Лекція 4

Оператори

У цій главі...

4.1. Арифметичні оператори

Метою будь-якої мови є переклад думки в деяку матеріальну форму, доступну для розуміння іншими людьми. Мова програмування також являє собою засіб спілкування, тільки співрозмовником є комп'ютер. Правда, іноді утомлені програмісти в спілкуванні з комп'ютером переходять на ненормативну лексику, однак, як правило, успіху це не приносить. Єдиний вид міркувань, доступний комп'ютеру, — обчислення. Отже, мова програмування повинна надавати програмісту можливість подати обчислення в доступній комп'ютеру формі. Для цього в кожній мові програмування існують правила утворення *виразів*.

Вираз — це основний засіб обчислень, що складається з операторів і операндів. Порядок обчислення виразів визначається правилами асоціативності і пріоритетами операторів.

Головна мета всіх імперативних мов програмування, до яких належить і C++, — досягти результату, послідовно змінюючи значення змінних. Єдиний спосіб змінити значення змінної — привласнити їй значення іншої змінної чи виразу. Просте копіювання значень з однієї змінної в іншу, хоч і зустрічається досить часто, занадто тривіально, щоб виразити складні алгоритми. Отже, необхідно мати можливість обчислити значення виразу, а потім привласнити його змінній. Таким чином, головну роль в імперативних мовах програмування грає оператор присвоювання, а інші потрібні для обчислення виразів. Оскільки оператор присвоювання зв'язаний з багатьма арифметичними і порозрядними операторами, ми розглянемо його наприкінці глави, після опису всіх операторів, передбачених у мові C++.

Природно, всі обчислення в комп'ютері носять математичний характер. Навіть якщо комп'ютер застосовується в яких-небудь інших областях, неявно вся робота зводиться до перетворення двійкових чисел — змінюється лише их інтерпретація. Таким чином, не буде великим перебільшенням сказати, що всі обчислення певною мірою є арифметичними.

У мові C++ арифметичні вирази складаються з операторів, операндів, круглих дужок і викликів функцій. Оператор може бути *унарним*, тобто з одним операндом, *бінарним*, що має два операнди, і *тернарним*, у якого є три операнди. Відразу обмовимося, що всі арифметичні оператори в C++ є або унарними (плюс `+`, мінус `–`, інкремент `++`, декремент `––`), або бінарними (додавання `+`, віднімання `–`, множення `*`, ділення `/`, ділення по модулю `%`). Єдиний тернарний оператор у мові C++ (`:?`) належить до групи логічних операторів і буде розглянутий пізніше.

Арифметичні оператори бувають префіксними, інфіксними і постфіксними. Префіксні оператори завжди передують своїм операндам, інфіксні записуються між операндами, а постфіксні — після них. Наприклад, +1 — префіксна форма запису, 2+3 — інфіксна, а 23+ — постфіксна (так називаний *польський запис*).

У C++ унарні оператори є як префіксними, так і постфіксними. Унарні оператори `+` і `–` завжди передують операнду, а оператори інкремента `++` і декремента `––` можуть бути як префіксними, так і постфіксними. Усі бінарні оператори є інфіксними, тобто вони з'являються між операндами. Це наближає мову C++ до природного способу запису алгебраїчних виразів.

Призначення арифметичного виразу — обчислення. Для цього комп'ютер повинний витягти з пам'яті значення операндів, помістити їх в арифметико-логічний пристрій і застосувати до них арифметичні оператори. Порядок обчислення арифметичного виразу визначається *пріоритетами* операторів, *правилами асоціативності* і можливими *побічними ефектами*, як правило, зовсім небажаними.

Розглянемо кожний з арифметичних операторів.

У мові C++ існують унарні операції плюс і мінус. Унарний плюс називається *тотожним оператором*, оскільки він не впливає на операнд. Інакше кажучи, якщо перед операндом стоїть плюс, це не завжди означає, що отримане в результаті число буде позитивним. У цьому легко переконатися, запустивши на виконання наступну програму.

Унарний плюс — тотожний оператор

```
#include <iostream.h>

int main()
{
    int i = -1;
```

```
    cout << +i;

    return 0;
}
```

У результаті на екран буде виведене число `-1`. Таким чином, можна припустити, що унарний плюс включений у мову "для симетрії". На відміну від його, унарний мінус завжди змінює знак операнда на протилежний.

Унарний мінус

```
#include <iostream.h>

int main()
{
    int i = -1;
    cout << -i;

    return 0;
}
```

Результат роботи цієї програми — число `1`.

Бінарні оператори плюс і мінус є абсолютно очевидними. Вони обчислюють суму і різницю двох операндів. Єдина особливість цих операторів зв'язана з переповненням змінних інтегрального типу (див. главу 3).

Оператори інкремента `++` і декремента `--` більш складні. Кожний з них є унарнім і має як префіксну, так і постфіксну форму.

```
  i++; // Збільшує значення змінної i на одиницю
  ++i; // Збільшує значення змінної i на одиницю
  i--; // Зменшує  значення змінної i на одиницю
  --i; // Зменшує  значення змінної i на одиницю
```

Коли вони застосовуються самостійно, жодних проблем не виникає. Після їхнього виконання значення змінної або збільшується на одиницю (`++`), або зменшується на одиницю (`--`). У цьому випадку префіксні і постфіксні форми запису еквівалентні.

```
  // Еквівалентні оператори, що збільшують значення змінної i на одиницю
   i++;
   ++i;
   i=i+1;
  // Еквівалентні оператори, що зменшують значення змінної i на одиницю
   i--;
   --i;
   i=i-1;
```

Великі проблеми виникають, коли оператори інкремента і декремента використовуються усередині виразів. У цьому випадку між постфіксною і префіксною формами запису виникають відмінності. Навіть найпростіше присвоювання ускладнюється, оскільки його результат залежить від форми оператора інкремента чи декремента.

Наприклад, постфіксна форма оператора інкремента у виразі

```
   a=b++;
```

еквівалентна наступним операторам.

```
   a=b;
  b++;
```

У той же час префіксна форма

```
   a=++b;
```

еквівалентна таким операторам.

```
   ++b;
   a=b;
```

Постфіксна і префіксна форми оператора декремента мають такі ж властивості. Інакше кажучи, постфіксна форма змінює значення операнда `b` після його присвоювання змінній `a`, а префіксна форма — до присвоювання. В обох випадках операнд праворуч збільшиться чи зменшиться на одиницю, а лівий операнд оператора присвоювання обчислюється по-різному. Інакше кажучи, постфіксну форму оператора інкремента можна назвати "підставити і збільшити", а префіксну — "збільшити і підставити".

