Лекція 2

Змінні

У цій главі...

2.1. Атрибути змінних

Знаючи алфавіт мови і структуру програми, ми можемо виконувати лише найпростіші дії, наприклад можемо створити і скомпілювати "порожню" програму, що являє собою деяку форму. Тепер час приступити до наповнення цієї форми конкретним змістом. У мові C++ зміст програми визначається виразами, що, у свою чергу, складаються з даних і операторів. Дані можуть бути або змінними, або константами. Для спрощення подальшого викладу дамо деякі означення.

Змінна — це абстрактна назва комірки чи декількох комірок пам'яті. Кожна змінна має шість атрибутів: ім'я, адресу, значення, тип, область видимості і час життя.

Правила утворення імен чи ідентифікаторів, сформульовані в главі 1. Однак щоб виконувати над змінною визначені дії, одного імені недостатньо. Необхідно знати, у якій області пам'яті вона розташована, а також діапазон припустимих значень і набір передбачених операцій. Крім того, необхідно контролювати, чи доступна змінна в конкретній точці програми.

Адреса змінної — це адреса комірки пам'яті, зв'язаної з даною змінною. Якщо змінна займає декілька осередків, її адресою вважається адреса першого осередку. У різних місцях програми можуть існувати різні змінні, що мають однакові імена, але різна адреси. Наприклад, якщо програма складається з двох функцій `f1()` і `f2()`, у кожній з них можна визначити змінну, що має те саме ім'я, скажемо `var`. Це різні змінні, оскільки вони зв'язані з різними функціями і відділені одна від одної невидимою стіною, що обмежує їхню *область видимості*. Такі змінні називаються *локальними*. Вони з'являються усередині функцій чи окремих блоків.

Ось як виглядає оголошення локальної цілочисельної змінної `var` усередині функції `f()`, що не має жодних аргументів і не повертає жодних значень.

```
void f()
{
  int var;
  …
}
```

змінна `var` існує тільки усередині функції `f()` і є недоступною ззовні. Вона виникає при вході у функцію і зникає після виходу з неї.

Глобальні змінні доступні в будь-якій точці програми. Оголошення глобальної змінної повинне знаходитися поза будь-якою функцією. Їхні імена також можуть збігатися з іменами локальних змінних (таке явище називається *маскуванням*), але при цьому виникає *конфлікт імен*. Для його розв'язання існує особливий засіб — оператор розв'язування області видимості `::`. Глобальна змінна існує від моменту її оголошення і знищується після завершення роботи програми.

Розглянемо приклад маскування глобальної змінної і спосіб розв'язання конфлікту імен.

Приклад маскування глобальної змінної

```
#include <iostream.h>
int var; // Оголошення глобальної змінної
int main()
{
 var = 0;
 {
    int var; // Оголошення локальної змінної
    var = 1;
 } // Локальна змінна var автоматично знищується
 cout << var; // Виводимо значення глобальної змінної

 return 0;
}
```

Запустивши цю програму на виконання, ми побачимо на екрані число 0. Це значить, що програма розрізняє дві однойменні змінні `var` — глобальну і локальну. Локальна змінна `var`, оголошена усередині блоку, обмеженого фігурними дужками, маскує глобальну змінну `var`. Якщо програміст досить уважний, він легко

упорається з цією ситуацією. Однак професіонал не став би ризикувати, називаючи локальну і глобальну змінні однаковими іменами. А якщо вже з якихось причин довелося порушити правила гарного тону, на допомогу прийде оператор розв'язування області видимості `::`. Поставивши його перед ім'ям змінної `var`, ми зв'яжемо це ім'я не з локальної, а з глобальною змінною.

Дозвіл області видимості

```
#include <iostream.h>
int var; // Оголошення глобальної змінної
int main()
{
 var = 0;
 {
    int var; // Оголошення локальної змінної
    var=0;
    ::var = 1; // Привласнюємо одиницю глобальній змінній
 } // Локальна змінна var автоматично знищується
 cout << var; // Виводимо значення глобальної змінної

return 0;
}
```

Тепер на екрані ми побачимо значення глобальної змінної, рівне 1, оскільки оператор розв'язування області видимості зняв конфлікт імен. Помітимо, що цей оператор зв'язує ім'я лише з глобальної змінної незалежно від глибини вкладення блоків.

Дозвіл області видимості у вкладених блоках

```
#include <iostream.h>
int var; // Оголошення глобальної змінної
int main()
{
 var = 0;
 {
   int var; // Оголошення локальної змінної
   var=1;
   {
     int var;
    var = 2;
    ::var = 1; // Привласнюємо одиницю глобальної змінної,
               // а не змінної var з найближчого зовнішнього блоку
   }
 } // Локальна змінна var автоматично знищується
 cout << var; // Виводимо значення глобальної змінної

return 0;
}
```

Крім того, та сама змінна може "розмножуватися", створюючи свої "клони". Наприклад, рекурсивна функція при кожнім виклику створює чергову копію змінної, оголошеної в її тілі. Новий "клон" має те ж ім'я, але знаходиться в іншому осередку. З концепцією адреси змінної тісно зв'язане поняття *вказівника*, тобто змінної, що зберігає адресу іншої змінної.

Значення змінної — це двійкове число, записане в комірках пам'яті, зв'язаних з даною змінною. Осередок являє собою окрему одиницю пам'яті, що має адресу. Як правило, у більшості сучасних комп'ютерів розмір осередку дорівнює одному байту, що складається з восьми битів.

Тип змінної визначає діапазон значень, що вона може набувати, і набір операцій, що до неї можна застосовувати. Поняття типу в мові C++ є принципово важливим, тому ми розглянемо його окремо, а поки зупинимося на концепції *зв'язування*, що також представляє дуже великий інтерес.

Зв'язування — це процес установлення зв'язку між атрибутом і змінною. Момент, коли цей зв'язок установлюється, називається *часом зв'язування*. Зв'язування може відбуватися під час компіляції або виконання програми, у залежності від цього розрізняють *статичне* і *динамічне зв'язування*.

Змінна стає доступною, лише якщо вона зв'язана з типом даних. Тут важливі два моменти: яким чином указується тип і коли відбувається зв'язування. У мовах програмування тип указується за допомогою оголошення: явного чи неявного. *Явне оголошення* — це оператор програми, у якому перераховані імена змінних і зазначений їхній тип. *Неявне оголошення* — це механізм зв'язування змінних з типами на основі прийнятих угод. У цьому випадку оператор оголошення не потрібний — перша поява імені змінної в програмі вважається її неявним оголошенням. Класичний приклад неявного оголошення змінних можна зустріти в програмах мовою FORTRAN, у якому всі змінні, імена яких починаються з букв I, J, K, L, M і N, за замовчуванням вважаються цілочисельними, а інші — дійсними. І явне, і неявне оголошення є статичними.

Як ми уже відзначали, у мові C++ необхідно розрізняти оголошення і визначення. Оголошення встановлюють типи й інші атрибути, але не приводять до розподілу пам'яті. Визначення задають атрибути і виділяють пам'ять. Будь-яка змінна, як і будь-яка інша сутність у мові C++, може мати необмежену кількість погоджених між собою оголошень, але тільки одне визначення. У мові C++ можна навіть оголошувати змінні, зовнішні стосовно функції. Таке оголошення вказує компілятору тип змінної і те, що вона визначена (чи буде визначена) в іншому місці.

Динамічне зв'язування змінної характеризується тим, що тип в операторі не вказується. Зв'язування змінної з типом відбувається в момент присвоювання їй нового значення. При цьому змінна, що стоїть в лівій частині оператора присвоювання, зв'язується з типом сутності, зазначеної в правій частині. Мови, що допускають динамічне оголошення змінних, відрізняються високою гнучкістю, але мають два недоліки.

По-перше, вони не дозволяють контролювати коректність присвоювання, якщо змінні в лівій і правій частинах оператора присвоювання мають різні типи. Отже, розпізнати *несумісні типи* неможливо. Помітимо, що цього недоліку не позбавлена і мова C++, що у багатьох ситуаціях передбачає автоматичне перетворення типу правої частини оператора присвоювання в тип його лівої частини.

По-друге, динамічне зв'язування типів сполучене з великими додатковими витратами пам'яті і часу. Це зв'язано з тим, що кожна змінна повинна супроводжуватися докладним описом її типу, а пам'ять, виділена для неї, повинна мати змінний розмір.

З цих причин мова C++ використовує лише *явні статичні оголошення змінних*, хоча в механізмі приведення і перетворення типів можна побачити слабкі відгомони динамічного зв'язування.

Область видимості і *час життя змінної* тісно зв'язані між собою. Будь-яку змінну необхідно зв'язати зі своїм осередком. Для цього адресу осередку необхідно витягти зі списку вільної пам'яті (купи). Цей процес називається *розміщенням змінної в пам'яті*. Зворотний процес називається *видаленням змінної з пам'яті*, чи *знищенням об'єкта*. Він полягає в тім, що адреса осередку, раніше виділеної для збереження змінної, повертається в список вільної пам'яті.

Час життя змінної — це відрізок часу, протягом якого змінна зв'язана з визначеною коміркою пам'яті. Отже, початком існування змінної варто вважати момент її зв'язування з визначеною коміркою пам'яті, а кінцем — момент її відкріплення від цього осередку.

2.2. Основні типи даних

Перша частина курсу присвячена імперативному програмуванню, тому ми розглянемо лише елементарні типи, успадковані від мови C: символ, розширений символ, логічне змінна, ціле число, число з точкою, що плаває, число з точкою подвоєної точності, що плаває, і "порожній" тип. Їм відповідають наступні ключові слова: `char`, `wchar_t`, `bool`, `int`, `float`, `double` і `void`. Всі інші типи даних, включаючи вказівники і посилання, а також типи, визначені користувачем, є похідними. Розмір змінних і діапазон їхніх значень залежать від типу процесора і компілятора. У будь-якому випадку точкою відліку є розмір символу, що дорівнює одному байту. Розмір цілочисельної змінної звичайно дорівнює довжині слова, прийнятої в конкретній операційній системі. У 16-розрядних операційних системах (DOS і Windows 3.1) розмір слова дорівнює 2 байт, а в 32- розрядних — 4 байт (Windows 9X, Windows 2000 і т.д.). Кожен компілятор мови C++ передбачає допоміжний тип `size_t`. Насправді це — не самостійний тип, а лише синонім цілочисельного типу, що має найменший розмір. Інакше кажучи, якщо написати в програмі оголошення

```
int var;
```

і спробувати виконати її в різних операційних системах, неможливо передбачити, скільки байтів буде виділено для змінної `var`: 2 чи 4. Однак якщо змінити тип змінної `var`

```
size_t var;
```

то компілятор сам визначить її розмір, керуючись своїми установками.

