

Специфікатори класів зберігання. Виділення пам'яті



Лекція №6

Дисципліна «Програмування»

2-й семестр



Специфікатори класів зберігання

В мові **C** існує п'ять ключових слів, які представляють специфікатори класів зберігання: **auto**, **register**, **static**, **extern**, **_Thread_local** (C11).

Специфікатор **auto** вказує змінну з автоматичною тривалістю зберігання. Він може застосовуватися тільки в оголошеннях змінних з областю видимості в межах блоку, які вже мають автоматичну тривалість зберігання. Головним його призначенням є документування.

Специфікатор **register** також може використовуватися тільки зі змінними, що мають область видимості в межах блоку. Він поміщає змінну в регістровий клас зберігання, що рівносильно запиту на мінімізацію часу доступу до неї. Він також запобігає взяттю адреси цієї змінної.



Специфікатори класів зберігання

Специфікатор **static** створює об'єкт зі статичною тривалістю зберігання, який з'являється після завантаження програми в пам'ять і зникає при завершенні програми.

Якщо **static** застосовується в оголошенні з областю видимості в межах файлу, то область видимості обмежується одним цим файлом.

Якщо **static** використовується в оголошенні з областю видимості в межах блоку, то область видимості обмежується цим блоком.

Таким чином, об'єкт існує і зберігає своє значення протягом виконання програми, але може бути доступним за допомогою ідентифікатора, тільки коли виконується код всередині його блоку.

Статична змінна з областю видимості в межах блоку не має зв'язування. Статична змінна з областю видимості в межах файлу має внутрішнє зв'язування.



Специфікатори класів зберігання

Специфікатор **extern** вказує, що ви оголошуєте змінну, яка була визначена в будь-якому іншому місці.

Якщо оголошення, яке містить **extern**, має область видимості в межах файлу, то змінна, на яку йде посилання, повинна мати зовнішнє зв'язування.

Якщо оголошення з **extern** має область видимості в межах блоку, то змінна, на яку йде посилання, може мати або зовнішнє, або внутрішнє зв'язування, що залежить від визначального оголошення цієї змінної.

Специфікатор **_Thread_local** створює копію змінної для кожного потоку та знищується разом з ним. Він може використовуватися разом зі специфікаторами **static** і **extern**.



Приклад використання

Файл **main.c**

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

void report_count();
void accumulate(int k);
int count = 0;           // область видимості в межах файлу
                          // зовнішнє зв'язування

int main(void)
{
    int value;            // автоматична змінна
    register int i;       // регістрова змінна

    SetConsoleOutputCP(1251);

    printf("Введіть додатне ціле число (0 для завершення): ");
    while (scanf("%d", &value) == 1 && value > 0)
    {
        ++count;         // використання змінної з області
                          // видимості в межах файлу
        for (i = value; i >= 0; i--)
            accumulate(i);
        printf("Введіть додатне ціле число (0 для завершення):");
    }
}
```



Приклад використання

Файл `main.c`

```
    report_count();  
    return 0;  
}  
  
void report_count()  
{  
    printf("=====\n");  
    printf("Кількість разів виконання циклу: %d\n", count);  
    printf("=====\n");  
}
```

Файл `accumulate.c`

