проте, перший оператор в дійсності є частиною функції **trystat()** і виконується кожного разу, коли функція викликана. Це дія часу виконання. Другий оператор насправді не стосується функції **trystat()**. Якщо ви застосуєте відлагоджувач для покрокового виконання програми, то побачите, що програма як би пропускає цей крок. Причина в тому, що після того, як програма завантажилася в пам'ять, статичні змінні та зовнішні змінні вже знаходяться в потрібних місцях. Розміщення оператора оголошення до функції **trystat()** повідомляє компілятору, що тільки функції **trystat()** дозволено бачити дану змінну: це не оператор, який виконується під час виконання.

Використовувати модифікатор **static** для параметрів функції неможна:

```
int wontwork(static int flu);       // не дозволено
```

Іншим терміном для статичної змінної з областю видимості в межах блоку є «локальна статична змінна». Крім того, якщо ви читали ранню літературу по **C**, то мабуть побачили, що цей клас зберігання називали внутрішнім статичним класом зберігання. Проте, слово внутрішній застосовувалося для вказування на оголошення всередині функції, а не на внутрішнє зв'язування.

12 Статичні змінні з зовнішнім зв'язуванням

Статична змінна з зовнішнім зв'язуванням має область видимості в межах файлу, зовнішнє зв'язування і статичну тривалість зберігання. Такий клас іноді називають **зовнішнім класом зберігання**, а змінні цього типу – **зовнішніми змінними**. Зовнішня змінна створюється шляхом розміщення оголошення за рамками усіх функцій. Відповідно до документації, зовнішня змінна може додатково бути оголошена всередині функції, в якій вона використовується, з застосуванням ключового слова **extern**. Якщо будь-яка зовнішня змінна визначена в одному файлі початкового коду та використовується в іншому файлі початкового коду, то оголошення цієї змінної в іншому файлі з ключовим словом **extern** є обов'язковим. Оголошення виглядає наступним чином:

```
int Errupt;          // змінна, що має зовнішнє визначення
double Up[100];      // масив, що має зовнішнє визначення
extern char Coal;    // обов'язкове оголошення, якщо
                     // Coal визначається в іншому файлі
void next(void);

int main(void)
{
    extern int Errupt;     // необов'язкове оголошення
    extern double Up[];    // необов'язкове оголошення
}

void next(void)
{
    ...
}
```

Зверніть увагу, що ви не зобов'язані вказувати розмірність масиву в необов'язковому оголошенні `double Up[]`. Це пояснюється тим, що початкове оголошення `double Up[100];` вже надало таку інформацію. Групу оголошень `extern` всередині `main()` можна повністю опустити, оскільки зовнішні оголошення мають область видимості в межах файлу, тому вони відомі від місця оголошення і до кінця файлу. Однак вони призначені для документування намірів застосовувати ці змінні в `main()`.

Якщо ключове слово `extern` відсутнє в оголошенні всередині функції, створюється окрема автоматична змінна. Тобто, заміна

```
extern int Errupt;
```

оголошенням

```
int Errupt;
```

в `main()` призводить до того, що компілятор створює автоматичну змінну на ім'я `Errupt` – окрему локальну змінну, яка відрізняється від початкової змінної `Errupt`. Ця локальна змінна буде знаходитися в області видимості під час виконання `main()`, але для інших функцій, таких як `next()`, що розташовані в тому ж самому файлі, в області видимості буде зовнішня змінна `Errupt`. Тобто, змінна з областю видимості в межах блоку «приховує» змінну з тим самим іменем, що має область видимості в межах файлу, коли відбувається виконання операторів в цьому блоці. Якщо за будь-якої малоймовірної причини вам дійсно необхідна локальна змінна, що має те ж саме ім'я, що й глобальна змінна, можете скористатися в локальному оголошенні специфікатором класу зберігання `auto`, щоб явно документувати свій вибір.

Зовнішні змінні мають статичну тривалість зберігання. Таким чином, масив `Up` існує і зберігає свої значення незалежно від того, виконується `main()`, `next()` або будь-яка інша функція.