Стандарт мови C++ обумовлює лише *мінімальний діапазон*, у якому змінюються значення змінних кожного типу, а їхній розмір у байтах залишає встановлювати компілятору.

2.2..1. Символи

Символьні дані подаються у вигляді цифрових кодів відповідно до системи кодування ASCII (American Standard Code for Information Interchange — Американський стандартний код обміну інформацією), у якій для кодування 128 різних символів використовується діапазон значень 0–127. Для обробки цих кодів у мові C++ передбачений окремий елементарний тип — `char`.

В даний час програми уже не обмежуються англійською мовою. Програми, що розповсюджуються Інтернетом, повинні підтримувати різні національні алфавіти, що, як відомо, відрізняються великою розмаїтістю. Це змусило розроблювачів стандарту мови C++ включити в нього тип розширеного символу `wchar_t`. Розмір цього типу залежить від компілятора (у Visual C++ він дорівнює 2 байт, а в старих компіляторах — 1 байт). Строго говорячи, тип `whar_t` не є самостійним. У залежності від компілятора він перевизначається на основі одного з елементарних типів.

2.2..2.2.Логічні змінні

Логічні змінні належать до булевого типу, що позначається ключовим словом `bool`. Вони набувають лише два значення — істина і хибність. Донедавна як логічні значення використовувалися лише цілі числа. У виразах, що використовують такі операнди, будь-які ненульові значення є істинними, а нуль вважається помилковим значенням. У ході стандартизації мови C++ цей недолік був виправлений, і з'явився особливий тип `bool`. Змінні цього типу набувають два значення: `true` і `false`. Таким чином, програмісти можуть самі вибирати, яким способом користатися. Перевагу має, звичайно, новий стиль. Він більш простий і дозволяє уникнути багатьох непорозумінь. Однак прихильники традиційного підходу можуть використовувати числові значення замість булевих.

Теоретично, для подання булевої змінної достатньо одного біта. Однак у мові C++ мінімальною одиницею, що має адресу, є байт. Тому розмір таких змінних дорівнює 1 байт.

2.2..3. Цілі числа

Колись люди винайшли натуральні числа. З їхньою допомогою люди перелічують предмети і привласнюють їм порядкові номери. Потім, у міру розвитку товарно-грошових стосунків, люди стали робити борги і зрозуміли, що додатних чисел явно недостатньо, і придумали від'ємні (Брахмагупта, близько 628 р.). Тому зовсім не дивно, що найбільш розповсюдженим елементарним числовим типом є *ціле число* (`int`).

З програмістської точки зору цілі числа являють собою набір бітів, до того ж один з бітів (як правило, старший) задає знак (0 означає додатне число, а 1 — від'ємне). Цілі типи даних є зовсім природними і підтримуються комп'ютером на апаратному рівні.

Для представлення негативних цілих чисел використовується або додатковий, або зворотний двійковий код. Додатковий код негативного цілого числа в двійковій системі утворюється шляхом логічного доповнення додатного числа (усі біти заміняються протилежними) і додавання до нього одиниці. При зворотному записі від'ємне значення цілого числа зберігається як логічне доповнення до його абсолютного значення. Точне подання цілого числа залежить від конкретного комп'ютера й операційної системи.

Якщо заздалегідь відомо, що деяка цілочисельна величина є тільки додатною, не варто витрачати окремий біт для індикації знаку. Це вдвічі збільшує діапазон зміни числа. Для того щоб подати ціле число без знаку, використовується модифікатор `unsigned`. Для симетрії в мові C++ передбачений і модифікатор `signed`, але оскільки всі цілочисельні змінні за замовчуванням мають знак, це ключове слово практично не вживається. Цілі числа зі знаком змінюються від –32768 до 32767. Цілі числа без знака змінюються від 0 до 65535.

Оскільки символ у мові C++ задається у вигляді коду, він практично нічим не відрізняється від цілого числа, щоправда, у силу невеликого розміру діапазон його зміни дуже вузький. Наприклад, модифікований тип `signed char` дозволяє подати цілі числа від –128 до 127. Але ж код не може бути від'ємним, отже, 128 від'ємних чисел витрачаються даремно! Щоб виправити ситуацію і заодно розширити діапазон припустимих символів, у мові передбачений модифікований тип `unsigned char`. Символи без знаків кодуються числами від 0 до 255. як правило, цього достатньо.

Крім того, цілі числа можна модифікувати за допомогою ключових слів `short` і `long`. За замовчуванням модифікатор `short` еквівалентний типу `int`. Набагато кориснішим виявився модифікатор `long`, що дозволяє розширити діапазон зміни цілих чисел більш ніж у 30 000 разів: від –2147483648 до 2147483647. Скориставшись модифікатором `unsigned`, можна подати цілі числа від 0 до 4294967295.

За замовчуванням усі цілі чи, як їх ще називають, *інтегральні* типи мають знак, і тому модифікатор `signed` практично не застосовується.

Цілі числа володіють однією неприємною особливістю. Оскільки їм виділяється фіксований обсяг пам'яті, вони можуть переповняти його. Особливо це стосується типів `char` і `int`, що мають щодо невеликий діапазон. Розглянемо конкретний приклад.

Вихід цілих чисел за межі діапазону

```
#include <iostream.h>
int main()
{
 char c = 127;
    c=c+1;
 cout << (int)c << endl;

 unsigned char uc = 255;
 uc = uc+1;
 cout << (int)uc << endl;

 int i=32767;
 i=i+1;
 cout << i << endl;

 unsigned int ui=65535;
 ui=ui+1;
 cout << ui << endl;

 long int li=0x7FFFFFFFL;
 li=li+1;
 cout << li << endl;

 unsigned long int uli=0xFFFFFFFFUL;
 uli=uli+1;
 cout << uli << endl;

 return 0;
}
```

На екрані з'являться наступні числа.

```
-128
0
-32768
0
-2147483648
0
```

Подивимося, що відбувається при переповненні інтегральних типів.

```
char:
    01111111   127
   +00000001     1
   =10000000  -128 (додатковий код)
 unsigned char:
    11111111   255
   +00000001     1
   =00000000     0 (старший біт виходить за межі розрядної сітки)
 int:
    0111111111111111  32767
   +0000000000000001      1
   =1000000000000000 -32768 (додатковий код)
 unsigned int:
    1111111111111111  65535
   +0000000000000001      1
   =0000000000000000      0 (старший біт загублений)
 long int:
    01111111111111111111111111111111  2147483647
   +00000000000000000000000000000001           1
   =10000000000000000000000000000000 -2147483648 (додатковий код)

 unsigned long int:
    11111111111111111111111111111111  4294967295
   +00000000000000000000000000000001           1
   =00000000000000000000000000000000           0(старший біт загублений)
```

2.2..4. Числа з точкою, що плаває

У мові C++ передбачені три типи чисел із точкою, що плаває: float, double і long double. Варто мати на увазі, що насправді вони є раціональними і лише апроксимують дійсні числа. Отже, *комп'ютер оперує винятково раціональними числами.*

Числа з точкою, що плаває, породжують великі проблеми. По-перше, двійковий спосіб запису мало пристосований для таких чисел. Наприклад, деякі скінченні десяткові величини неможливо подати скінченним набором двійкових чисел. По-друге, ірраціональні числа доводиться апроксимувати раціональними числами із скінченною кількістю двійкових знаків у дробовій частині. По-третє, при обчисленнях часто відбувається втрата точності, що приводить до обчислювальної катастрофи.

Число з точкою, що плаває, складається зі знакового біта, показника ступеня і мантиси. В даний час найбільш розповсюдженим є формат, описаний стандартом IEEE Floating-Point Standard 754.

Тип	Довжина, біт	Знаковий біт	Показник ступеня, біт	Мантиса, біт
float	32	1	8	23
double	64	1	11	52
long double	80	1	15	64

Оскільки числа з точкою, що плаває, лише апроксимують дійсні числа, виникає питання, наскільки точно вони це роблять і в якому діапазоні можуть змінюватися. Крім того, не всі цифри в поданні числа з точкою, що плаває, є значущими.

Тип	Min	Max	Eps
float	1.1754944351e-38	3.4028233466e+38	1.192092896e-07
double	2.2250738585072014e-308	1.79769331348623258e-308	2.220446049250313e-16
long double	3.3621031431120935063e-4932	1.189731495357231765e+4932	1.08420217248550443412e-19

Зверніть увагу на те, що мінімальна різниця між двома числами набагато більше, ніж просто мінімально можливе значення змінної. Це не випадково. Справа в тім, що кожен тип числа з точкою, що плаває, характеризується кількістю значущих цифр після десяткової точки. Для типу float ця кількість точних цифр дорівнює 6, для типу double — 15, для типу long double — 19. Інші цифри є випадковими. Як бачимо, саме ці числа визначають порядок мінімальної різниці між двома змінними. Параметри, що характеризують точність апроксимації і діапазон зміни чисел із точкою, що плаває, визначаються в заголовному файлі float.h.

Отже, проаналізувавши властивості елементарних типів, можна дійти наступного висновку: для подання цілих чисел у більшості випадків краще використовувати тип long, а не int, а для обчислень з дійсними числами — тип double, а не float. Тип long double варто вибирати лише для дуже точних обчислень.

Варто зауважити, що деякі математичні задачі неможливо розв'язати за допомогою стандартної арифметики. Зокрема, у процесі розв'язування погано обумовлених систем лінійних алгебраїчних рівнянь можуть виникати раціональні числа, що виходять за межі припустимого діапазону. Нагадаємо, що будь-яке раціональне число можна подати у вигляді дробу, чисельник і знаменник якої являють собою цілі числа. Ці числа можуть складатися з тисяч цифр, кожна з який необхідна. Наприклад, у ході розв'язування методом Гаусса системи лінійних алгебраїчних рівнянь з матрицею Гільберта, порядок якої дорівнює 100, виникає раціональне число, чисельник якого складається з 251 цифри. Якщо заокруглити це число, виникне велика помилка. Тим часом, навіть самий "довгий" тип чисел із точкою, що плаває - long double - надає в наше розпорядження лише 19 значущих цифр. За таких умов точне рішення знайти неможливо. У силу цих причин необхідно розробити власний тип раціонального числа, що дозволяє зберігати досить велику кількість цифр, а операції над ними реалізувати самим, користаючись звичайними правилами арифметики. Таким чином, числа будуть масивами цифр, а обчислення стануть символьними. У цьому випадку успіх гарантований, якщо, звичайно, потужність комп'ютера й обсяг його пам'яті досить великі. Закон збереження труднощів ніхто скасувати не може!

Як бачимо, необхідні засоби для опису власних типів даних. У мові C++ передбачена така можливість, однак мова про неї піде трохи пізніше.