Як правило, оператор інкремента застосовують до целочиельних операндів, хоча на цей рахунок не існує жодних обмежень.

Оператор інкремента

```
#include <iostream.h>

int main()
{
```

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

    double d = 1;
    d++;
    cout << d << endl;

    return 0;
}
```

У результаті змінні `i` і `d` будуть дорівнювати 2. Зверніть увагу на те, що в цій програмі оператор інкремента винесений в окремий рядок. Це зроблено, щоб уникнути непорозумінь, що підстерігають програміста, коли він використовує оператор інкремента усередині виразу. Навіть найпростіший вивід на екран може призвести до непорозумінь. Наприклад, програма, у якій оператор виводу комбінується з постфіксною інкрементацією, виводить на екран не нове, а старе значення змінної.

Постфіксна форма оператора інкремента

```
#include <iostream.h>

int main()
{
    int i = 1;
    cout << i++ << endl;

    return 0;
}
```

Тут на екран виводиться число 1. Щоб виправити ситуацію і досягти бажаного результату, слід застосувати префіксний оператор.

Префіксна форма оператора інкремента

```
#include <iostream.h>

int main()
{
    int i = 1;
    cout << ++i << endl;

    return 0;
}
```

Тепер, як і слід було очікувати, на екран виводиться число 2.

Ще одна тонкість полягає в тім, що постфіксна форма операторів інкремента і декремента як результат повертає *константу*, збільшену чи зменшену на одиницю. Отже, постфіксну форму цих операторів неможливо застосувати дещо разів подряд.

```
i++++; // Помилка!
```

У той же час префіксна форма обох операторів повертає *посилання* на змінну, що дозволяє створювати з них ланцюжки.

Ланцюжки префіксних форм

```
#include <iostream.h>

int main()
{
    int i = 1;
    ++--++----++i;
    cout << i << endl;

    return 0;
}
```

Спробуйте самі догадатися, чому дорівнює результат!

Ще більше плутанини виникає, коли в одному виразі змішуються два і більш оператори інкремента чи декремента. Розглянемо конкретний і досить несподіваний приклад.

Змішування різних форм інкремента: перша версія

```
#include <iostream.h>

int main()
{
```

```
    int i = 1,j;
    j=(++i)*(++i);
    cout << j << endl;

    return 0;
}
```

Природно було б очікувати, що оператор `j=(++i)*(++i);` виконується в такий спосіб.

```
tmp1=++i;       // i=2
tmp2=++i;       // i=3
j= tmp1*tmp2;   // j=6
```

Нітрохи не бувало! На екран виводиться число 9! Інакше кажучи, розшифровка коду виглядає в такий спосіб.

```
tmp=++i;       // i=2, tmp=2
tmp=++i;       // i=3, tmp=3
j= tmp*tmp;    // i=3, j=9
```

Ну що ж, усе логічно. Спочатку змінна `i` збільшується на одиницю, а потом її остаточне значення перемножується. Дещо неприродно, але звикнути можна, оскільки префіксна форма оператора інкрементації припускає, що операнди обчислюються в першу чергу. А що відбудеться, якщо це вираз ми помістимо в оператор виводу.

Змішування різних форм інкремента: друга версія

```
#include <iostream.h>

int main()
{
    int i = 1,;
    cout << (++i)*(++i) << endl;

    return 0;
}
```

На щастя, компілятор Visual C++ 6.0 поводиться послідовно — ми знову одержуємо число 9. А тепер скомбінуємо різні форми.

Змішування різних форм інкремента: третя версія

```
#include <iostream.h>

int main()
{
    int i = 1;
    cout << (++i)*(++i)<< " " << i;
    cout << " " << i << endl;
    i=1;
    cout << (i++)*(i++)<< " " << i;
    cout << " " << i << endl;
    i=1;
    cout << (++i)*(i++)<< " " << i;
    cout << " " <<i << endl;
    i=1;
    cout << (i++)*(++i)<< " " << i;
    cout << " " << i << endl;

    return 0;
}
```

Результат такий.

```
9 1 3
1 1 3
4 1 3
4 1 3
```

З першою комбінацією ми розібралися. Друга знову виглядає, на перший погляд, дивно. Але варто згадати, що обидві форми оператора інкремента є постфіксними, як усе стає на свої місця.

Оператор

```
j=(i++)*(i++);
```

еквівалентний наступним операторам.

```
tmp1=i;         // i = 1, tmp1 = 1
tmp2=i;         // i = 1, tmp2 = 1
j=tmp1*tmp2;    // j=1
i++;
i++;
```

Поки потік керування не перейшов до наступного рядку програми, значення змінної `i` залишається рівним одиниці, хоча проміжні змінні обчислюються в повній відповідності з правилами.

Звідкіля ж береться число 4, коли дві форми оператора інкремента змішуються в одному виразі. Розглянемо вираз `(i++)*(i++)`.

```
++i;              // i=2;
tmp1=i;           // i = 2, tmp1 = 2
tmp2=i;           // i = 2, tmp2 = 2
j=tmp1*tmp2;      // j=4
i++;              // i = 3
```

Вираз `(i++)*(++i)` "розкладається" аналогично.

Тут напрошується деякий логічний висновок, що підсумував би правила обчислення різних комбінацій постфіксних і префіксних форм інкрементації і декрементації, але доведеться поставити крапку. Справа в тім, що ці правила стандартом мови C++ не регламентуються — кожен компілятор може приймати власні правила.

Скомпілюємо ту ж програму за допомогою компілятора Turbo C++ 3.0. Результати обчислень приведені нижче.

```
6 1 3
2 1 3
4 1 3
3 1 3
```

Як бачимо, вони більше відповідають інтуїтивним очікуванням. Зокрема, оператор

```
j=(++i)*(++i);
```

еквівалентний наступним операторам.

```
i++;
tmp1=i;        // i = 2, tmp1 = 2
i++;
tmp2=i;        // i = 3, tmp2 = 3
j=tmp1*tmp2;   // j = 6, i = 3
```

Як і колись, потік керування не перейшов до наступного рядку програми, значення змінної `i` залишається рівним одиниці, а проміжні змінні тепер обчислюються інакше.

Розглянемо вираз `(i++)*(i++)`.

```
tmp1=i;       // i = 1, tmp1 = 1
i++;          // i = 2
tmp2=i;       // i = 2, tmp2 = 2
++i;          // i = 3
j=tmp1*tmp2; // j = 2
```

Вираз `(i++)*(++i)` "розкладається" так.

```
tmp1=i;        // i = 1, tmp1 = 1
i++;           // i = 2
tmp2=i;        // i = 2, tmp2 = 2
++i;           // i = 3
j=tmp1*tmp2;   // j = 4
```

У той же час вираз `(i++)*(++i)` тепер не симетричний виразу `(++i)*(i++)`.