В наступних трьох прикладах показані чотири можливі комбінації зовнішніх і автоматичних змінних. В прикладі №1 присутня одна зовнішня змінна `Hocus`, яка відома і `main()`, і `magic()`.

Приклад №1

```
int Hocus;
int magic();

int main(void)
{
    extern int Hocus;  // змінна Hocus оголошена як зовнішня
}

int magic()
{
    extern int Hocus;  // та ж сама змінна Hocus, що й раніше
    ...
}
```

В прикладі №2 є одна зовнішня змінна **Hocus**, яка відома обом функціям. На цей раз функція **magic()** знає про неї за замовчуванням.

Приклад №2

```
int Hocus;
int magic();

int main(void)
{
    extern int Hocus;  // змінна Hocus оголошена як зовнішня


int magic()
{
    // змінна Hocus не оголошена, але відома
    ...
}
```

В прикладі №3 створюються чотири змінні. Змінна **Hocus** в **main()** є автоматичною за замовчуванням і локальною для **main()**. Змінна **Hocus** в **magic()** явно оголошена як автоматична і відома тільки **magic()**. Зовнішня змінна **Hocus** не відома **main()** або **magic()**, але буде відома будь-якій іншій функції в даному файлі, що не має власної локальної змінної **Hocus**. Нарешті, **Pocus** – це зовнішня змінна, яка відома **magic()**, але не **main()**,тому що **Pocus** знаходиться за **main()**.

Приклад №3

```
int Hocus;
int magic();
```

```
int main(void)
{
    int Hocus;        // змінна Hocus оголошена, за
                      // замовчуванням є автоматичною
    ...
}

int Pocus;
int magic()
{
    auto int Hocus;   // локальна змінна Hocus оголошена
                      // як автоматична
    ...
}
```

Наведені приклади ілюструють область видимості зовнішніх змінних, яка поширюється від місця їх оголошення і до кінця файлу. Вони також відображають час життя змінних. Зовнішні змінні **Hocus** і **Pocus** існують протягом всього часу виконання програми, а оскільки вони не обмежені будь-якою однією функцією, вони не зникають після завершення конкретної функції.

13 Ініціалізація зовнішніх змінних

Як і автоматичні, зовнішні змінні можуть ініціалізуватися явно. На відміну від автоматичних, зовнішні змінні за замовчуванням ініціалізуються нулем, якщо ви не ініціалізували їх. Це правило можна застосовувати також до елементів зовнішньо визначеного масиву. Однак для ініціалізації змінних з областю видимості в межах файлу можна використовувати тільки константні вирази, що відрізняється від випадку автоматичних змінних.

```
int x = 10;               // допустимо, 10 - це константа
int y = 3 + 20;           // допустимо, константний вираз
size_t z = sizeof(int);   // допустимо, константний вираз
int x2 = 2 * x;           // недопустимо, x – це змінна
```

За умови, що типом не є масив, вираз **sizeof** вважається константним.

14 Використання зовнішньої змінної

Давайте розглянемо простий приклад, в якому задіяна зовнішня змінна. Припустимо, що дві функції з іменами **main()** і **critic()** повинні мати доступ до

змінної **units**. Це можна зробити, оголосивши **units** за межами двох функцій, що згадувалися раніше, як це показано в наступній програмі. Текст програми має такий вигляд:

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

int units = 0;           // зовнішня змінна

void critic(void);

int main(void)
{
    SetConsoleOutputCP(1251);

    extern int units;    // необов'язкове повторне оголошення
    printf("Скільки фунтів важить маленький бочонок олії?\n");
    scanf("%d", &units);
    while(units != 56)
        critic();
    printf("Ви вгадали!\n");
    return 0;
}