```
extern int count;           // посилальне оголошення,  
                             // зовнішнє зв'язування  
static int total = 0;      // статичне визначення,  
                             // внутрішнє зв'язування  
void accumulate(int k);    // прототип функції accumulate()
```

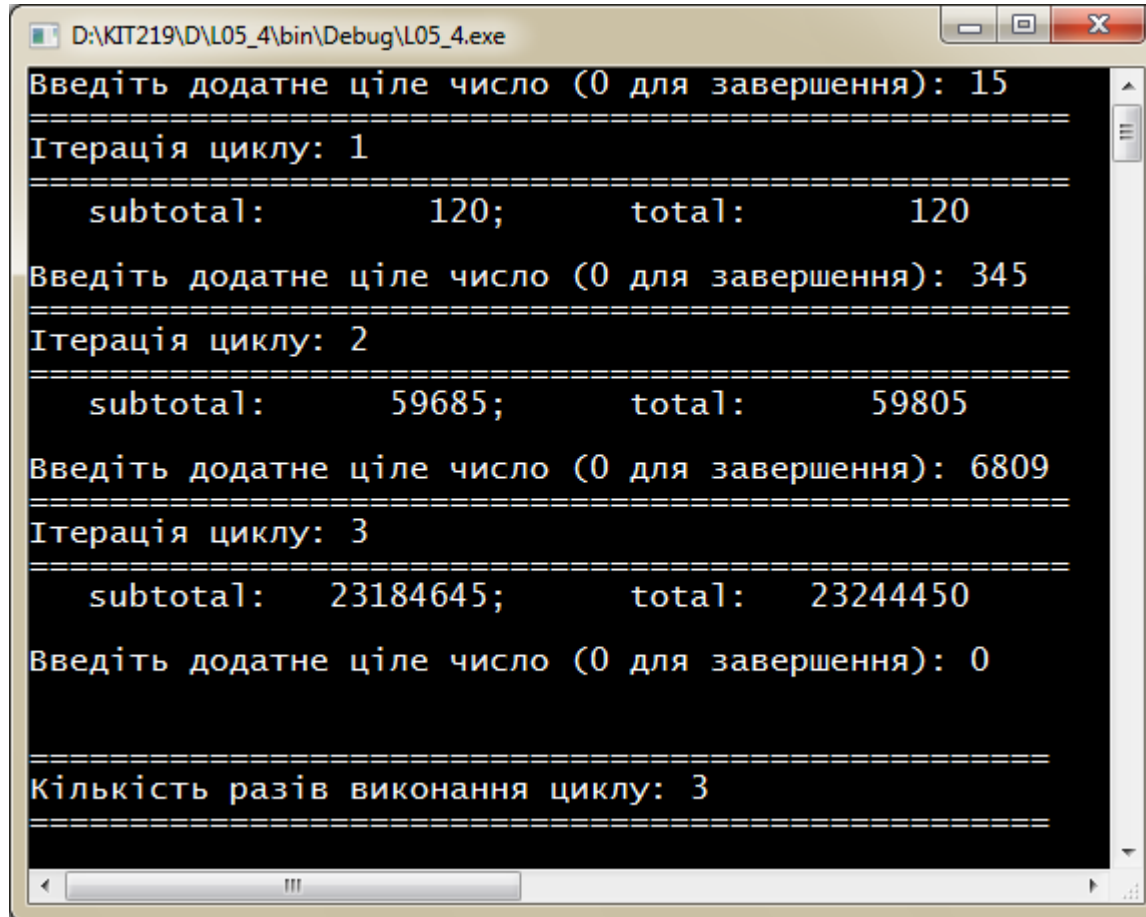


Приклад використання

Файл `accumulate.c` (закінчення)

```
void accumulate(int k)           // k має область видимості в межах
                                // блоку, зв'язування відсутнє
{
    static int subtotal = 0;     // статична змінна,
                                // зв'язування відсутнє
    if(k <= 0)
    {
        printf("=====\n");
        printf("Ітерація циклу: %d\n", count);
        printf("=====\n");
        printf("    subtotal: %10d;        total: %10d\n",
                subtotal, total);
        subtotal = 0;
    }
    else
    {
        subtotal += k;
        total    += k;
    }
}
```

Приклад використання



```
D:\KIT219\D\L05_4\bin\Debug\L05_4.exe
Введіть додатне ціле число (0 для завершення): 15
=====
Ітерація циклу: 1
=====
    subtotal:      120;      total:      120
Введіть додатне ціле число (0 для завершення): 345
=====
Ітерація циклу: 2
=====
    subtotal:    59685;      total:    59805
Введіть додатне ціле число (0 для завершення): 6809
=====
Ітерація циклу: 3
=====
    subtotal: 23184645;      total: 23244450
Введіть додатне ціле число (0 для завершення): 0

=====
Кількість разів виконання циклу: 3
=====
```




Класи зберігання та функції

Функції також мають класи зберігання. Функція може бути або **зовнішньою** (за замовчуванням), або **статичною**. В стандарті **C99** додана третя можливість – **вбудована функція**.

Доступ до зовнішньої функції можуть отримувати функції в інших файлах, але статична функція може застосовуватися тільки всередині файлу, де вона визначена.