```
tmp1=i;       // i = 1, tmp1 = 1
i++;          // i = 2
++i;          // i = 3
tmp2=i;       // i = 3, tmp2 = 3
j=tmp1*tmp2; // j = 3
```

Таку мінливість правил досить важко пояснити, і пристосовуватися до них недоцільно. Варто просто уникати таких заплутаних ситуацій і не змішувати різні форми операторів в одному виразі.

4.1.1. Пріоритети арифметичних операторів

Значення виразу залежить від порядку обчислення операторів, тобто від їхніх пріоритетів. Операції множення, ділення, а також ділення по модулю, інкремента і декремента мають більш високий пріоритет, ніж операції додавання і множення. Круглі дужки дозволяють змінити цей порядок. Наприклад, вираз `A*B+C-D/E` за замовчуванням еквівалентний виразу `(A*B)+C-(D/E)`. Інакше кажучи, спочатку обчислюється добуток змінних `A` і `B`, потім —ділення змінної `D` на змінну `E`, результат добутку додається до змінної `C`, і на закінчення з отриманого числа віднімається частка `D/E`. Це означає, що пріоритет операції множення і ділення вище, ніж пріоритет операції додавання, причому операції, що мають однаковий пріоритет, виконуються справа наліво.

Поняття пріоритету операторів прийшло в програмування з математики. Саме в математиці використовується ієрархія операторів, що дозволяє однозначно визначати значення алгебраїчних виразів, не прибігаючи до масового використання дужок. Зокрема, операція множення в математичних виразах завжди має більш високий пріоритет, ніж операція додавання. Таким чином, *пріоритети операторів* визначають порядок, у якому виконуються оператори в арифметичних виразах.

Два символи операції не повинні “сусідити”. Якщо так відбулося, варто відокремити їх круглими дужками. Наприклад, вираз `A+B*(-C+D)` цілком коректний, а вираз `A+B*-C+V` містить синтаксичну помилку.

Як ми могли переконатися, порядок виконання операторів визначається не тільки пріоритетом операторів, але і правилами асоціативності.

4.1.2. Правила асоціативності

Операції множення, ділення і ділення по модулю мають однаковий пріоритет. Розглянемо вираз `A*B/C*D/E`. Він еквівалентний виразу `((A*B)/C)*D)/E`. Це пояснюється тим, що порядок виконання операторів, що мають однаковий пріоритет, визначається правилами *асоціативності*. Асоціативність оператора буває право- і лівосторонньою, що відповідає виконанню операторів справа наліво чи зліва направо. У мові C++ зустрічаються обидва види асоціативності. Наприклад, у виразі `A*B/C` оператори виконуються зліва направо, а у виразі `++--i` — справа наліво. Оператори `*`, `/`, `%`, бінарний `+`, бінарний `-` є лівоасоціативними, а оператори `++`, `--`, унарний `-`, унарний `+` — правоасоціативними.

У компіляторах мови C++ використовуються природні математичні правила асоціативності арифметичних операторів. Однак варто мати на увазі, що математика — наука абстрактна, а програмування — конкретна. У контексті наших міркувань це означає, що такі абстракції, як нескінченно малі числа, нескінченно великі числа і навіть звичайні дійсні числа з нескінченною кількістю цифр, у програмуванні відсутні (принаймні, у комп'ютері ще нікому не вдавалося подати число π). Комп'ютер дозволяє зберігати лише скінченні числа, що неминуче обмежує їхній діапазон. Отже, якщо в математиці від перестановки місць доданків сума не змінюється, то в програмуванні усе може виявитися інакше.

Легко можна уявити собі ситуацію, у якій додавання чисел типу `double` може виглядати некомутативним. Розглянемо, наприклад, вираз `A+B+C` , де змінні `A` і `B`, що мають тип `double`, дорівнюють 1.7e+308, а змінна `C` типу `double` дорівнює -1.7e+308. Сума чисел `A` і `B` виходить за межі припустимого діапазону, тому якщо оператори додавання виконуються зліва направо, ми відразу ж вийдемо за верхню межу. З іншого боку, вираз `A+C+B` не повинен привести до переповнення, оскільки, виконуючи додавання зліва направо, ми залишаємося в межах припустимих значень. Теоретично це так. Однак сучасні компілятори оптимізують обчислення виразів, і тому обоє виразів дійсно мають те саме значення: 1.7e+308.

Комутативне додавання

```
#include <iostream.h>
int main()
{
    double a = 1.7e+308, b=1.7e+308, c=-1.7e+308, d, e ;

    d=a+b+c;
    e=a+c+b;
    cout << d << " " << e;

    return 0;
}
```

Результат: `1.7e+308  1.7e+308`

Некоммутативність обчислень можлива, але зв'язана вона, у першу чергу, з неоднозначністю апроксимації дійсних чисел.

Некомутативне додавання

```
#include <stdio.h>

int main()
{
    float a,b, c, d;
    a=0.999999;
    b=1;
    c=a+b;
    d=b+a;
    printf("%14.10f \n%14.10f",c,d);
}
```

Результати виконання програми після компіляції в середовищі Visual C++ 6.0 такі.

```
 1.9999990463
 1.99999989867
```

Слід зазначити, що ця ж програма, скомпільована за допомогою компіляторів Turbo C++ 3.0 і CodeWarrior 8.0, виводить на екран “комутативні” результати.

```
 1.9999990463
 1.9999990463
```

Зрозуміло, причина невідповідності полягає в тім, що при роботі з числами типу `float` значущими вважаються тільки перші шість цифр (інші — сміття), але ведь в обчисленнях беруть участь усі цифри! На це необхідно зважати.

Розглянемо як ілюстрацію наступний приклад.

Ефект “випадкових цифр”

```
#include <stdio.h>
int main()
{
    float a,b, c, d;
    a=0.999999;
    b=1;
    c=10e+09*(a+b);
    d=10e+09*(b+a);
    printf("%f \n%f",c,d);

    return 0;
}
```

Тепер результати в середовищі Visual C++ 6.0 відрізняються набагато сильніше.

```
 19999989760.000000
 19999989867.210388
```

Компілятори Turbo C++ 3.0 і CodeWarrior 8.0 як і раніше приводять до комутативних результатів.