2.3. Специфікатори збереження

У залежності від виду зв'язування змінні в мові C++ підрозділяються на чотири категорії: статичні, автоматичні, динамічні і зовнішні.

2.3..1. Статичні змінні

Статичними називаються змінні, які зв'язуються з коміркою пам'яті на етапі компіляції і залишаються зв'язаними з нею аж до припинення виконання програми. Статичні змінні часто виявляються корисними. Яскравим прикладом є глобальні змінні, котрі доступні в будь-якій точці програми і, отже, повинні бути зв'язані

з однією і тією ж коміркою протягом усього часу виконання програми. Крім того, іноді буває зручно, щоб змінні, що оголошуються усередині функцій, зберігали своє значення між окремими викликами. У цьому випадку також необхідно, щоб статична змінна не змінювала адреси.

Ще однією перевагою статичних змінних є їхня ефективність. Статичні змінні допускають пряму адресацію, тоді як динамічні вимагають більш повільної непрямої адресації за допомогою вказівників і посилань. Крім того, створення і видалення статичних змінних не вимагає додаткових витрат часу.

Однак статичне зв'язування обмежує гнучкість програм. Зокрема, якби мова C++ містила лише статичні змінні, рекурсивні функції виявилися б неможливими.

Для того щоб оголосити статичну змінну, у мові C++ використовується специфікатор `static`. Врахуйте, що це слово в мові C++ багатозначне. По-перше, воно є спадщиною мови C и означає змінні, що зберігають свої значення між окремими викликами функціями. По-друге, цим ключовим словом позначають особливі члени класів, що належать відразу всім його об'єктам і жодному окремо. Таким чином, зміст специфікатора `static` залежить від його контексту.

Статичні змінні розділяються на локальні і глобальні. Локальна статична змінна, як і глобальна, не змінює адреси при виконанні програми. Однак локальна статична змінна, на відміну від глобальної, доступна лише усередині свого блоку чи функції, зберігаючи свої значення між викликами функції.

Проілюструємо застосування локальних статичних змінних наступною програмою. Вона виконує п'ять рекурсивних викликів функції `f()` і "катапультується" за допомогою стандартної функції `exit()`. При кожнім виклику функції на екран виводиться значення статичної змінної `counter` і адреса локальної змінної `var` (щораз новий!).

Використання локальних статичних змінних

```
#include <iostream.h>
#include <process.h>
void f();
int main()
{
 f();

 return 0;
}

void f()
{
 int var=1;
 static int counter = 0; // Ініціалізація локальної статичної змінної
 counter = counter + 1;
 if(counter > 5) exit(0);
 cout << "counter = " << counter;
 cout << " &var " << &var << " &counter = " << &counter << endl;
 f();
}
```

У результаті роботи цієї програми на екран будуть виведені приблизно такі рядки (адреси можуть бути іншими).

```
counter = 1 &var = 0x0065fdc0 &counter = 0x0041f018
counter = 2 &var = 0x0065fdb4 &counter = 0x0041f018
counter = 3 &var = 0x0065fda8 &counter = 0x0041f018
counter = 4 &var = 0x0065fd9c &counter = 0x0041f018
counter = 5 &var = 0x0065fd90 &counter = 0x0041f018
```

Якби б статичних змінних не було, для підрахунку викликів довелося б застосовувати глобальні змінні, породжуючи небажані побічні ефекти. Крім описаного вище прикладу, локальні статичні змінні виявляються корисними при генерації псевдовипадкових чисел, коли при кожному наступному виклику генератора необхідно задавати нове початкове значення.

Локальну статичну змінну можна ініціалізувати, задавши її початкове значення під час оголошення. Оскільки змінна є статичної, її початкове значення привласнюється лише один раз, під час оголошення.

Якщо глобальна змінна описана за допомогою специфікатора `static`, вона існує тільки в рамках поточного файлу. Це дозволяє створити захист від побічних ефектів. Наприклад, можна ізолювати усі функції, що використовують деяку глобальну статичну змінну, в окремому файлі, виключивши в такий спосіб можливість неконтрольованих побічних ефектів.

Справедливості заради варто зауважити, що в стандарті мови C++ ця особливість модифікатора `static` оголошена небажаною. Оскільки в мові з'явилися простори імен, що дозволяють уникнути конфліктів імен і запобігати неконтрольованих змін значень змінних, модифікатор `static` морально застарів.

2.3..2.3.Автоматичні змінні

Автоматичними називаються змінні, які з коміркою зв'язуються динамічно, тобто під час виконання операторів оголошення, а з типом — статично. Такі змінні іменуються автоматичними, тому що вони створюються і знищуються без утручання програміста.

У мові C++ усі локальні змінні за замовчуванням є автоматичними. За винятком адреси, всі атрибути автоматичних змінних зв'язуються статично. Виключення складають об'єднання, що у різні моменти часу можуть містити змінні різних типів і, отже, зв'язуються з типом динамічно.

Мова C++ містить ключове слово `auto`, яке можна використовувати для оголошення локальних змінних. Оскільки усі локальні змінні за замовчуванням вважаються автоматичними, це ключове слово вказувати необов'язково.

Автоматичні змінні є антиподами статичних. Вони виникають при вході в блок чи функцію, у якій вони описані, і знищуються при виході з неї. Таким чином, якщо при оголошенні автоматична змінна ініціалізується деяким значенням, це буде відбуватися заново при кожнім виклику функції. На противагу цьому ініціалізація статичної змінної відбувається лише один раз, і при наступних викликах функції оператор оголошення буде зігнорований. Підкреслимо, що це стосується лише оголошення статичної змінної. Якщо її початкове значення задається за допомогою звичайного оператора присвоювання, він буде виконуватися при кожнім виклику функції.

2.3..3. Динамічні змінні

Динамічні змінні не мають імен. Вони розташовуються в динамічній пам'яті, чи *купі*, і видаляються з неї під час виконання програми. Для звертання до цих змінних потрібні особливі засоби: вказівники і посилання. Динамічні змінні створюються або оператором `new`, або стандартною функцією (`malloc()`, `calloc()` чи `realloc()`). Знищення динамічних змінних здійснюється оператором `delete` (разом з оператором `new`) чи функцією `free()` (у сполученні з функціями `malloc()` , `calloc()` чи `realloc()`).

Оператор пам'яті `new` у якості операнда використовує ім'я типу. Результатом цього оператора є динамічна змінна, що має тип операнда, і вказівник, що зберігає її адресу. Динамічна змінна зв'язується з типом під час компіляції, тому це зв'язування є статичним. Втім, динамічні змінні зв'язуються з виділеною областю пам'яті в ході виконання програми, тому цей вид зв'язування є динамічним.

Як приклад розглянемо фрагмент програми, у якій створюється вказівник на динамічні змінні цілочисельного типу за допомогою оператора `new` і функцій `malloc()` і `calloc()`.

```
#include <iostream>
int main()
{
 int* var = new int // Зв'язуємо динамічну змінну з типом
 ...
 delete var;        // Звільняємо область пам'яті,
                    // зайняту динамічної змінної
 return 0;
}
```

У цьому фрагменті створюється динамічна цілочисельная змінна. Вона не має імені, а доступ до неї можливий лише через вказівник `var`. Наприкінці програми ця змінна видаляється з пам'яті оператором `delete`. Якщо цей оператор пропустити, динамічна змінна буде займати пам'ять аж до кінця виконання програми. Отже, якщо змінна більше не потрібна, неї варто видалити з динамічної пам'яті.

Розглянемо ще один приклад, у якому відбувається розміщення `num` символьних динамічних змінних, кожна з який за означенням займає 1 байт, і обчислення розміру вільної пам'яті до і після розміщення.

Обчислення розміру вільної пам'яті

```
#include <iostream.h>
#include <alloc.h>

int main()
{
 void *var=NULL;
 int num = 1;
 long size1, size2, diff;
 size1 = coreleft();
 cout << size1 << endl;
 var= new int[num];
 size2 = coreleft();
 cout << size2 << endl;
 diff = size1-size2;
 cout <<  diff  << endl;
```

```
 return 0;
}
```

Програма, наведена вище, має одну особливість. Мінімальний розмір динамічної пам'яті, що виділяється окремій символьній змінній, дорівнює не одному байту, як варто було б очікувати, а 16 байт (ця величина залежить від компілятора), причому 4 байт із них є службовими. Мінімальна область пам'яті, виділювана змінним, називається *параграфом*.

Цей ефект виявляється під час експериментів із програмою, коли задаються різні значення змінної num. Коли num змінюється в діапазоні від 1 до 12, масиву символьних змінних виділяється 16 байт. Це легко пояснити — 12 байт для символів, що займають по одному байті, плюс 4 службових байт. Однак як тільки число num стає рівним 13, одного параграфа уже достатньо, і масиву виділяється 32 байт. Якщо число num змінюється від 13 до 28, двох параграфів цілком достатньо, однак, починаючи з num, рівного 29, потрібний ще один параграф (у сумі вийде 48 байт) і т.д. Таким чином, *розміщаючи змінні в динамічній пам'яті, не варто економити*. Ефективність використання купи прямо залежить від розміру розміщених у ній даних. Чим більше розмір масиву, що записується в купу, тим менше відсоток "надлишкової" пам'яті, що виділяється, але не використовується.

2.3..4. Зовнішні змінні

Як правило, оголошення змінної одночасне є її визначенням. Однак указавши перед ім'ям змінної специфікатор extern, можна оголосити її, не визначаючи. Це дозволяє одержати доступ до змінної, описаної в іншій частині програми.

Розглянемо приклад. Нехай наша програма складається з двох файлів.

Зовнішня змінна

```
// Файл 1
#include <iostream.h>

int main()
{
 extern int ext_var;  // Зовнішня змінна
 cout << ext_var;

 return 0;
}

//Файл 2
 int ext_var=1;
```

У результаті на екрані з'явиться число 1. Специфікатор extern, зазначений усередині функції main(), означає, що змінна ext_var оголошено в іншому місці. Таким чином, ми можемо використовувати змінну ext_var до її визначення.

Одночасно з оголошенням зовнішньої змінної можна виконати її ініціалізацію. Однак ця можливість реалізується лише для глобальних змінних. У попередньому прикладі змінна ext_var була локальною. Отже, спроба ініціалізувати її виявилася б безуспішною. Це цілком природно, адже локальна змінна існує лише усередині блоку, де вона визначена. У свою чергу блок неможливо "розподілити" по декількох файлах. Не знайшовши визначення зовнішньої змінної усередині блоку, компілятор переходить до глобальних змінних. Ініціалізація локальної змінної усередині блоку чи функції ізолювала б її значення від іншої частини програми, і специфікатор extern утратив би зміст.