```
double gamma(double);           // за замовчуванням є зовнішньою
static double beta(int, int);
extern double delta(double, int);
```

Функції `gamma()` і `delta()` можуть використовуватися функціями в інших файлах, які є частиною програми, але `beta()` – ні. Через обмеження функції `beta()` одним файлом, в інших файлах можна застосовувати інші функції з цим самим іменем.



Функція генерації випадкових чисел

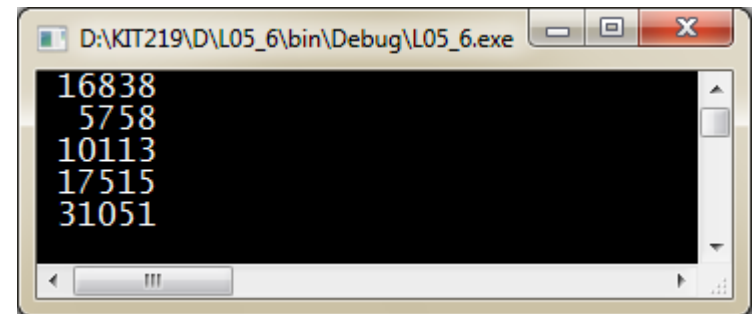
Файл **rand0.c**

```
static unsigned long int next = 1;  // початкове число
int rand0(void)
{
    // магічна формула генерації псевдовипадкових чисел
    next = next * 1103515245 + 12345;
    return (unsigned int)(next / 65536) % 32768;
}
```



Файл **main.c**

```
#include <stdio.h>
extern int rand0(void);
int main(void)
{
    int count;
    for(count = 0; count < 5; count++)
        printf("%6d\n", rand0());
    return 0;
}
```





Функція генерації випадкових чисел

Файл `rand1.c`

```
static unsigned long int next = 1;  // початкове число
int rand1(void)
{
    // магічна формула генерації псевдовипадкових чисел
    next = next * 1103515245 + 12345;
    return (unsigned int)(next / 65536) % 32768;
}

void srand1(unsigned int seed)
{
    next = seed;
}
```

2

Файл `main.c`

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

extern void srand1(unsigned int x);
extern int rand1(void);
```



Функція генерації випадкових чисел

Файл `main.c` (закінчення)