```
 19999989760.000000
 19999989760.000000
```

Як бачимо, стандартні математичні обчислення варто виконувати з великою обережністю, ретельно контролюючи їх точність.

4.1.3. Дужки

Коли природний порядок обчислення операторів не підходить, пріоритет і правила асоціативності програміст може змінити за допомогою круглих дужок. Наприклад, розглянутий нами вище вираз `A+B*C` можна змінити в такий спосіб: `(A+B)*C`.

У цьому випадку в першу чергу обчислюється вираз, укладений в дужки, який би пріоритет не мали інші оператори. Це цілком відповідає звичайним математичним правилам обчислення алгебраїчних виразів.

4.1.4. Перетворення типів

При використанні у виразі операндів різних типів компілятор виконує неявні перетворення. Розглянемо два види таких перетворень: при присвоюванні і при обчисленні арифметичного виразу.

Допустимо, в операторі присвоювання типи лівого і правого операндів відрізняються. У цьому випадку значення правого операнда приводиться до типу лівого операнда оператора присвоювання. Якщо тип лівого операнда вимагає для свого представлення менше бітів, ніж тип правого операнда, перетворення називається *звужуючим*. В іншому випадку воно іменується *розширювальним*.

Перетворення типів

```
#include <iostream.h>
int main()
{
    char ch = 127;
    unsigned char uc = 127;

    //Звужуюче перетворення: тип правого операнда ширше лівого

    ch = uc;  // Якщо uc <= 127, результати збігаються
    cout << "Звужуюче перетворення " << endl;
    cout << "char ch          = " << (int)ch << endl;
    cout << "unsigned char ch = " << (int)uc << endl;

    uc = 129;
    ch = uc;  // Якщо uc >127, результати не збігаються
    cout << "char ch          = " << (int)ch << endl;
    cout << "unsigned char ch = " << (int)uc << endl;

    //Розширювальне перетворення: тип лівого операнда ширший

    ch = 127;
    uc = ch;  // Якщо ch <= 127, результати збігаються
    cout << endl <<"Розширювальне перетворення " << endl;
```

```
    cout << "char ch          = " << (int)ch << endl;
    cout << "unsigned char ch = " << (int)uc << endl;

    uc = 129;
    ch = uc;  // Якщо uc >127, результати не збігаються
    cout << "char ch          = " << (int)ch << endl;
    cout << "unsigned char ch = " << (int)uc << endl;

    return 0;
}
```

Проаналізуємо результати.

```
Звужуюче перетворення
char ch          = 127
unsigned char ch = 127
char ch          = -127
unsigned char ch = 129

Розширювальне перетворення
char ch          = 127
unsigned char ch = 127
char ch          = -127
unsigned char ch = 129
```

Поки значення змінної ch не виходить за межі припустимого діапазону, жодних проблем не виникає. Однак при переповненні вступають у дію правила інтерпретації знакового біта і переносу зайвого розряду. Зверніть увагу на те, що проблеми виникають як при звужуючих, так і при розширювальних перетвореннях. Отже, справа не стільки в звуженні діапазону, скільки в зміні правил інтерпретації знакового біта: змінна типу char підкоряється правилу переносу зайвого розряду і з урахуванням знакового біта стає рівною -127, а змінна типу unsigned char приймає значення 129, вважаючи знаковий біт звичайним розрядом.

Подивимося, як виконуються перетворення цілочисельних типів.

Перетворення цілочисельних типів

```
#include <iostream.h>

int main()
{
    int anInt = 32767;
    unsigned int unSign = 32767;
    //Звужуюче перетворення: тип правого операнда ширше лівого

    anInt = unSign;  // Якщо unSign <= 32767,
                     // результати збігаються
    cout << "Звужуюче перетворення " << endl;
    cout << "int anInt       = " << anInt  << endl;
    cout << "unsigned unSign = " << unSign << endl;

    unSign = 37800;
    anInt = unSign;  // Якщо unSign > 32767,
                     // результати не збігаються
    cout << "int anInt       = " << anInt << endl;
    cout << "unsigned UnSign = " << unSign << endl;

    // Розширювальне перетворення: тип лівого операнда ширше правого

    anInt = 32767;
    unSign = anInt;  // Якщо anInt <= 32767, результати збігаються
    cout << "Розширювальне перетворення " << endl;
    cout << "int anInt       = " << anInt << endl;
    cout << "unsigned unSign = " << unSign << endl;
    anInt = 32800;
    unSign = anInt;  // Якщо anInt > 32767, результати не збігаються
    cout << "Розширювальне перетворення " << endl;
    cout << "int anInt       = " << anInt << endl;
    cout << "unsigned unSign = " << unSign << endl;
    return 0;
}
```

Результати такі.

```
Звужуюче перетворення
int anInt       = 32767
unsigned unSign = 32767
int anInt       = -27736
unsigned UnSign = 37800

Розширювальне перетворення
int anInt       = 32767
unsigned unSign = 32767
int anInt       = -37736
unsigned unSign = 37800
```

Як бачимо, тут простежуються ті ж закономірності. Це цілком природно, оскільки типи `char` і `int` є спорідненими. Відзначимо лише, що приведені результати справедливі для 16-розрядних реалізацій. При роботі з 32-розрядними операційними системами верхню межу 32767 варто замінити числом 2147483647. Легко бачити, що це відповідає типу `long` у 16-розрядній операційній системі.

Перетворення типів `signed` у `unsigned` є одним з найбільш простих, оскільки стосується лише одного біта. Однак при перетворенні більш далеких типів справа обстоїть складніше. Розглянемо взаємне перетворення типів `char` і `int`.

Перетворення “далеких” типів

```
#include <stdio.h>

int main()
{
    char ch = 0xBB;             //10111011
    short aShort=0x9999;        //1001100110011001
    int anInt = 0xAAAA;         //1010101010101010
    long  aLong = 0xBBBBAAAA;   //1011101110111011

    printf("Початкові значення \n");
    printf("char   = %x %d \n", ch, sizeof(ch));
    printf("aShort =  %x %d \n", aShort, sizeof(aShort));
    printf("anInt  = %x %d \n", anInt, sizeof(anInt));
    printf("aLong  = %lx %d \n", aLong, sizeof(aLong));

    ch = aShort;
    aShort = anInt;
    anInt = aLong;
    printf("\nПісля присвоювання short->char, int->short, long->int\n");
    printf("char   = %x %d \n", ch, sizeof(ch));
    printf("aShort = %x %d \n", aShort, sizeof(aShort));
    printf("anInt  = %x %d \n", anInt, sizeof(anInt));
    printf("aLong  = %lx %d \n", aLong, sizeof(aLong));

    printf("\n Після присвоювання long -> char\n");
    ch = aLong;
    printf("char   = %x %d \n", ch, sizeof(ch));

    return 0;
}
```

Результати роботи цієї програми заслуговують на увагу (вони отримані в 16-розрядній операційній системі).