Ініціалізація локальної зовнішньої змінної

```
 // Ініціалізація локальної зовнішньої змінної неможлива
#include <iostream.h>

int main()
{
   extern int ext_var=2;  // Помилка: ext_var – локальня зовнішня
   cout << ext_var;

   return 0;
}
```

Специфікатор extern можна інтерпретувати як "далі буде". Це значить, що визначення змінної буде розміщено в іншім місці, а поки компілятор повинний дозволити її застосування. Зокрема, визначення зовнішньої змінної не обов'язково повинне розміщатися в іншому файлі, хоча це найпоширеніший варіант. Воно може знаходитися, наприклад, наприкінці програми.

Ініціалізація локальної зовнішньої змінної

```
// Ініціалізація локальної зовнішньої змінної після оголошення
#include <iostream.h>

int main()
{
 extern int ext_var;  // Оголошення зовнішньої змінної (далі буде)
 cout << ext_var;     // Використання зовнішньої змінної

 return 0;
}
int ext_var = 0; // Визначення змінної ext_var
```

2.3..5. Реєстрові змінні

Специфікатор `register` дозволяє зберігати значення змінної в регістрі центрального процесора, а не в пам'яті. Це набагато підвищує швидкість виконання операцій над цією змінною.

Чи зберігати змінну в регістрі, компілятор вирішує сам. Отже, це навіть не інструкція, а просто рекомендація, що впливає на процес оптимізації коду. У регістрах центрального процесора, як правило, зберігаються символи і цілі числа, що інтенсивно використовуються програмою. Яскравим прикладом такої змінної є лічильник циклу. Проілюструємо його на прикладі обчислення факторіала.

Специфікація register

```
#include <iostream.h>
int main()
{
 register int factorial=5;
 for(register n=4; n>=0; n-1) factorial = factorial * n;

 return 0;
}
```

Специфікатор `register` можна застосовувати тільки до локальних змінних і формальних параметрів функцій. До глобальних змінних він не застосовується.

Як відомо, у мові С неможливо обчислити адреса реєстрової змінної за допомогою оператора &, оскільки реєстрові змінні зберігаються, як правило, у процесорі, а його пам'ять не адресується. У той же час у С++ обчислення адреси реєстрової змінної допускається.

2.3..6. Простори імен

З концепцією області видимості тісно зв'язане поняття *простору імен*. Простори імен призначені для локалізації імен ідентифікаторів і запобігання їхніх конфліктів. При програмуванні на С++ використовується велика кількість бібліотек, що містять визначення констант, функцій і класів.

Ключове слово `namespace` дозволяє розв’язати ці проблеми. Якщо в програмі не визначений жодний простір імен, то імена всіх сутностей знаходяться в глобальному просторі імен і можуть конфліктувати одне з одним. Ключове слово `namespace` розділяє глобальний простір імен на декларативні області і локалізує область видимості об'єктів, оголошених усередині нього.

Власне кажучи, простір імен являє собою іменовану область видимості.

```
namespace ім'я
{
 // Оголошення і визначення
 }
```

Ідентифікатори, оголошені усередині поточного простору імен, можна використовувати як звичайно. Для посилань на об'єкти, що знаходяться поза цим простором, необхідно застосовувати оператор розв’язування області видимості і вказувати ім'я простору імен.

Оголошення і визначення об'єктів не обов'язково впроваджувати в простір імен. Якщо вони розташовані за межами простору, перед їх іменами також необхідно вказати ім'я простору імен і оператор розв’язування області видимості.

```
namespace name
{
  double aDouble;
}
int name::anInt=1;
name::aDouble=123.45;
```

У цьому фрагменті обидві змінні `aDouble` і `anInt` належать до простору імен `name`, лише змінна `aDouble` оголошена і визначена безпосередньо в просторі `name`, а змінна `anInt` — поза ним.

Досить нудно вказувати ім'я простору імен перед кожної змінної, яка використовується за його межами, і застосовувати оператор розв'язування області видимості. Для того щоб уникнути цього, у мові C++ передбачене оголошення `using`, що має наступний вид.

```
using namespace name;
using name::member;
```

У першому варіанті ім'я `name` задає назву простору імен. Всі елементи цього простору стають частиною поточної області видимості і можуть використовуватися без указівки кваліфікатору. В другому варіанті в область видимості включається тільки конкретний елемент простору імен.

Наступна програма ілюструє застосування оголошення простору імен `using`.

Простір імен і директива using: перший варіант

```
// Демонстрація простору імені й оголошення using
#include <iostream.h>

// У просторі імен name оголошені дві цілочисельні змінні
namespace name
{
   int member1=1;
   int member2=2;
}

int main()
{
 // Використовується тільки член member1
 using name::member1;

 // Тепер кваліфікатор не потрібний
 cout << "name::member1 = " << member1 << endl;

 // Ця змінна member2 належить глобальному простору імен
 int member2 = 1000;
 cout << "member2       = " << member2 << endl;
 // Перед змінної member2 із простору name потрібний кваліфікатор
 cout << "name::member2 = " << name::member2 << endl;

 // Тепер доступний весь простір імен name
 using namespace name;
 member1=1000;
 member2=2000;
 cout << "name::member1 = " << member1 << endl;
 cout << "name::member2 = " << member2 << endl;

 return 0;
}
```

У результаті на екран будуть виведені наступні рядки.

```
  name::member1 = 1;
member2       = 1000;
name::member2 = 2;
name::member1 = 1000;
name::member2 = 2000;
```

Зверніть увагу на те, що простору імен, оголошені за допомогою директиви `using`, не створюють конфлікту імен. Змінні, оголошені в новому просторі імен, просто додаються в поточну область видимості, маскуючи однойменні змінні. Таким чином, наприкінці програми доступними стають як глобальної простір імен, так і простір імен `name`.

Щоб проілюструвати сказане, злегка змінимо програму.

Простір імен і директива using: другий варіант

```
// Демонстрація простору імені й оголошення using
#include <iostream.h>

// У просторі імен name оголошені дві цілочисельні змінні
namespace name
{
 int member1=1;
 int member2=2;
}
```

```
int main()
{
 // Використовується тільки член member1
 using name::member1;

 // Тепер кваліфікатор не потрібний
 cout << "name::member1 = " << member1 << endl;

 // Ця змінна member2 належить глобальному простору імен
 int member2 = 1000;
 cout << "member2       = " << member2 << endl;
 // Перед змінною member2 із простору name потрібний кваліфікатор
 cout << "name::member2 = " << name::member2 << endl;

 // Тепер доступний весь простір імен name
 using namespace name;
 member1=1000;
 // member2=2000;
 cout << "name::member1 = " << member1 << endl;
 // Ця змінна member2 належить до глобального простору імен
 cout << "name::member2 = " << member2 << endl;

 return 0;
}
```

Нові результати виглядають у такий спосіб.

```
name::member1 = 1;
member2       = 1000;
name::member2 = 2;
name::member1 = 1000;
name::member2 = 1000;
```

Існує особливий різновид просторів імен — *неіменовані простори імен*. Їхнє оголошення має наступний вид.

```
namespace
{
 // Оголошення і визначення
}
```

Неіменовані простори імен дозволяють створювати унікальні ідентифікатори, область видимості яких обмежена файлом. Поза файлом ці змінні вважаються невидимими.

Неіменований простір імен аналогічний специфікатору `static`, застосованому до глобальній змінній. Як і специфікатор `static`, вона обмежує область видимості глобального імені межами файлу. Розглянемо два файли, що є частиною однієї і тієї ж програми.

Доступ до зовнішньої змінної

```
// Файл 1
#include <iostream.h>

int main()
{
 extern int ext_var;  // Зовнішня змінна
 cout << ext_var; // Помилка! Звертання до недоступної змінної!

 return 0;
}

//Файл 2
// змінна ext_var невидима поза цим файлом
static int ext_var=1;
```

Аналогічного ефекту можна домогтися за допомогою неіменованого простору імен.

Неіменований простір імен

```
// Файл 1
#include <iostream.h>

int main()
{
 extern int ext_var;  // Зовнішня змінна
 cout << ext_var; // Помилка! Звертання до недоступної змінної!
```

```
  return 0;
}

//Файл 2
// змінна ext_var невидима поза цим файлом
namespace
{
 int ext_var=1;
}
```

Те саме простір імен можна повідомляти декілька разів. Наприклад, в одному з попередніх прикладів змінні member1 і member2 можна було б оголосити в такий спосіб.

Багаторазове оголошення простору імен

```
// Демонстрація простору імені й оголошення using
#include <iostream.h>

// У просторі імен name оголошені дві цілочисельні змінні
namespace name
{
 int member1=1;
}
namespace name
{
 int member2=2;
}

int main()
{
 // Використовується тільки член member1
 using name::member1;

 // Тепер кваліфікатор не потрібний
 cout << "name::member1 = " << member1 << endl;

 // Ця змінна member2 належить глобальному простору імен
 int member2 = 1000;
 cout << "member2       = " << member2 << endl;

 // Перед змінною member2 із простору name потрібний кваліфікатор
 cout << "name::member2 = " << name::member2 << endl;

 // Тепер доступний весь простір імен name
 using namespace name;
 member1=1000;
 member2=2000;
 cout << "name::member1 = " << member1 << endl;
 // Ця змінна member2 відноситься до глобального простору імен
 cout << "name::member2 = " << member2 << endl;

 return 0;
}
```

Результати роботи цієї програми збігаються з попередніми.

```
name::member1 = 1;
member2       = 1000;
name::member2 = 2;
name::member1 = 1000;
name::member2 = 2000;
```

Простір імен повинен бути оголошений поза усіх областей видимості. Повідомляти простір імен усередині функції не можна. У той же час простору імен можуть бути вкладеними. Проілюструємо сказане наступною програмою.

Вкладені простори імен

```
// Демонстрація простору імені й оголошення using
#include <iostream.h>
```

```
// У просторі імен nameOuter оголошена цілочисельна змінна
// і вкладений простір імен
namespace nameOuter
{
 int member1=1;
 // У просторі імен nameInner оголошені дві цілочисельні змінні
 namespace nameInner
 {
    int member2=2;
 }
}

int main()
{
 // Використовується тільки член member1
 using nameOuter::member1;

 // Тепер кваліфікатор не потрібний
 cout << "nameOuter::member1 = " << member1 << endl;

 // Ця змінна member2 належить глобальному простору імен
 int member2 = 1000;
 cout << "member2 = " << member2 << endl;

 // Перед змінною member2 із простору name потрібні кваліфікатори
 cout << "nameOuter::nameInner::member2 = "
      << nameOuter::nameInner::member2 << endl;

 // Тепер доступний весь простір імен name
 using namespace nameOuter;
 member1=1000; // Звертання до простору імен nameOuter
 nameInner::member2=2000; // Звертання до простору імен nameInner
 cout << "nameOuter::member1 = " << member1 << endl;
 // Ця змінна member2 належить до глобального простору імен
 cout << "nameInner::member2 = " << nameInner::member2 << endl;

 return 0;
}
```

Результати роботи цієї програми тепер виглядають так.

```
nameOuter::member1 = 1;
member2 = 1000;
nameOuter::nameInner::member2 = 2;
nameOuter::member1 = 1000;
nameOuter::nameInner::member2 = 2000;
```

Простір імен `nameInner` вкладений в простір імен `nameOuter`. Отже, для доступу до змінного `member2` із простору імен `nameInner` необхідно вказувати два кваліфікатору зовнішній і внутрішній простори імена. Після виконання оголошення

```
using namespace nameOuter;
```

ім'я зовнішнього простору можна пропустити і вказати лише ім'я внутрішнього простору імен: `nameInner::member2`.