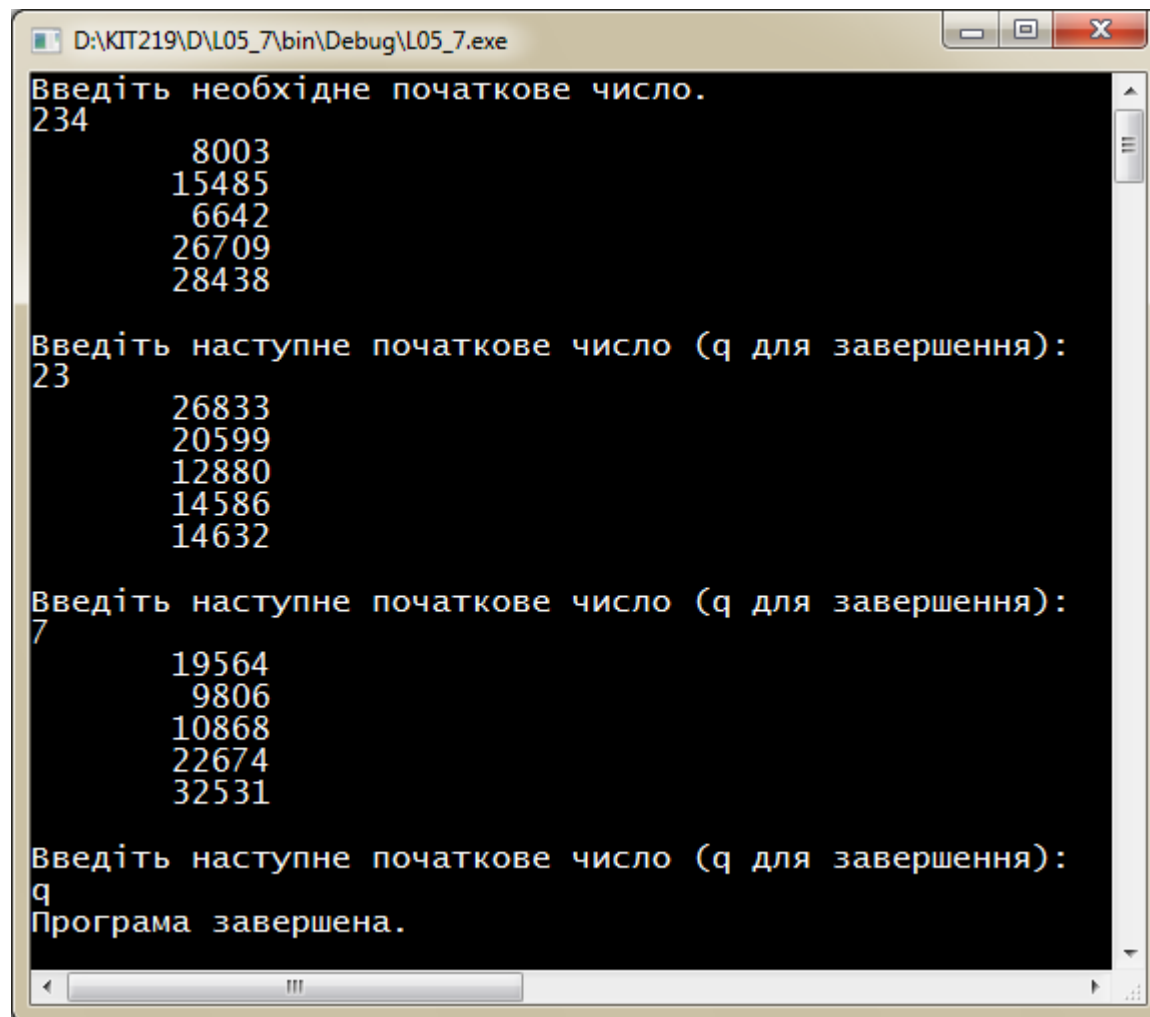
```
int main(void)
{
    int count;
    unsigned seed;

    SetConsoleOutputCP(1251);

    printf("Введіть необхідне початкове число.\n");
    while(scanf("%u", &seed) == 1)
    {
        srand1(seed);    // нове визначення початкового числа
        for(count = 0; count < 5; count++)
            printf("%12d\n", rand1());
        printf("\nВведіть наступне початкове число "
               "(q для завершення): \n");
    }
    printf("Програма завершена.\n");
    return 0;
}
```



Функція генерації випадкових чисел



```
D:\KIT219\D\L05_7\bin\Debug\L05_7.exe
Введіть необхідне початкове число.
234
      8003
     15485
      6642
     26709
     28438

Введіть наступне початкове число (q для завершення):
23
      26833
     20599
     12880
     14586
     14632

Введіть наступне початкове число (q для завершення):
7
      19564
      9806
     10868
     22674
     32531

Введіть наступне початкове число (q для завершення):
q
Програма завершена.
```



Функція malloc()

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

Функція `malloc()` **виділяє блок пам'яті** розміром `size` байт, і повертає вказівник на початок блоку.

Якщо пам'яті недостатньо, функція `malloc()` повертає нульовий вказівник (**NULL**).

Перш ніж намагатися використовувати вказівник, треба завжди перевіряти значення, що повертається.

Вміст виділеного блоку пам'яті не ініціалізується, він залишається з невизначеними значеннями.

Прототип функції знаходиться в файлі `stdlib.h`.



Застосування функції malloc()

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

```
struct Addr
```

```
{
```

```
    char name[40];
```

```
    char street[40];
```

```
    char city[40];
```

```
    char state[3];
```

```
    char zip[10];
```

```
};
```

```
struct Addr *get_struct(void)
```

```
{
```

```
    struct Addr *p, addr;
```

```
    if( !(p = (struct Addr *) malloc(sizeof(addr))) )
```

```
    {
```

```
        printf("Allocation error.");
```

```
        exit(0);
```

```
    }
```

```
    return p;
```

```
}
```

Програма виділяє необхідну кількість пам'яті для того, щоб розмістити структуру типу **Addr**.