```
Початкові значення
char   = ffbb 1
aShort = 9999 2
anInt  = aaaa 2
aLong  = bbbbaaaa 4

Після присвоювання short->char, int->short, long->int
char   = ff99 1
aShort = aaaa 2
anInt  = aaaa 2
aLong  = bbbbaaaa 4

Після присвоювання long->char
char   = ffaa 1
```

Прокоментуємо побачене. Спочатку всі змінні ініціалізуються визначеними шестнадцатерічними значеннями (так зручніше простежити за змінами, що відбуваються при звужуючих перетвореннях). Потім на друк виводяться шестнадцатерічне значення змінних і їх розмір. Незважаючи на те що розмір змінної типу `char` дорівнює 1, на друк виводяться чотири шістнадцятирічні цифри. Перші дві (`ff`) є незначними.

Як бачимо, після присвоювання змінної типу `char` значення типу `short`, старші розряди цілочисельної змінної пропадають і змінна `ch` стає рівною `99` (у шестнадцатеричній системі). Типи `short` і `int` є еквівалентними, тому присвоювання `aShort = anInt` ніяких особливостей не має. Наступне присвоювання `anInt = aLong` є звужуючим, тому, як і колись, старші розряди (`bbbb`) відкидаються.

Таким чином, можна сформулювати правило: *при звужуючому перетворенні інтегральних типів у результат копіюються лише молодші біти, а зайві старші біти ігноруються*. У 16-розрядній операційній системі при перетворенні `int->char` губляться старші вісім розрядів, а при перетворенні `long->char` — 24 біт. При трансформації `long- >int` відбувається втрата 16 старших біт. У 32-розрядних системах розміри типів `long` і `int` збігаються, отже, утрата розрядів відбувається лише під час перетворення `int->char` і дорівнює 24 біт.

Найпростіше перетворення відбувається між типами `int` і `float`. Якщо змінна типу `float` привласнюється змінної типу `int`, дробова частина відкидається, при зворотному присвоюванні до цілочисельної змінної приписується нульова дробова частина.

Перетворення `int` ↔ `float`

```
#include <stdio.h>

int main()
{
    int anInt = 10;
    float aFloat = 1.23;

    anInt = 1.23;
    aFloat = anInt;
    printf("anInt  = %d \n",anInt);
    printf("aFloat = %f \n",aFloat);

    return 0;
 }
```

У підсумку одержуємо наступне.

```
anInt  = 1
aFloat = 1.000000
```

Дробова частина дійсного числа була загублена під час присвоювання. Ці закономірності поширюються і на присвоювання змінних модифікованих цілочисельних типів, а також змінних типу `double` і `long double`. Зауважимо, однак, що тип `double` перетворюється в тип `int` не прямо, а через тип `float`.

Відзначимо ще одну особливість перетворення цілих чисел у дійсні. Наприклад, 32-розрядні компілятори дозволяють зберігати не менш дев'яти значущих десяткових цифр, у той же час така ж кількість розрядів, надана для збереження числа типу `float`, дозволяє зберегти лише 7 значущих цифр. Перетворення типу дозволяє вирішити цю проблему.

Перетворення типів під час присвоювання

```
#include <stdio.h>

int main()
{
    long int aLong = 1111111111;
    float aFloat1 = 1111111111;
    float aFloat2 = anInt;
    printf(" anInt   = %d\n  aFloat1 = %f\n aFloat2 = %f\n",
             anInt, aFloat1, aFloat2);
    return 0;
 }
```

Результати дивні.

```
aLong   = 1111111111
aFloat1 = 1111111168.00000
aFloat2 = 1111111111.00000
```

Як бачимо, ініціалізація змінної типу `float` занадто великим цілочисельним значенням приводить до втрати двох значущих цифр. Присвоювання цього недоліку не має.

Ми розглянули перетворення типів під час присвоювання. Перейдемо тепер до неявного приведення типів у ході обчислення змішаних арифметичних виразів. Ці перетворення підкоряються наступному правилу: *типи*

операндів перетворяться з більш вузьких у більш широкі. Це явище називається *розширенням типів*. Правило розширення можна сформулювати так: *тип результату збігається із найширшим типом операнда*.

Проілюструємо це правило наступною програмою.

Розширення типів

```
#include <iostream.h>

int main()
{
    char ch = 0;
    unsigned int unSigned = 1U;
    int anInt = 2;
    short aShort = 3;
    long aLong = 4L;
    unsigned unLong = 5U;
    float aFloat = 6.0;
    double aDouble = 7.0;
    long double aLongDouble = 8.0;

    int size = sizeof(ch + unSigned + anInt + aShort + aLong +
                      unLong + aFloat + aDouble + aLongDouble);
    cout << "ch + unSigned + anInt + aShort + aLong" << endl <<
           "unLong + aFloat + aDouble + aLongDouble";
    cout << endl << "sizeof = ";
    cout << size << endl;
    return 0;
 }
```

Ця програма виглядає дещо громіздко, але достатньо наочно ілюструє основну думку — *розмір виразу залежить від максимально широкого типу операнда*. Приведені результати отримані в 32-розрядній системі.

```
ch + unSigned + anInt + aShort + aLong
unLong + aFloat + aDouble + aLongDouble
sizeof = 8
```

Оскільки арифметичні вирази практично ніколи не “висять у повітрі” і частіше є правим операндом оператора присвоювання, правило визначення типу результату в сполученні з правилами приведення типів набуває особливого значення. Можна обчислити найскладніший вираз і втратити старші розряди результату просто через те, що його тип був визначений невірно.

4.2. Оператори порівняння

У мові C++ передбачено шість операторів порівняння: < (менше), <= (менше чи дорівнює), > (більше), >= (більше чи дорівнює), == (дорівнює), != (не дорівнює).

Оператори порівняння носять логічний характер, тобто результатом є “істина” і “хибність”. У мові C++ ці поняття мають двоїстий характер. По-перше, істинним вважається будь-яке значення, що не дорівнює нулю, а помилкове значення завжди дорівнює 0. По-друге, у C++ існує особливий тип даних `bool` і булеві константи `true` і `false`. У ході виконання операторів порівняння значення 0 автоматично перетвориться в константу `false`, а будь-яке ненульове значення — у константу `true`. І навпаки, константа `true` перетвориться в значення 1, а константа `false` — у число 0. Отже, результатом обчислення вираз, що містять оператори порівняння, є константи `true` і `false`. Відзначимо, що, внаслідок автоматичного перетворення булевих констант у числа 0 і 1 і навпаки, мова C++ допускає змішані вирази. Розглянемо приклад.

Оператори порівняння

```
#include <iostream.h>

int main()
{
    bool Boolean1, Boolean2;
    int i1 = 10, i2 = 20;
    Boolean1 = i1 > i2;
    Boolean2 = i2 < i1;
    cout << "Boolean1 = " << Boolean1<< endl;
    cout << "i1 > i2 = " << (i1 > i2) << endl;
    cout << "Boolean2 = " << Boolean2<< endl;
    cout << "i2 > i1 = " << (i2 > i1) << endl;
    cout << "Boolean1 + Boolean2 = " << Boolean1+Boolean2<< endl;
    cout << "i2 > i1 + Boolean2 = " << (i2 > i1) + Boolean2 << endl;
```

```
    return 0;
}
```

У результаті одержуємо наступне.