2.4. Кваліфікатори доступу

У мові C++ існує два кваліфікатори, що керують доступом до змінного і модифікацією: `const` і `volatile`. Їх називають *cv-квалификаторами.*

2.4..1. Константи

Якщо перед оголошенням змінної стоїть кваліфікатор `const`, це значить, що вона не може змінюватися. Отже, єдиний спосіб привласнити їй яке-небудь значення — ініціалізувати її під час оголошення. Розглянемо приклад.

```
const int const_var=1;
```

Тут з'являється цілочисельна змінна `const_var`, значення якої дорівнює 1. Це значення надалі змінити неможливо.

Найчастіше кваліфікатор `const` застосовують для запобігання змін аргументів функції. Наприклад, якщо функція одержує константний вказівник на деяке значення, змінити це значення буде неможливо.

Константний вказівник

```
#include <iostream.h>

void print(const int *p);

int main()
{
 int var=10;
 const int *p=&var;
 print(p);

 return 0;
}

void print(const int *p)
{
 *p=20; // Неможливо! Значення var захищене кваліфікатором const
 cout << *p;
}
```

Якщо видалити з функції `print()` оператор, що намагається модифікувати значення змінної, на яку посилається вказівник `p`, на екрані з'явиться число 10.

Ще одним призначенням кваліфікатора `const` є захист звичайних змінних від модифікації. Це дозволяє запобігти як ненавмисні помилки при програмуванні, так і небажану модифікацію змінної ззовні, наприклад, через мережу.

Константні змінні

```
#include <iostream.h>

int main()
{
 const int var=10;
 var = var + 1;  // Помилка!

 return 0;
}
```

2.4.Мінливі сутності

Деякі сутності можуть змінюватися неявно, наприклад, операційною системою, апаратним забезпеченням або рівнобіжним потоком керування. Такі змінні відзначають за допомогою кваліфікатора `volatile`. У цьому випадку вміст змінної змінюється без допомоги оператора присвоювання. як правило, компілятори мови C++ вважають, що змінні, що ніколи не зустрічаються в лівій частині оператора присвоювання, є константами і намагаються оптимізувати програму, змінюючи порядок обчислень. Крім того, компілятор може перемістити таку змінну в область швидкого доступу, наприклад у регістри процесора. Кваліфікатор `volatile` запобігає таким змінам, оскільки оптимізована програма може не відповідати очікуванням її автора. Оголошення мінливої змінної виглядає в такий спосіб.

```
int volatile var;
```

Кваліфікаторы `const` і `volatile` можна використовувати одночасно. Припустимо, що програма одержує ззовні деяке значення, наприклад, через порт. Тоді значення, що надходить через порт, можна захистити кваліфікатором `const`. Це не скасовує можливості модифікувати цю змінну зовнішніми засобами, але захищає її від внутрішніх зазіхань.

```
const volatile int val_port = (const volatile int) 10;
```

Тепер змінна `val_port` може змінюватися лише зовнішніми засобами, але жодний оператор програми змінити її не зможе.

Одночасне застосування кваліфікаторов const і volatile

```
#include <iostream.h>
const volatile int var = (conat volatile int) 10;

int main()
{
 var = 20; // Помилка!
 cout << var << endl;

 return 0;
}
```

2.5. Приведення типів

Повернемося до програми, що ілюструє переповнення змінної типу `char`.

Приведення типу

```
#include <iostream.h>

int main()
{
 char c = 127;
    c=c+1;
 cout << (int)c << endl;

 return 0;
}
```

Ця програма виводить на екран число `-128`. Для того щоб вивести це число, нам знадобилося перетворити символ у ціле число. Ця процедура називається *приведенням типу*.

2.5..1. Приведення типу в стилі мови С

Класична схема приведення типу має двох різновидів.

(*новий_тип*)<*операнд_старого_типу*>

новий_тип(*операнд_старого_типу*)

У якості *операнда* старого типу може виступати будь-яке окреме змінна чи вираз. Розглянемо часто зустрічається ситуацію.

Целочисельне ділення

```
#include <iostream.h>

int main()
{
 int n = 1, m = 2;
 double x;
 x= n/m;
 cout << x << endl;

 return 0;
}
```

Багато хто з подивом виявляють, що при діленні `n` на `m` змінна `x` дорівнює 0. Уся справа в тім, що результат розподілу цілого числа на інше ціле число є цілим числом. Спроба привести результат розподілу до типу `double` виявляється безрезультатною.

Приведення результату цілочисельного ділення

```
#include <iostream.h>

int main()
{
 int n = 1, m = 2;
 double x;
 x= (double)(n/m);
 cout << x << endl;

 return 0;
}
```

У результаті замість цілочисельного нуля ми одержимо нуль типу `double`. Щоб виправити ситуацію, необхідно привести до типу `double` або дільник, або ділене, лише в цьому випадку результатом ділення буде правильне число типу `double`.

Приведення чисельника: перший варіант

```
#include <iostream.h>

int main()
{
 int n = 1, m = 2;
 double x;
 x= (double)n/m;
 cout << x << endl;
```

```
  return 0;
}
```

Ще один спосіб приведення типів виглядає так.

Приведення чисельника: другий варіант

```
#include <iostream.h>

int main()
{
 int n = 1, m = 2;
 double x;
 x= double(n)/m;
 cout << x << endl;

 return 0;
}
```

У мові C++ є ще чотири способи приведення типів — оператори `static_cast`, `const_cast`, `dynamic_cast` і `reinterpret_cast`. Їхньої схеми виглядають так.

`static_cast<`*`тип`*`>(`*`вираз`*`)`
`const_cast<`*`тип`*`>(`*`вираз`*`)`
`dynamic_cast<`*`тип`*`>(`*`вираз`*`)`
`reinterpret_cast<`*`тип`*`>(`*`вираз`*`)`

Усі згадані вище форми приведення типів рівноправні, однак в об'єктно-орієнтованому програмуванні більш коректними є наступні чотири оператори.

2.5.Оператор static_cast

Розглянутий оператор дуже нагадує звичайне приведення в стилі мови С. Вираз
`(`*`новий_тип`*`)<`*`вираз_старого_типу`*`>`
і
`static_cast<`*`новий_тип`*`>(`*`вираз_старого_типу`*`)`
еквівалентні. Наприклад, попередгю програму можна привести до більш сучасного виду.

Застосування оператора static_cast

```
include <iostream.h>

main()
{
 int n = 1, m = 2;
 double x;
 x= static_cast<double>(n)/m;
 cout << x << endl;
}
```

2.5..3. Оператор const_cast

Цей оператор призначений для роботи з атрибутами `const` і `volatile`. Найчастіше він застосовується для скасування атрибута `const`. Новий тип повинний збігатися з вихідним, за винятком атрибутів `const` і `volatile`. Загальний вид оператора `const_cast` приведений нижче.

`const_cast<`*`тип`*`>(`*`вираз`*`);`

Тут параметр *тип* задає результуючий тип приведення, а параметр *вираз* є виразом, результат якого приводиться до нового типу.

Розглянемо програму, що демонструє застосування оператора `const_cast`.

Оператор const_cast

```
// Застосування оператора const_cast до константного посилання
#include <iostream.h>

int main()
{
 int x = 10;
 const int  &new_x=x;
 cout << "Значення x перед const_cast = " << x << endl;
 //  new_x=2.0*x; — Це помилка, тому що new_x — константне посилання
 const_cast<int &> (new_x) =  2*x;
 //  new_x=2.0*x; — Це помилка, тому що new_x — константне посилання
 cout << "Значення x після const_cast =: " << x << endl;
```

```
  return 0;
 }
```

Програма обчислює наступні результати.

```
Значення x перед const_cast = 10
Значення x після const_cast = 10
```

Необхідно відзначити, що оператор `const_cast` лише *тимчасово* видаляє атрибут `const` у посилання, і уже в наступної терміну присвоєння стає знову неможливим.

Оскільки застосування оператора `const_cast` відкриває можливість модифікації константних величин, його варто використовувати дуже обережно. Наприклад, компілятор Visual C++ 6.0 не дозволяє знімати атрибут `const` у змінних убудованих типів, за винятком вказівників і посилань. Наступна, цілком безневинна програма викликала помилку.

Застосування оператора const_cast до константної змінної

```
  // Застосування оператора const_cast до константної змінної
#include <iostream.h>

int main()
{
  const int  x=10;
  const_cast<int> (x) = 20; // Привести тип const int до типу int неможливо
  cout << "Значення x після const_cast: " << x << endl;
  return 0;
}
```

Таким чином, працюючи з оператором `const_cast`, ми не зможемо перетворити константи убудованих типів у звичайні змінні навіть на короткий час.

2.5..4. Оператор dynamic_cast

Даний оператор здійснює динамічне приведення типу з наступною перевіркою коректності приведення. Якщо воно виявилося некоректним, дане приведення не виконується. Загальний вид оператора `dynamic_cast` такий.

```
dynamic_cast <нові_тип> (вираз);
```

Новий тип повинний бути вказівним чи посилальним, а вираз, що приводиться, повинний обчислювати вказівник чи посилання.

Як правило, оператор `dynamic_cast` використовується для приведення класів, що утворюють ієрархії спадкування.

2.5..5. Оператор reinterpret_cast

За допомогою оператора `reinterpret_cast` один тип перетворюється в зовсім іншій. Наприклад, він може перетворити вказівник у ціле число, а ціле число — у вказівник. Оператор `reinterpret_cast` має наступний вигляд.

```
reinterpret_cast<новий_тип>(вираз)
```

Цей процес можна подати в такий спосіб. Оскільки кожен тип характеризується розміром змінної і набором операцій, спочатку змінна перетворюється в простий ланцюжок байтів, що з цього моменту інтерпретується інакше.

Розглянемо приклад, що демонструє застосування оператора `reinterpret_cast`.