Застосування функції malloc()

2

```
int main(void)
{
    struct Addr *str;

    str = get_struct();

    printf("name    = %s\n", str->name);
    printf("street  = %s\n", str->street);
    printf("city    = %s\n", str->city);
    printf("state   = %s\n", str->state);
    printf("zip     = %s\n", str->zip);
    printf("\n");
    free(p);
    return 0;
}
```

D:\KIT219\D\L6_03\bin\Debug\L6_03.exe

```
name    = И'2
street  = nGW\bin;C:\Program Files\CodeBlocks\MinGW;C:\
city    = W;C:\Windows\System32;C:\Windows;C:\Windows\S
state   = ows\System32\wbeR1010-X4
zip     = \System32\wbeR1010-X4
```




Застосування функції malloc()

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

```
int main(void)
```

```
{
```

```
    double *ptd;
```

```
    int max = 0;
```

```
    int number;
```

```
    int i = 0;
```

```
    SetConsoleOutputCP(1251);
```

```
    puts("Введіть максимальну кількість елементів типу double.");
```

```
    if (scanf("%d", &max) != 1)
```

```
    {
```

```
        puts("Введена кількість є некоректною. "
```

```
            "Програма завершена.");
```

```
        exit(EXIT_FAILURE);
```

```
    }
```

```
    ptd = (double *) malloc(max * sizeof(double));
```

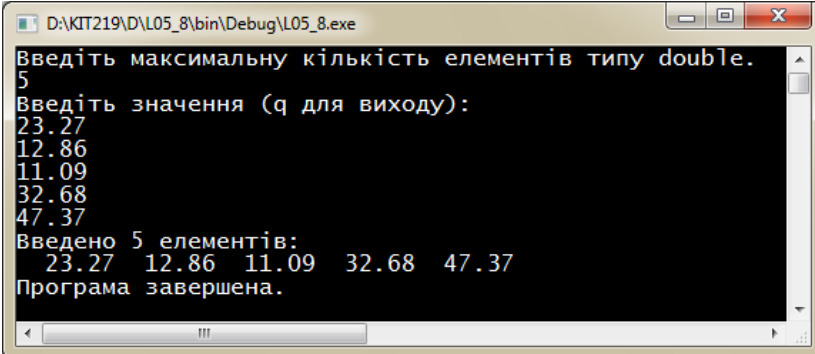
Програма виділяє пам'ять для розміщення необхідної кількості елементів масиву типу **double**.



Застосування функції malloc()

```
if (ptd == NULL)
{
    puts("Не вдалося виділити пам'ять. Програма завершена.");
    exit(EXIT_FAILURE);
}
// ptd тепер вказує на масив з max елементів
puts("Введіть значення (q для виходу):");
while (i < max && scanf("%lf", &ptd[i]) == 1)
    ++i;
printf("Введено %d елементів:\n", number = i);
for (i = 0; i < number; i++)
{
    printf("%7.2f", ptd[i]);
    if (i % 7 == 6) putchar('\n');
}
if (i % 7 != 0)
    putchar('\n');
puts("Програма завершена.");
free(ptd);
return 0;
```

4



```
D:\KIT219\DL05_8\bin\Debug\L05_8.exe
Введіть максимальну кількість елементів типу double.
5
Введіть значення (q для виходу):
23.27
12.86
11.09
32.68
47.37
Введено 5 елементів:
 23.27  12.86  11.09  32.68  47.37
Програма завершена.
```



Функція calloc()

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

Функція `calloc()` повертає вказівник на перший байт виділеної області пам'яті, розмір якої дорівнює $(\text{num} * \text{size})$, де `size` задається в байтах.

Це означає, що функція `calloc()` виділяє достатньо пам'яті для масиву з `num` об'єктів кожен з яких має розмір `size` байт.

Якщо пам'яті недостатньо, повертається нульовий вказівник (`NULL`).

Перш ніж намагатися його використовувати, треба завжди перевіряти значення, що повертається.

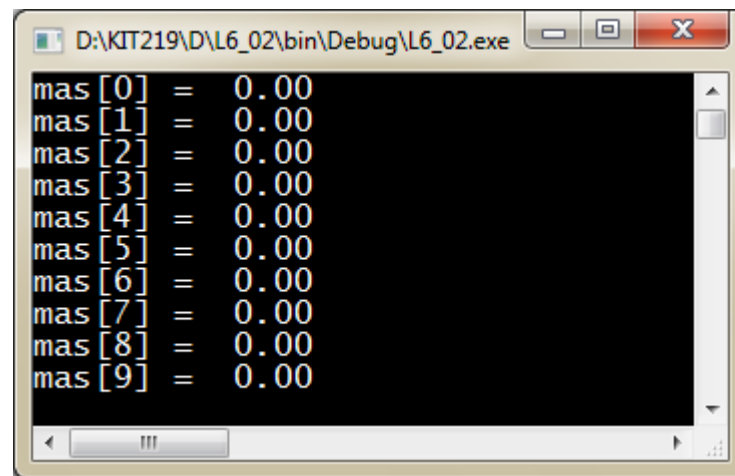
Вміст виділеного блоку пам'яті ініціалізується нулями.