```
Boolean1 = 0
i1 > i2 = 0
Boolean2 = 1
i2 > i1 = 1
Boolean1 + Boolean2 = 1
i2 > i1 + Boolean2 = 2
```

Цікавою властивістю операторів порівняння є можливість створення ланцюжків.

```
a1 > a2 > a3
```

Оператори порівняння мають властивість лівої асоціативності, отже, порядок их обчислення такий: спочатку обчислюється крайній ліворуч оператор `a1 > a2`, а потім його результат порівнюється із змінною `a3`. Відзначимо, що змінні `a2` і `a3` не порівнюються. Зрозуміло, цей порядок можна змінити, застосувавши дужки: `a1 > (a2 > a3)`.

Крім того, оператори порівняння мають більш низький пріоритет, ніж арифметичні оператори. Таким чином, вираз `a1 > a2 + a3` обчислюється так: спочатку складаються змінні `a2 + a3`, а потім результат порівнюється з змінної `a1`.

4.3. Логічні оператори

У мові C++ використовуються три логічних оператори: && (AND), || (OR) і ! (NOT). Таблиця істинності для логічних операторів має наступний вид.

x	y	x&&y	x||y	!x
false	false	false	false	true
false	true	false	true	true
true	true	true	true	false
true	false	false	true	false

Правила мови дозволяють поєднувати в одному виразі декілька операторів. Скористаємося цим для того, щоб обчислити значення оператора XOR. Результат істинний, якщо тільки один з операндів є істинним.

XOR

```
#include <iostream.h>

int main()
{
    bool x,y,z;
    x = true;
    y = true;
    z =  (x || y) && !(x && y);
    cout << x << " " << y << " " << z << endl;

    x = true;
    y = false;
    z =  (x || y) && !(x && y);
    cout << x << " " << y << " " << z << endl;

    x = false;
    y = true;
    z =  (x || y) && !(x && y);
    cout << x << " " << y << " " << z << endl;

    x = false;
    y = false;
    z =  (x || y) && !(x && y);
    cout << x << " " << y << " " << z << endl;

    return 0;
}
```

Результати обчислення операції XOR приведені нижче.

```
1 1 0
1 0 1
0 1 1
0 0 0
```

Відношення між операторами порівняння і логічних операторів непрості: не всі логічні оператори мають більш низький пріоритет, ніж оператори порівняння. Вищий пріоритет має оператор `!`, за ним у порядку убування пріоритету слідують оператори `>`, `>=`, `<`, `<=`, `==`, `!=`, `&&` і `||`.

Дужки дозволяють змінити порядок обчислення виразів, що містять логічні оператори. Наприклад, якщо забрати дужки з вираз, що обчислює операцію XOR, результати стануть зовсім іншими.

Вплив дужок на порядок обчислень

```
#include <iostream.h>

int main()
{
    bool x,y,z;
    x = true;
    y = true;
    z = x || y && !x && y;
    cout << x << " " << y << " " << z << endl;

    x = true;
    y = false;
    z = x || y && !x && y;
    cout << x << " " << y << " " << z << endl;

    x = false;
    y = true;
    z = x || y && !x && y;
    cout << x << " " << y << " " << z << endl;

    x = false;
    y = false;
    z = x || y && !x && y;
    cout << x << " " << y << " " << z << endl;

    return 0;
}
```

Результати роботи цієї програми приведені нижче.

```
 1 1 1
 1 0 1
 0 1 1
 0 0 0
```

Особливо цікавою властивістю виразів, що містять логічні оператори, є *скорочені обчислення*. Наприклад, якщо в якийсь момент результат стає очевидним, обчислення припиняються. Таким чином, коли булеві змінні `x` і `y` дорівнюють `false`, значення вираз `x || y` дорівнює `false`, і вираз `(x || y) && !(x && y)` також дорівнює `false` незалежно від значення виразу `!(x && y)`. Отже, обчислення останнього виразу стає зайвим і не виконується.

4.4. Побітові оператори

Числа в комп'ютері подаються в двійковій системі числення. Для зручності використання вони конвертуються в десяткову, восьмеричну чи шістнадцатерічну систему числення і надалі обробляються у відповідності зі своїм типом — `char`, `int`, `float`, `double`, `long double`. Тому досить дивно, що в мові C++ немає окремого типу для представлення двійкових чисел. Однак цю незручність легко перебороти, запам'ятавши нескладну таблицю. Уявіть собі 32-розрядне ціле число.

```
  0 1 0 1 1 1 0 1 0 1 1 1 1 0 0 1 1 0 1 1 1 0 1 1 1 0 0 0 1 0 0 1
  ↑                                                             ↑
 32-й біт                                                    1-й біт
```

Запам'ятати це число і працювати з ним довольно складно. Для того щоб спростити нашу задачу, розіб'ємо набір нулів і одиниць на байти (8 біт) і напівбайти (4 біт) і скористаємося наступною таблицею.

Напівбайт	Десяткове значення	Шістнадцатерічне значення
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6

0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Зовсім очевидно, що запам'ятати цю таблицю набагато простіше. Тепер двійкове число в 32-розрядному комп'ютері можна подати в такому виді.

```
  0 1 0 1 1 1 0 1 0 1 1 1 1 0 0 1 1 0 1 1 1 0 1 1 1 0 0 0 1 0 0 1
 |       |       |       |       |       |       |       |       |
    5       D       7       9       B       B       8       9
```

Таким чином, дане число в шістнадцятирічному виді дорівнює 0x5D79BB89. Звичайно, такий запис також виглядає досить громіздко, але це краще, ніж сукупність тридцяти двох нулів і одиниць.

І все ж комп'ютер виконує обчислення саме над двійковими числами, тому в мові C++ передбачені побітові операції, що дозволяють перевіряти і встановлювати задані розряди або множини розрядів у цілочисельних змінних. Розглянемо їх по черзі.

Порозрядний оператор `~` інвертує всі розряди свого операнда. Він називається *оператором побітового заперечення*, чи *доповненням до одиниці*. Цей оператор є унарним і префіксним. Він обнуляє кожен одиничний біт і заміняє одиницею кожен нульовий біт.