void critic(void)
{
    // необов'язкове повторне оголошення опущене
    printf("Вам не пощастило. Спробуйте ще раз.\n");
    scanf("%d", &units);
}
```

Результат роботи програми наведено на рис. 1.4.

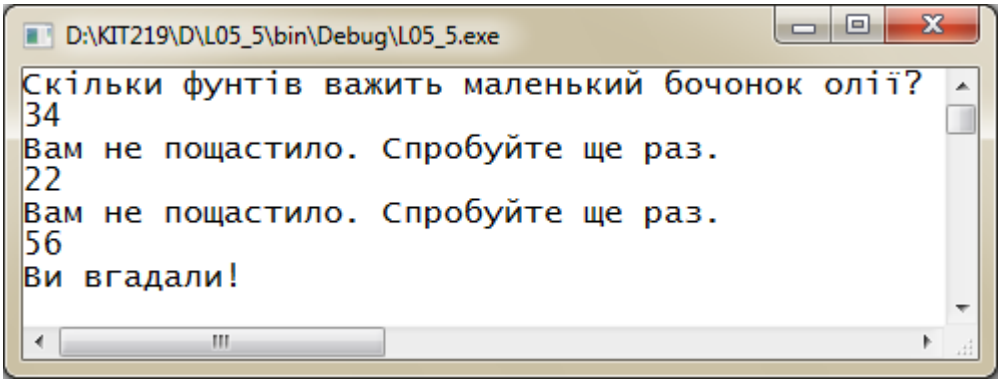

Рисунок 1.4 – Результат роботи програми, яка демонструє використання зовнішньої змінної

Зверніть увагу, що друге значення для **units** читається функцією **critic()**, але **main()** також відоме нове значення після завершення циклу **while**. Таким чином, і **main()**, і **critic()** використовують ідентифікатор **units** для доступу до однієї і тієї ж самої змінної. В рамках термінології **C** ми говоримо, що змінна **units** має область видимості в межах файлу, зовнішнє зв'язування і статичну тривалість зберігання.

Ми зробили **units** зовнішньою змінною, визначивши її за межами визначень усіх функцій. Це і все, що необхідно зробити для забезпечення доступності **units** усім наступним функціям у файлі.

Давайте поглянемо на деякі деталі. Перш за все, оголошення змінної **units** там, де воно знаходиться, робить її доступною оголошеним далі функціям без додаткових умов. Відповідно, функція **critics()** користується змінною **units**.

Аналогічно, нічого не треба буде робити для надання доступу до **units** функції **main()**. Однак в **main()** є таке оголошення:

```
extern int units;
```

У прикладі, що розглядається, це оголошення є головним чином формою документування. Специфікатор класу зберігання **extern** повідомляє компілятору, що будь-яке згадування **units** в даній функції відноситься до змінної, що оголошена за межами цієї функції, можливо, навіть поза самим файлом. І знову **main()** і **critic()** працюють зі змінною **units**, яка визначена зовні.

15 Зовнішні імена

Стандарти **C99** і **C11** потребують, щоб компілятори розрізняли перші **63** символи для локальних ідентифікаторів і перші **31** символи для зовнішніх ідентифікаторів. Це корегує попередню вимогу з розпізнавання перших **31** символів для локальних і перших **6** символів для зовнішніх ідентифікаторів. Цілком можливо, що ви маєте справу зі старими правилами. Причина того, що правила для імен зовнішніх змінних є більш обмежуючими, ніж правила для імен локальних змінних, пов'язана з тим, що зовнішні імена повинні дотримуватися правил локального середовища, які можуть бути більш жорсткими.

16 Визначення та оголошення

Поглянемо більш уважно на різницю між визначенням змінної та її оголошенням. Подивіться на наступний приклад:

```
int tern = 1;           // змінна tern визначена

int main(void)
{
    extern int tern;    // використання tern, що визначена
                        // десь в іншому місці
    ...
}
```

Тут змінна **tern** оголошується двічі. Перше оголошення призводить до того, що для змінної відводиться місце в пам'яті. Воно створює визначення змінної. Друге оголошення просто вказує компілятору на необхідність застосування змінної **tern**, яка була створена раніше, тому це не є визначенням. Перше оголошення називається **визначальним оголошенням**, а друге – **посилальним оголошенням**. Ключове слово **extern** говорить про те, що оголошення не є визначенням, оскільки воно повідомляє компілятор про необхідність пошуку визначення десь в іншому місці.

Припустимо, що ви записали наступний код:

```
extern int tern;
int main(void)
{
    ...
}
```

Компілятор припустить, що дане визначення **tern** знаходиться в іншому місці програми, можливо, в іншому файлі. Це оголошення не призводить до виділення пам'яті. Таким чином, не використовуйте ключове слово **extern** для створення зовнішнього визначення. Застосовуйте його тільки для посилання на існуюче зовнішнє визначення.

Зовнішня змінна може бути ініціалізована тільки один раз, і це повинно відбуватися при визначенні змінної. Погляньте на наступний код:

```
// файл one.c
char permis = 'N';
```

```
// файл two.c
extern char permis = 'Y';     // помилка
```

Помилка полягає в тому, що визначальне оголошення у файлі **one.c** вже було створено, і воно ініціалізувало змінну **permis**.

17 Статичні змінні з внутрішнім зв'язуванням

Змінні з цим класом зберігання мають статичну тривалість зберігання, область видимості в межах файлу і внутрішнє зв'язування. Вони створюються шляхом їх визначення поза межами будь-яких функцій (як і у випадку зовнішніх змінних) з вказуванням специфікатора класу зберігання **static**:

```
static int svil = 1;  // статична змінна, внутрішнє зв'язування
int main(void)
{
    ...
}
```

Змінні подібного роду колись отримали назву зовнішніх статичних змінних, але це дещо заплутує, оскільки вони мають внутрішнє зв'язування. Нажаль, новий компактний термін знайти не вдалося, тому залишається варіант статична змінна з внутрішнім зв'язуванням. Звичайна зовнішня змінна може використовуватися функціями в будь-якому файлі, який є частиною програми, але статична змінна з внутрішнім зв'язуванням може застосовуватися тільки функціями в тому ж самому файлі. Всередині функції можна повторно оголосити будь-яку змінну з областю видимості в межах файлу, використовуючи специфікатор класу зберігання **extern**. Таке оголошення не змінює тип зв'язування.

Розглянемо наступний код:

```
int traveler = 1;             // зовнішнє зв'язування
static int stayhome = 1;      // внутрішнє зв'язування
int main(void)
{
    extern int traveler;      // використання глобальної
                              // змінної traveler
    extern int stayhome;      // використання глобальної
                              // змінної stayhome
    ...
}
```

Змінні **traveler** і **stayhome** є глобальними в цій конкретній одиниці трансляції, але тільки **traveler** можна застосовувати в інших одиницях трансляції. Два оголошення, що використовують **extern**, документують той факт, що в **main()** застосовуються дві глобальні змінні, але **stayhome** продовжує мати внутрішнє зв'язування.

18 Використання декількох файлів

Відмінність між внутрішнім і зовнішнім зв'язуваннями важлива тільки в ситуації, коли програма будується з декількох одиниць трансляції, тому давайте коротко розглянемо дане питання.

Складні програми на **C** часто складаються з декількох окремих файлів початкового коду. Іноді в цих файлах виникає необхідність спільного використання будь-якої зовнішньої змінної. Щоб зробити це в **C**, необхідно передбачити визначальне оголошення в одному файлі та посилальне оголошення в інших файлах. Це означає, що в усіх оголошеннях крім одного (визначального оголошення) повинне бути присутнім ключове слово **extern**, а для ініціалізації змінної слід застосовувати тільки визначальне оголошення.

Зверніть увагу, що зовнішня змінна, яка визначена в одному файлі, не буде доступною в іншому файлі до тих пір, поки її також там не оголосити (з використанням **extern**). Саме по собі зовнішнє оголошення лише робить змінну потенційно доступною для іншого файлу.

Однак історично склалося таким чином, що багато компіляторів в цьому відношенні дотримуються інших правил. Наприклад, такі системи як **Unix** дозволяють оголошувати змінну в декількох файлах без зазначення ключового слова **extern** за умови, що тільки одне оголошення містить ініціалізацію. Оголошення з ініціалізацією вважається визначенням.