Застосування оператора reinterpret_cast

```
// Демонстрація оператора reinterpret_cast
#include <iostream.h>

int main()
{
 int i;
 int *p=&i;
 cout <<"Адреса змінної i, записана у вказівнику p =" << p << endl;

 i = reinterpret_cast<int> (p); // Приведення вказівника
                                // до цілого числа
 cout << "Десяткове подання адреси змінної i" << i << endl;
 cout << "Шістнадцятирічне подання адреси змінної i"
      <<  hex <<i << endl;
```

```
 return 0;
}
```

Оператор `reinterpret_cast` перетворить вказівник `p` у ціле число, що являє собою адресу змінної `i` у десятковій системі числення. Потім програма виводить це число в шістнадцятирічному вигляді.

```
0x0065FDF4
6684148
65fdf4
```

Як бачимо, шістнадцятирічне значення змінної і збігається з її адресою. Отже, приведення вказівника до цілого типу відбулося успішно.

2.6. Вказівники і посилання

Вказівник — це змінна, у якій зберігається адреса комірки пам'яті чи нульова адреса. Як правило, нульова адреса задається константою `NULL`. Він застосовується в ситуаціях, коли вказівник не містить жодної реальної адреси, як звичайно говорять, “ні на що не посилається”.

Вказівники являють собою багатоцільовий механізм. По-перше, вони забезпечують непряму адресацію змінних, надаючи альтернативний доступ до їхніх значень. По-друге, (головне!) вони дозволяють керувати розподілом *динамічної пам'яті*, чи *купи*.

Як відомо, звичайні іменовані змінні розташовуються або в стеку, якщо вони є локальними, або в окремому сегменті пам'яті, призначеному для збереження глобальних змінних. Обидва види цих змінних називаються *статичними*, оскільки пам'ять для них виділяється на етапі компіляції. Інакше кажучи, зв'язування їх імен з адресами є статичним.

Змінні, розташовувані в купі, називаються *динамічними*. Як правило, вони є *безіменними*, а доступ до них забезпечується чи вказівниками посиланнями.

2.6..1. Вказівники

Тип вказівника визначається типом змінної, на яку він посилається. Отже, кількість типів вказівників збігається з кількістю існуючих типів змінних.

```
char* pChar;            // Вказівник на символ
int* pInt;              // Вказівник на цілочисельну змінну
unsigned* pUnInt;       // Вказівник на цілочисельну змінну без знака
long int* pLongInt;     // Вказівник на довгу цілочисельну змінну
float* pFloat;          // Вказівник на змінну типу float
double* pDouble;        // Вказівник на змінну типу double
long double* pLDoubel;  // Вказівник на змінну типу long double
```

Для вказівників передбачені дві основні операції: *присвоювання* і розіменування.

Операція присвоювання використовується для зміни значення вказівника. Якщо вказівник застосовується для доступу до комірки динамічної пам'яті, його значення обчислюється за допомогою механізмів керування купою. Існує два різновиди цього механізму: успадкований від мови C (функції `malloc()`, `realloc()`, `calloc()` і `free()`) і новий об’єктно-орієнтований спосіб (оператори `new` і `delete`).

Якщо вказівник використовується як засіб непрямої адресації статичних змінних, для одержання адреси змінної застосовується оператор узяття адреси &.

Оператор узяття адреси

```
#include <iostream.h>

int main()
{
 int var=1;           // Оголошення змінної var
 int *pVar = &var;    // Оголошення й ініціалізація вказівника pVar
 cout << "Значення вказівника pVar  = " << pVar << endl;
 cout << "Адреса вказівника pVar    = " << &pVar << endl;
 cout << "Значення змінної var      = " << *pVar << endl;

 return 0;
}
```

Проаналізуємо результати роботи цієї програми (адреси можуть бути іншими).

```
Значення вказівника pVar  = 0x8fa50ffe
Адреса вказівника pVar    = 0x8fa50ffa
Значення змінної var      = 1
```

Зверніть увагу на те, що вказівник розміщається в окремій крмірці. Отже, його використання приводить до додаткових витрат пам'яті. Крім того, як бачимо, оператор узяття адреси дозволяє ідентифікувати комірки, у яких написані змінні. Оскільки в даному випадку всі змінні є локальними, вони розташовуються в стеку функції

main() за принципом LIFO (“last in, first out” — “останнім увійшов, першим вийшов”). Це виражається у тому, що адреси змінних, оголошених одна за іншою, убувають. Продемонструємо цей факт наступним прикладом.

Адресація локальних змінних: перший варіант

```
#include <iostream.h>

int main()
{
  int a, b, c;
  cout << &a << " " << (int)&a << endl;
  cout << &b << " " << (int)&b << endl;
  cout << &c << " " << (int)&c << endl;
  return 0;
}
```

На екрані з'являться наступні числа (у першому стовпчику — шістнадцятирічні адреси змінної, у другому — їхні десяткові еквіваленти).

```
0x8faa0ffe 4094
0x8faa0ffc 4092
0x8faa0ffa 4090
```

З огляду на, що розмір стеку за замовчуванням дорівнює 4096, а розмір типу float — 2 байт, легко прогнозувати результати роботи злегка зміненого варіанта програми.

Адресація локальних змінних: другий варіант

```
#include <iostream.h>

int main()
{
 float a, b, c;
 cout << &a << " " << (int)&a << endl;
 cout << &b << " " << (int)&b << endl;
 cout << &c << " " << (int)&c << endl;
 return 0;
}
```

Як і слід було очікувати, ми одержуємо наступні результати.

```
0x8faa0ffc 4092
0x8faa0ff8 4088
0x8faa0ff4 4084
```

Розмір вказівника залежить від розміру комірки і може бути дорівнює 2 чи 4 байт. Припустимо, що мінімальна комірка пам'яті комп'ютера складається з 16 біт. У такому випадку, якщо вказівник посилається на цілочисельну змінну, витрати пам'яті зростають удвічі (за умови, що інших змінних немає), якщо використовується вказівник на змінну типу float — на 50%, якщо на змінну типу double — на 25%. Напрошується висновок: щоб виправдати застосування вказівників, варто адресувати великі ділянки пам'яті. Саме з цієї причини найчастіше вказівники застосовуються для адресації масивів і структур. Крім того, вказівники дозволяють модифікувати аргументи функції, але цю тему ми розглянемо пізніше.

Отже, розібравши присвоювання, перейдемо до розіменування. Сам по собі вказівник нічого не вартий, поки ми не можемо з його допомогою одержати чи змінити значення змінної, на яку він посилається. Для цього в мові передбачена операція розіменування *. Для того щоб розіменувати вказівник, досить перед його ім'ям поставити зірочку.

```
*pVar = 10;
cout << "Значення змінної var   = " << *pVar << endl;
```

Вказівники є дуже могутнім і широко розповсюдженим механізмом, тому всі області їхнього застосування в одному розділі охопити неможливо. З цієї причини ми розглянемо їх в окремій главі.

Відзначимо ще декілька особливостей вказівників, що ми будемо використовувати надалі. По-перше, оскільки вказівники можуть зберігати довільні адреси, ніщо не заважає їм посилатися на інші вказівники. Для того щоб оголосити вказівник, що посилається на інший вказівник, необхідно вказати його тип і поставити перед його ім'ям другу зірочку.

```
int **p; // Вказівник на вказівник цілочисельного типу
```

Ця властивість відкриває можливість створювати ланцюжка посилань. як правило, більше двох рівнів непрямої адресації не використовується, оскільки в практичних додатках цього цілком достатньо. Однак варто мати на увазі, що кількість цих рівнів практично нічим не обмежено. Непряма адресація вказівників дозволяє ігнорувати тип змінних, на які вони посилаються. Адже вказівники, що адресують вказівники, завжди посилаються на осередки фіксованого розміру (2 чи 4 байт), тому їх можна вільно змінювати місцями.

По-друге, вказівники можуть містити адреси не тільки статичних, але і динамічних змінних. Для цього їх необхідно ініціалізувати за допомогою функцій malloc() чи calloc() або оператора new.

Для ілюстрації повернемося до програми, наведеної в розділі 2.3..3, але тепер ми будемо не тільки контролювати розмір вільної пам'яті, але і виводити на екран адреси динамічних змінних.

Керування вільною пам'яттю

```
#include <iostream.h>
#include <alloc.h>
#include <stdlib.h>

int main()
{
 int *var=NULL;
 int size = sizeof(wchar_t);
 cout << "size = " << size << endl;
 long size1, size2, diff;
 unsigned long first, second;
 size1 = coreleft();
 cout << "size1 = " << size1 << endl;

 var= new int;

 size2 = coreleft();
 cout << "size2   = " << size2 <<endl;
 first = (unsigned long int)var;
 cout << "Адреса  =" << var << " " << first << endl;
 diff = size1-size2;
 cout <<"Diff     =" << diff << endl;

 size1 = coreleft();
 cout << "size1   = " << size1 << endl;

 var= new int;

 size2 = coreleft();
 cout << "size2 = " << size2 << endl;
 diff = size1-size2;
 second = (unsigned long int)var;
 cout <<"Адреса   = " << var << " " << second << endl;
 cout <<"Diff     = " << diff << endl;
 cout <<"Різниця адрес = " << second-first;

 return 0;
}
```

Розглянемо результати.

```
size1  = 59936
size2  = 59920
Адреса = 0x91550004 2438725636
Diff   = 16
size1  = 59920
size2  = 59904
Адреса = 0x91560004 2438791172
Diff   = 16
Різниця адрес = 65536
```

Як бачимо, кожній змінній, як і раніше виділяється цілий параграф, розміром 16 байт, причому розміщаються динамічні змінні не поруч, а в різних сегментах (саме про це свідчить різниця між їх адресами, рівна 65536).

Спробуйте виконати наступну програму, а потім модифікуйте її, змінюючи тип динамічної змінної. Ви переконаєтеся, що різниця між адресами двох послідовно створених динамічних змінних завжди дорівнює 65536.

Обчислення різниці між адресами змінних

```
#include <iostream.h>

int main()
{
  double *a = new double;
  double *b = new double;
  double *c = new double;
```

```
  cout << a << " " << (unsigned long int)a << endl;
  cout << b << " " << (unsigned long int)b <<" "
       <<(unsigned long int)b-(unsigned long int)a << endl;
  cout << c << " " << (unsigned long int)c <<" "
       <<(unsigned long int)c-(unsigned long int)b << endl;
  return 0;
}
```

Особливий інтерес у мові C++ викликають вказівники типу `void*`, що можуть посилатися на змінні будь-яких типів. Ці вказівники неможливо розіменувати, оскільки немає способу визначити, скільки байтів для цього необхідно витягти з пам'яті. Зазвичай вказівники типу `void*` використовуються для роботи з "голими" байтами, коли зовсім неважно, якого типу дані зберігаються в зазначеній області пам'яті. Крім того, вказівники цього типу можна перетворити, настроївши їх на конкретний тип. Зворотне перетворення, наприклад `int*` у `void*`, неможливе, оскільки зовсім очевидно, що при цьому відбудеться втрата інформації.