```
~0x5D79BB89 = 1010 0010 1000 0110 0100 0100 0111 0110
```

Основним призначенням операції доповнення до одиниці є інвертування логічних значень “істина” і “хибність”, а також перетворення позитивних чисел у негативні і навпаки.

Перша мета очевидна. Оскільки в мові C++ нуль трактується як “хибність”, а будь-яке ненульове значення як “істина”, заперечення нуля (`~0`) дорівнює `-1` і означає “істина”, а `~(-1)` дорівнює нулю й означає “хибність”. Варто мати на увазі, що в такому випадку програміст змушений відмовитися від широкої інтерпретації істинного значення як довільного ненульового значення й обмежитися тільки значенням `-1`.

Для перетворення позитивних чисел у негативні і навпаки до доповнення числа до одиниці додається одиниця: `~var+1`. Припустимо, що цілочисельна змінна `var` дорівнює 3. Продемонструємо перетворення двійкових чисел,.

```
  3     = 0000 0000 0000 0000 0000 0000 0000 0011
 ~3     = 1111 1111 1111 1111 1111 1111 1111 1100
 ~3 + 1 = 1111 1111 1111 1111 1111 1111 1111 1101
```

Останній двійковий запис трактується комп'ютером як додатковий код числа 3, до того ж перший біт є знаковим: 0 — плюс, 1 — мінус.

Порозрядний оператор зсуву битів ліворуч переносить усі біти на задану кількість позицій ліворуч, при цьому вивільнювані позиції заповнюються нулями, а старші біти зникають. Цей оператор є бінарним: його перший операнд — змінювана цілочисельна змінна, а другий — цілочисельна змінна, що задає величину зсуву.

Наприклад, якщо в 16-розрядному комп'ютері цілочисельні змінні `var1` і `var2`, що не мають знаку, дорівнюють 3 і 4, то вираз `var1<<var2` дорівнює 48, тобто 0000 0000 0011 0000. У двійковій системі числення зсуву біта на одну позицію ліворуч еквівалентний множенню на 2. Таким чином, `3<<4 = 3*2*2*2*2 = 3*16 = 48`.

Аналогічно порозрядний оператор зсуву бітів праворуч переносить усі біти на задану кількість позицій праворуч, при цьому вивільнювані позиції праворуч заповнюються нулями, а молодші біти зникають. Цей оператор також бінарний: його першим операндом є змінювана цілочисельна змінна, а другим — цілочисельна змінна, що задає величину зсуву.

Наприклад, якщо в 16-розрядному комп'ютері цілочисельні змінні `var1` і `var2`, що не мають знаку, дорівнюють 32 і 4, то вираз `var1>>var2` дорівнює 2, тобто 0000 0000 0000 0010. У двійковій системі числення зсуву бита на одну позицію праворуч еквівалентний діленню на 2. Таким чином, `32>>4 = 32/(2*2*2*2) =32/16 = 2`.

Порозрядна операція & (AND) є бінарною. Вона задається наступною таблицею істинності.

Біт1	Біт2	Біт1 & Біт2
0	0	0
0	1	0
1	0	0
1	1	1

Як бачимо, операція & може виявитися корисною для фільтрування одиниць, що входять у двійкового подання вихідного числа. У цьому випадку другий операнд називається *маскою*. При виконанні операції & у результат копіюються всі нульові розряди вихідного числа, а одиниці копіюються, тільки якщо вони

відповідають одиницям заданої маски. Наприклад, у 16- розрядному комп'ютері результат вираз 3&4 виглядає в такий спосіб.

```
3    = 0000 0000 0000 0101
4    = 0000 0000 0000 1000
3&4  = 0000 0000 0000 0000
```

Порозрядна операція | (OR) також є бінарною. Вона задається наступною таблицею істинності.

Біт1	Біт2	Біт1 | Біт2
0	0	0
0	1	1
1	0	1
1	1	1

Очевидно, що операція | дозволяє вставляти в результат необхідні розряди. При виконанні даної операції в результат копіюються всі одиничні розряди вихідного числа, а нулі копіюються, тільки якщо вони відповідають нулям заданої маски. У 16-розрядному комп'ютері результат вираз 3|4 виглядає в такий спосіб.

```
3    = 0000 0000 0000 0101
4    = 0000 0000 0000 1000
3|4  = 0000 0000 0000 1101
```

Порозрядна операція ^ (XOR) є бінарної. Вона задається наступною таблицею істинності.

Біт1	Біт2	Біт1 || Біт2
0	0	0
0	1	1
1	0	1
1	1	0

Якщо два розряди операндів дорівнюють, що результируючий біт дорівнює нулю, а якщо відрізняються — одиниці. Таким чином, одиниця в одному з операндів інвертує розряд в іншому. У 16-розрядному комп'ютері результат вираз 3^4 виглядає в такий спосіб.

```
3    = 0000 0000 0000 0101
4    = 0000 0000 0000 1000
3^4  = 0000 0000 0000 1101
```

Побітові оператори, як правило, використовуються для того, щоб установити чи скинути визначений біт чи групу бітів, а також для швидкого множення чи ділення на 2. Нагадаємо, що всі операнди побітових операторів повинні бути цілочисельними.

Наприклад, щоб перевірити, чи установлений визначений біт, варто виконати наступну програму (на 16-розрядному комп'ютері).

Перевірка встановленого біта

```
#include <iostream.h>

int main()
{
    int a = 32;
    cout << " Чи встановлений другий біт?" << endl;
    if (a & 0x02)  // 0000 0000 0010 0000 = 32
                   // 0000 0010 0000 0010 =  2
    cout << "Біт установлений.";
    else cout << "Біт скинутий";
    return 0;
}
```

Щоб установити чи скинути необхідний біт, можна скористатися наступними операторами.

Установка (скидання) біта

```
#include <iostream.h>

int main()
{
    int a = 32;
    cout << "Установлюємо другий біт." << endl;
    a= a ^ 0x02;   // 0000 0000 0010 0000 = 32
                   // 0000 0000 0000 0010 =  2
                   // 0000 0000 0010 0010 = 32 ^ 2
    cout << "Біт установлений.";
    return 0;
}
```

4.5. Оператори присвоювання

Без оператора присвоювання неможливо уявити собі жодну програму. Найпростіша форма оператора присвоювання виглядає так.