Прототип функції знаходиться в файлі `stdlib.h`.

Застосування функції calloc()

```
#include <stdlib.h>
#include <stdio.h>
float *get_mem(void)
{
    float *p;
    p = (float *) calloc(10, sizeof(float));
    if(!p) {
        printf("Allocation failure.");
        exit(1);
    }
    return p;
}
int main(void)
{
    float *mas;
    mas = get_mem();
    for(int i = 0; i < 10; i++)
        printf("mas[%d] = %5.2f\n", i, mas[i]);
    free(p);
    return 0;
}
```

Програма повертає вказівник на динамічно виділений масив для 10 чисел типу **float**.



```
D:\KIT219\D\L6_02\bin\Debug\L6_02.exe
mas[0] = 0.00
mas[1] = 0.00
mas[2] = 0.00
mas[3] = 0.00
mas[4] = 0.00
mas[5] = 0.00
mas[6] = 0.00
mas[7] = 0.00
mas[8] = 0.00
mas[9] = 0.00
```



Функція `realloc()`

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

Функція `realloc()` змінює величину виділеної пам'яті, на яку вказує `ptr`, на нову величину, що задається параметром `newsize`.

Величина `newsize` задається в байтах і може бути більше або менше оригіналу.

Повертається вказівник на блок пам'яті, оскільки може виникнути необхідність перемістити блок при зростанні його розміру. В такому випадку вміст старого блоку копіюється в новий блок і інформація не втрачається.

Якщо вільної пам'яті недостатньо для виділення блоку розміром `newsize`, то повертається нульовий вказівник (`NULL`).

Прототип функції знаходиться в файлі `stdlib.h`.



Застосування функції `realloc()`

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(void)
{
    char *p;
    p = (char *) malloc(17);
    if(!p) {
        printf("Allocation error.");
        exit(1);
    }
    strcpy(p, "This is 16 chars");
    p = (char *) realloc(p, 18);
    if(!p) {
        printf("Allocation error.");
        exit(1);
    }
    strcat(p, ".");    printf(p);
    free(p);
    return 0;
}
```

Програма виділяє 17 байтів пам'яті, копіює рядок "This is 16 chars" в цю область, а потім використовує функцію `realloc()`, щоб збільшити розмір блоку до 18 байтів і помістити в кінці крапку.



Функція `free()`

```
void free(void * ptr);
```

Функція `free()` звільняє пам'ять, на яку вказує параметр `ptr`. В результаті ця пам'ять може виділятися знову.

Обов'язковою умовою використання функції `free()` є те, що пам'ять, яка звільняється, повинна попередньо бути виділена з використанням однієї з наступних функцій: `malloc()`, `calloc()` або `realloc()`.

Використання неправильного вказівника під час виклику цієї функції зазвичай веде до руйнування механізму керування пам'яттю.

Прототип функції знаходиться в файлі `stdlib.h`.



Кваліфікатори типів ANSI C

Відомо, що змінна характеризується типом і класом зберігання. В стандарті **c90** були додані ще дві властивості: **постійність** і **непостійність**. Ці властивості оголошуються за допомогою ключових слів **const** і **volatile** відповідно, які створюють **кваліфіковані типи**.

В стандарті **c99** з'явився третій кваліфікатор, **restrict**, що призначений для підтримки компілятора при оптимізації.

В стандарті **c11** доданий четвертий кваліфікатор, **_Atomic**. Стандарт **c11** надає додаткову бібліотеку **stdatomic.h** для підтримки паралельного програмування.

Стандарт **c99** наділяє кваліфікатори типів новою властивістю: один і той самий кваліфікатор можна вказувати в оголошенні декілька разів, і надмірні кваліфікатори ігноруються:

```
const const const int n = 6;    // те ж саме, що й  
                                // const int n = 6;
```