Вказівники типу void*

```
#include <iostream.h>
#include <alloc.h>

int main()
{
 void *p = malloc(sizeof(int)); // Виділяємо два байти
 *(int*) p=1;   // Записуємо в динамічну змінну число 1,
                // використовуючи приведення типу
 cout << *(int*)p;

 return 0;
}
```

Зверніть увагу на декілька можливих помилок.

Демонстрація розповсюджених помилок

```
#include <iostream.h>
#include <alloc.h>

int main()
{
 void *p = malloc(sizeof(int)); // Виділяємо два байти
 *p=1;                          // Неприпустимий тип!
 int *q = malloc(sizeof(int));  // Виділяємо два байти
 (void*)q=1;                    // Неприпустиме приведення
 cout << *(int*)q;

 return 0;
}
```

Існує ще один дуже важливий різновид вказівників — константні. Для опису такого вказівника використовується ключове слово `const`. Цей кваліфікатор забороняє модифікувати змінну. Однак для вказівника ситуація небагато ускладнюється, оскільки правильного самотнього вказівника не існує — він завжди зв'язаний з якою-небудь коміркою. (У противному випадку виникає "вказівник, що висить".) Навіть коли вказівник містить нульову адресу, говорять, що він посилається на нульову, тобто недоступну комірку. Таким чином, можна виділити декілька різновидів вказівників: звичайний вказівник, що посилається на звичайний осередок; звичайний вказівник, що посилається на константу; константний вказівник, що посилається на змінну; константний вказівник, що посилається на константу.

```
char* pChar="Рядок";              // Звичайний вказівник на звичайний рядок
const char* pCharConst = "Рядок"; //Звичайний вказівник на константний рядок
char* const pConstChar = "Рядок"; // Константний вказівник на звичайний рядок
const char* const pConstCharConst = "Рядок"; // Константний вказівник
                                             // на константний рядок
*pChar = '\n';               // Дозволений
*pCharConst = '\n';          // Помилка — вказівник на константу
*pConstChar = '\n';          // Дозволений  — константний вказівник
                             // на звичайний рядок
*pConstCharConst = '\n';     // Помилка  —  константний вказівник
                             // на константний рядок
pConstChar = pCharConst;     // Помилка —  константний вказівник
                             // не можна змінювати
pConstCharConst = pCharConst; // Помилка — константний вказівник
                              // не можна змінювати
```

Розпізнати константні вказівники і вказівники на константу досить просто. Для цього досить подумки змістити зірочку до найближчого слова. Якщо це слово виявилося ідентифікатором — це звичайний вказівник, а якщо зірочка примикає праворуч до ключового слова `const` — константний.

Втім, захист константного вказівника і змінних досить легко перебороти, навіть не застосовуючи оператор приведення `const_cast`. Механізм, що дозволяє цієї зробити, називається *непрямою модифікацією*.

Непряма модифікація

```
#include <iostream.h>

int main()
{
 const int var = 1;
 const int* const pVar = &a;
 cout << "Константа var до модифікації     = " << endl;
 cout << "Значення *pa до модифікації      = " << endl;
 *(int *)&a=100; // Непряма модифікація
 cout << "Адреса змінної var               = " << endl;
 cout << "Значення вказівника pVar         = " << endl;
 cout << "Константа var після модифікації  = " << endl;
 cout << "Значення *pa після модифікації   = " << endl;
   return 0;
}
```

У результаті значення константи не змінюється, але значення розіменованого константного вказівника, що посилається на неї, стає іншим. Зверніть увагу на те, що вказівник `pVar`, як і раніше, зберігає адресу константи `var`, от тільки його розіменування тепер не має до неї стосунку. Константа `var` завжди дорівнює 1, а значення `*pVar` стало рівним 100.

```
Константа var до модифікації     = 1
Значення *pa до модифікації      = 1
Адреса змінної var               = 0x0065FDF4
Значення вказівника pVar         = 0x0065FDF4
Константа var після модифікації  = 1
Значення *pa до модифікації      = 100
```

Для того щоб знову “синхронізувати” константу `var` і вказівник на неї, необхідно виконати наступний оператор.

```
*(int *)&a=a;
```

На жаль, вказівники настільки могутні, що користатися ними небезпечно. Зокрема, механізм непрямої модифікації наочно демонструє, що вказівники дозволяють обходити навіть такий захист, як модифікатор `const`. Необмежений доступ до всіх комірок пам'яті, що відкривають вказівники, таїть у собі чимало погроз. Деякі програмісти настільки побоюються вказівників, що навіть розробили мову Java, що дозволяє цілком виключити їхнє використання.

Одна з можливих проблем зв'язана з виникненням *вказівників,що висять*, які містять адресу динамічної змінної, раніше уже вилученої з пам'яті. Це може привести до різних неприємностей. Наприклад, звільнена комірка пам'яті, на яку посилається вказівник, може виявитися зайнятою новою динамічною змінною. Якщо тип нової змінної виявиться колишнім, це ще невелика неприємність. Однак якщо змінна належить до іншого типу, може трапитись катастрофа. Справа в тім, що тип вказівника визначає кількість байтів, що займає змінна. Коли значення вказівника змінюється на одиницю, він переноситься на наступну за областю пам'яті комірку, зайняту змінною. Інакше кажучи, якщо вказівник посилався на змінну типу `int`, він буде перенесений на дві комірки уперед, а якщо на змінну типу `float` — на 4 комірки. Те ж стосується операції віднімання від вказівника цілого числа. Узагалі говорячи, арифметика вказівників не багата операціями. До вказівників можна додавати цілі числа чи віднімати їх з вказівників. Крім того, вказівники можна віднімати один з одного, щоб визначити кількість комірок, що знаходяться між відповідними адресами. От і весь набір операцій. Причому кожна з них неявно залежить від типу змінної, на яку посилається вказівник. Можна сказати, що операція

```
p=p+1;
```

має невизначений результат, якщо невідомий тип вказівника. Уточнимо інформацію, указавши тип змінної.

```
int* p= new int;
p=p+1;
```

Тепер зрозуміло, що вказівник варто перенести на дві комірки уперед (у шістнадцятирічній операційній системі).

А тепер уявіть собі такий сценарій.

```
int* p = new int;  // Виділяємо пам'ять для цілочисельної змінної
*p=1;              // Привласнюємо їй одиницю
delete p;          // Видаляємо цілочисельну змінну
...                // Виконуємо операції
p=p+1;             // На скільки комірок перенести вказівник?
```

Проблема полягає в тім, що оператор `delete` звільняє динамічну пам'ять, виділену для цілочисельної змінної, але не знищує вказівник `p`. Зрозуміло, якщо операція над вказівником виконується відразу ж після звільнення динамічної змінної, нічого страшного не відбудеться. Спробуйте самі!

Небезпечні маніпуляції з вказівниками

```
#include <iostream.h>

int main()
{
  int *p = new int;
  *p=1;
  cout << p << " " << (unsigned long)p << endl;
  delete p;
  p=p+1;
  cout << p << " " << (unsigned long)p << endl;
  return 0;
}
```

Як бачите, нічого страшного. Однак потенційна небезпека все-таки існує — адже комірка, на яку продовжує посилатися вказівник `p`, вважається вільною. Ніщо не заважає операційній системі розмістити там дані програми і навіть службову інформацію. У першому випадку ви ризикуєте лише своїми даними, а в другому — можете знищити операційну систему.

Ще одна потенційна небезпека таїться в існуванні *загублених динамічних змінних*, недоступних програмі. Ці змінні утворять так називане *сміття*. Комірки пам'яті, що вони займають, не використовуються програмою, але і не повернуті операційній системі. Це приводить до *витоку пам'яті*, що може досягати значних розмірів.

Витік пам'яті

```
#include <iostream.h>

int main()
{
 int *p = new int;  // Установлюємо вказівник p на динамічну змінну
 *p=1;              // Привласнюємо їй одиницю
 cout << p << endl; // Виводимо її адреса
 p= new int; // Установлюємо вказівник p на динамічну змінну —
             // стара загублена!
 cout << p << " " << (unsigned long)p << endl;
 return 0;
}
```

Більшість цих проблем є наслідком тієї свободи, що вказівники надають програмістам. Люди, що віддають перевагу безпеку, погодилися на обмеження цієї вободи і створили мову, у якому немає вказівників, — мову Java. В основу механізму, що забезпечує безпеку програмування на цій мові, лягло поняття, що є й у мові C++, — *посилання*.

2.6.Посилання

При використанні вказівників для непрямої адресації статичних змінних ми зштовхнулися з неприємною обставиною: додатковою витратою пам'яті. Для того щоб розв’язати цю проблему (і не тільки цю), у мові C++ передбачений особливий тип змінних, що називаються *посиланнями, які є альтернативним ім'ям об'єкта*. Однак при описі механізму, що лежить в основі посилань, часто наводиться аналогія з константним вказівником. Таким чином, виникає ілюзія, що зовні посилання — це об'єкт, а власне кажучи — це константний вказівник. Спробуємо розібратися.

Застосування посилань

```
#include <iostream.h>

int main()
{
 int var=1;
 int* const pVar=&var;     // Оголошення константного вказівника
 int& sVar=var;            // Оголошення посилання

 cout << "Адреса змінної    = " << &var  << endl;
 cout << "Адреса посилання  = " << &sVar << endl;
 cout << "Адреса вказівника = " << &pVar << endl;
```

```
  return 0;
}
```

У підсумку одержимо наступні результати.

```
Адреса змінної     = 0x0065FDF4
Адреса посилання   = 0x0065FDF4
Адреса вказівника = 0x0065FDF0
```

Оголошення посилання характеризується двома особливостями. По-перше, після імені типу посилання вказується символ &. Зовні це нагадує оголошення вказівника, що також має тип, лише замість символу & використовується символ *. По-друге, посилання є константою, тобто зв'язується з об'єктом під час оголошення і ніколи не змінюється згодом. Церіднить її з константним вказівником, що також намертво прив'язується до деякої змінній. Однак вказівник являє собою окрему сутність — він має власну адресу. Крім того, константний вказівник ініціалізується адресою змінної, а посилання — самої змінної. І нарешті, для того щоб одержати значення змінної, на яку посилається вказівник, його необхідно розіменувати, а посилання розіменовується неявно. У той же час посилання відрізняється від об'єкта: вона не займає додаткової пам'яті і може знищуватися раніш (але не пізніше) знищення самого об'єкта.

Локальні посилання

```
#include <iostream.h>

int main()
{
 int var= 1;
 {
    int& sVar = var;   // Локальне посилання ініціалізується змінною var
    cout << sVar << endl;
 }

 {
    int var = 2;          // Локальна змінна var
    int&sVar = b;         // Посилання на змінну var
                          // (знищується одночасно з var)
 }

 cout << sVar << endl;   // Помилка (посилання sVar не існує)

 return 0;
 }
```

Оскільки посилання — це друге ім'я об'єкта, будь-які операції над посиланнями виконуються точно так само, як і над самим об'єктом. У той же час посилання дозволяють змінювати аргументи функції. Власне, у цьому і полягає їхнє основне призначення.

У мові Java посилання замінили собою вказівники. Це значно підвищило надійність програм, оскільки пам'ять виявилася захищеною.

2.7. Резюме

- *Змінна* — це абстрактна назва комірки чи декількох комірок пам'яті. Кожна змінна має шість атрибутів: ім'я, адресу, значення, тип, область видимості і час життя.
- *Адреса* змінної — це адреса комірки пам'яті, зв'язаної з даною змінною.
- *Область видимості* — частина програми, де змінна є доступною.
- *Локальні змінні* — змінні, область видимості якої обмежена тілом функції або блоку.
- *Глобальні* змінні — змінні, доступні в будь-якій точці програми.
- *Значення* змінної — це двійкове число, записане в комірках пам'яті, зв'язаних з даною змінною.
- *Тип* змінної визначає діапазон значень, що вона може набувати, і набір операцій, що до неї можна застосовувати.
- *Зв'язування* — це процес установлення зв'язку між атрибутом і змінною. Момент, коли цей зв'язок установлюється, називається *часом зв'язування*. Зв'язування може відбуватися під час компіляції або виконання програми, у залежності від цього розрізняють *статичне* і *динамічне зв'язування*.
- *Явне оголошення* — це оператор програми, у якому перераховані імена змінних і зазначений їхній тип.
- *Неявне оголошення* — це механізм зв'язування змінних з типами на основі прийнятих угод. У цьому випадку оператор оголошення не потрібний — перша поява імені змінної в програмі вважається її неявним оголошенням.
- Мова C++ використовує лише *явні статичні оголошення змінних*.

- *Область видимості* і *час життя змінної* тісно зв'язані між собою. Будь-яку змінну необхідно зв'язати зі своєю комірки. Для цього адресу комірки необхідно витягти зі списку вільної пам'яті (купи). Цей процес називається *розміщенням змінної в пам'яті*. Зворотний процес називається *видаленням змінної з пам'яті*, чи *знищенням об'єкта*. Він полягає в тім, що адреса комірки, раніше виділеної для збереження змінної, повертається в список вільної пам'яті.
- *Час життя змінної* — це відрізок часу, протягом якого змінна зв'язана з визначеною коміркою пам'яті. Отже, початком існування змінної варто вважати момент її зв'язування з визначеною коміркою пам'яті, а кінцем — момент її відкріплення від цього осередку.
- Логічні змінні належать до булевого типу, що позначається ключовим словом `bool`. Вони набувають лише два значення — істина і хибність.
- Для того щоб подати ціле число без знаку, використовується модифікатор `unsigned`. Для симетрії в мові C++ передбачений і модифікатор `signed`, але оскільки всі цілочисельні змінні за замовчуванням мають знак, це ключове слово практично не вживається. Цілі числа зі знаком змінюються від –32768 до 32767. Цілі числа без знака змінюються від 0 до 65535.
- Крім того, цілі числа можна модифікувати за допомогою ключових слів `short` і `long`. За замовчуванням модифікатор `short` еквівалентний типу `int`. Набагато кориснішим виявився модифікатор `long`, що дозволяє розширити діапазон зміни цілих чисел більш ніж у 30 000 разів: від –2147483648 до 2147483647. Скориставшись модифікатором `unsigned`, можна подати цілі числа від 0 до 4294967295.
- За замовчуванням усі цілі чи, як їх ще називають, *інтегральні* типи мають знак, і тому модифікатор `signed` практично не застосовується.
- У мові C++ передбачені три типи чисел із точкою, що плаває: `float`, `double` і `long double`. Варто мати на увазі, що насправді вони є раціональними і лише апроксимують дійсні числа. Отже, *комп'ютер оперує винятково раціональними числами*.

Тип	Довжина, біт	Знаковий біт	Показник ступеня, біт	Мантиса, біт
`float`	32	1	8	23
`double`	64	1	11	52
`long double`	80	1	15	64

Тип	Min	Max	Eps
`float`	1.1754944351e-38	3.4028233466e+38	1.192092896e-07
`double`	2.2250738585072014e-308	1.79769331348623258e-308	2.220446049250313e-16
`long double`	3.3621031431120935063e-4932	1.189731495357231765e+4932	1.084202172485504434l2e-19

- У залежності від виду зв'язування змінні в мові C++ підрозділяються на чотири категорії: статичні, автоматичні, динамічні і зовнішні.
- *Статичними* називаються змінні, які зв'язуються з коміркою пам'яті на етапі компіляції і залишаються зв'язаними з нею аж до припинення виконання програми.
- Статичні змінні розділяються на локальні і глобальні. Локальна статична змінна, як і глобальна, не змінює адреси при виконанні програми. Однак локальна статична змінна, на відміну від глобальної, доступна лише усередині свого блоку чи функції, зберігаючи свої значення між викликами функції.
- *Автоматичними* називаються змінні, які з коміркою зв'язуються динамічно, тобто під час виконання операторів оголошення, а з типом — статично. Такі змінні іменуються автоматичними, тому що вони створюються і знищуються без утручання програміста.
- У мові C++ усі локальні змінні за замовчуванням є автоматичними. За винятком адреси, всі атрибути автоматичних змінних зв'язуються статично. Виключення складають об'єднання, що у різні моменти часу можуть містити змінні різних типів і, отже, зв'язуються з типом динамічно.
- *Динамічні змінні* не мають імен. Вони розташовуються в динамічній пам'яті, чи *купі*, і видаляються з неї під час виконання програми. Для звертання до цих змінних потрібні особливі засоби: вказівники і посилання. Динамічні змінні створюються або оператором `new`, або стандартною функцією (`malloc()`, `calloc()` чи `realloc()`). Знищення динамічних змінних здійснюється оператором `delete` (разом з оператором `new`) чи функцією `free()` (у сполученні з функціями `malloc()`, `calloc()` чи `realloc()`).
- Мінімальна область пам'яті, виділювана змінним, називається *параграфом*.
- Як правило, оголошення змінної одночасне є її визначенням. Однак указавши перед ім'ям змінної специфікатор `extern`, можна оголосити її, не визначаючи. Це дозволяє одержати доступ до змінної, описаної в іншій частині програми.

- Специфікатор `register` дозволяє зберігати значення змінної в регістрі центрального процесора, а не в пам'яті. Це набагато підвищує швидкість виконання операцій над цією змінною.
- Простір імен — це іменована область видимості.
- У мові C++ існує два кваліфікатори, що керують доступом до змінного і модифікацією: `const` і `volatile`. Їх називають *cv-квалификаторами*.
- Якщо перед оголошенням змінної стоїть кваліфікатор `const`, це значить, що вона не може змінюватися. Отже, єдиний спосіб привласнити їй яке-небудь значення — ініціалізувати її під час оголошення.
- Деякі сутності можуть змінюватися неявно, наприклад, операційною системою, апаратним забезпеченням або рівнобіжним потоком керування. Такі змінні відзначають за допомогою кваліфікатора `volatile`.
- Класична схема приведення типу у мові C має два різновиди.
  (*новий_тип*)<*операнд_старого_типу*>
  новий_тип(*операнд_старого_типу*)
- У мові C++ є ще чотири способи приведення типів — оператори `static_cast`, `const_cast`, `dynamic_cast` і `reinterpret_cast`. Їхні схеми виглядають так.

```
static_cast<тип>(вираз)
const_cast<тип>(вираз)
dynamic_cast<тип>(вираз)
reinterpret_cast<тип>(вираз)
```

- *Вказівник* — це змінна, у якій зберігається адреса комірки пам'яті чи нульова адреса. Як правило, нульова адреса задається константою `NULL`. Він застосовується в ситуаціях, коли вказівник не містить жодної реальної адреси, як звичайно говорять, “ні на що не посилається”.
- Як відомо, звичайні іменовані змінні розташовуються або в стеку, якщо вони є локальними, або в окремому сегменті пам'яті, призначеному для збереження глобальних змінних. Обидва види цих змінних називаються *статичними*, оскільки пам'ять для них виділяється на етапі компіляції. Інакше кажучи, зв'язування їх імен з адресами є статичним.
- Змінні, розташовувані в купі, називаються *динамічними*. Як правило, вони є *безіменними*, а доступ до них забезпечується чи вказівниками посиланнями.
- При використанні вказівників для непрямої адресації статичних змінних ми зштовхнулися з неприємною обставиною: додатковою витратою пам'яті. Для того щоб розв’язати цю проблему (і не тільки цю), у мові C++ передбачений особливий тип змінних, що називаються *посиланнями*, *які є альтернативним ім'ям об'єкта*.

2.8. Контрольні питання

2.1. Що називається змінною?
2.2. Що називається адресою змінної?
2.3. Що називається областю видимості?
2.4. Що називається локальною змінною?
2.5. Що називається глобальною змінною?
2.6. Що називається значенням змінної?
2.7. Що називається типом змінної?
2.8. Що називається зв’язуванням?
2.9. Яки види зв’язування існують у мові C++?
2.10. Що називається явним оголошенням?
2.11. Який процес називається розміщенням змінної в пам'яті?
2.12. Який процес називається видаленням змінної з пам'яті, чи знищенням об'єкта.
2.13. Що називається часом життя змінної?
2.14. Який тип використовується у мові C++ для подання логічних величин?
2.15. Які типи використовується у мові C++ для подання цілих величин? Опишіть їхні особливості.
2.16. Які типи називаються інтегральними?
2.17. Який тип використовується у мові C++ для подання раціональних чисел? Опишіть їхні особливості.
2.18. Які категорії змінних існують в мові C++ у залежності від виду зв'язування?
2.19. Які змінні називаються статичними?
2.20. На які категорії розділяються статичні змінні
2.21. Які змінні називаються автоматичними?
2.22. Які змінні називаються динамічними? Які функції та оператори використовуються для їх утворення та знищення?
2.23. Що називається параграфом?
2.24. Опишіть ключове слово `extern`.
2.25. Опишіть ключове слово `register`.
2.26. Що таке простір імен?
2.27. Які кваліфікатори керують доступом до змінної та їхньою модифікацією?
2.28. Що означає класифікатор `const`?
2.29. Що означає кваліфікатор `volatile`.
2.30. Опишіть класичну схему приведення типу у мові C.
2.31. Опишіть чотири оператори приведення типу у мові C++
2.32. Що таке вказівник?
2.33. Що таке посилання?