```
<lvalue> = <rvalue>;
```

У лівій частині оператора присвоювання, як правило, стоїть змінна, а в правої — вираз. У принципі, будь-яке присвоювання являє собою копіювання значення змінної `rvalue` у змінну `lvalue`. У найпростішому випадку саме так всі і відбувається. Однак якщо вираз, що стоїть в правій частині, описує деяке обчислення, то його результат записується в тимчасовий змінну, значення якої і копіюється в комірку `rvalue`. Отже, оператор присвоювання повинний виконуватися останнім — після обчислення правої частини. Саме тому цей оператор має найменший пріоритет.

Крім того, якщо типи змінних `lvalue` і `rvalue` є різними, вступають у дію правила приведення типів.

Значенням оператора присвоювання є значення змінної `rvalue` (чи відповідного виразу). Ця особливість дозволяє застосовувати *множинне присвоювання*.

```
var1 = var2 = var3 = 100;
```

У програмах часто приходиться збільшувати деяку змінну на фіксовану величину. Якщо збільшення `delta` дорівнює 1, природно застосувати оператор інкрементації. У противному випадку приходиться писати наступний оператор.

```
var1 = var1 + delta;
```

На перший погляд, у цьому виразі немає нічого особливого. Однак якщо придивитися, ми помітимо, що для виконання присвоювання приходиться виконувати зайву роботу.

1. Обчислити адреса змінної `var1`.
2. Витягти значення цієї змінної.
3. Обчислити значення вираз `var1 + delta`.
4. Помістити це значення в тимчасову змінну.
5. Обчислити адреса змінної `var1`.
6. Скопіювати значення тимчасової змінної в змінну `var1`.

Як бачимо, пп. 1 і 5 збігаються. Для того щоб уникнути цього, у мові C++ передбачена *скорочена форма* операторів присвоювання. Особливо важливою властивістю цієї форми є те, що вона дозволяє сполучити оператор присвоювання з будь-яким арифметичним і побітовим оператором. Перелічимо ці скорочені оператори.

Скорочений оператор	Зміст
`a+=b`	`a=a+b`
`a-=b`	`a=a-b`
`a*=b`	`a=a*b`
`a/=b`	`a=a/b`
`a%=b`	`a=a%b`
`a<<=b`	`a=a<<b`
`a>>=b`	`a=a>>b`
`a&=b`	`a=a&b`
`a|=b`	`a=a|b`
`a^=b`	`a=a^b`

Нескладні експерименти показують, що така форма оператора присвоювання не завжди виправдує себе. Простіше говорячи, ці правила досить сильно залежать від конкретної реалізації. Розглянемо наступну програму.

Скорочена форма присвоювання

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

int main()
{
    time_t time0, time1;
    double diff;
    int a=1;

    time( &time0 );

    // Повна форма оператора присвоювання
    for(long i=1; i<=200000000; i++) a*=2;
    time( &time1);
    diff = difftime( time1, time0);
```

```
    printf("Повна форма:      %8.3f з\n",diff);

    // Скорочена форма оператора присвоювання
    time( &time0 );
    for(long i=1; i<=200000000; i++) a=a*2;
    time( &time1 );
    diff = difftime( time1, time0);
    printf("Скорочена форма: %8.3f с\n",diff);

    return 0;
}
```

Результати виконання цієї програми такі.

```
Повна форма:        9.000 с
Скорочена форма:    8.000 с
```

Як бачимо, скорочена форма виконується приблизно на 10% швидше. Втім, для того щоб ефективність скороченої форми оператора присвоювання проявилася більш-менш яскраво, потрібно виконати десятки мільйонів операцій. Крім того, реалізація скороченої форми присвоювання залежить від компілятора: в одному випадку виграш досягає 10%, в іншому — час виконання повної і скороченої форми присвоювання зовсім однаковий.

4.6. Резюме

- *Вираз* — це основний засіб обчислень, що складається з операторів і операндів. Порядок обчислення виразів визначається правилами асоціативності і пріоритетами операторів.
- Оператор може бути *унарним*, тобто з одним операндом, *бінарним*, що має два операнди, і *тернарним*, у якого є три операнди.
- Арифметичні оператори бувають префіксними, інфіксними і постфіксними. Префіксні оператори завжди передують своїм операндам, інфіксні записуються між операндами, а постфіксні — після них. Наприклад, +1 — префіксна форма запису, 2+3 — інфіксна, а 23+ — постфіксна (так називаний *польський запис*).
- Постфіксна форма операторів інкремента і декремента як результат повертає *константу*, збільшену чи зменшену на одиницю. Отже, постфіксну форму цих операторів неможливо застосувати дещо разів подряд: `c++++ // не можна`.
- Префіксна форма операторів інкремента і декремента як результат повертає *посилання*, збільшене чи зменшене на одиницю. Отже, префіксну форму цих операторів можна застосувати декілька разів поспіль: `++++c // можна`.
- *Пріоритети операторів* визначають порядок, у якому виконуються оператори в арифметичних виразах.
- Якщо в операторі присвоювання типи лівого і правого операндів відрізняються, то значення правого операнда приводиться до типу лівого операнда оператора присвоювання. Якщо тип лівого операнда вимагає для свого представлення менше битов, ніж тип правого операнда, перетворення називається *звужуючим*. В іншому випадку воно іменується *розширювальним*.
- *При звужуючому перетворенні інтегральних типів у результат копіюються лише молодші біти, а зайві старші біти ігноруються*. У 16-розрядній операційній системі при перетворенні `int->char` губляться старші вісім розрядів, а при перетворенні `long->char` — 24 біт. При трансформації `long- >int` відбувається втрата 16 старших біт. У 32-розрядних системах розміри типів `long` і `int` збігаються, отже, утрата розрядів відбувається лише під час перетворення `int->char` і дорівнює 24 біт.
- Найпростіше перетворення відбувається між типами `int` і `float`. Якщо змінна типу `float` привласнюється змінної типу `int`, дробова частина відкидається, при зворотному присвоюванні до цілочисельної змінної приписується нульова дробова частина.
- У ході обчислення змішаних арифметичних виразів перетворення підкоряються наступному правилу: *типи операндів перетворяться з більш вузьких у більш широкі*. Це явище називається *розширенням типів*. Правило розширення можна сформулювати так: *тип результату збігається із самим широким типом операнда*.
- У мові C++ передбачено шість операторів порівняння: `<` (менше), `<=` (менше чи дорівнює), `>` (більше), `>=` (більше чи дорівнює), `==` (дорівнює), `!=` (не дорівнює). Оператори порівняння носять логічний характер, тобто результатом є “істина” і “хибність”. У мові C++ ці поняття мають двоїстий характер. По-перше, істинним вважається будь-яке значення, що не дорівнює нулю, а помилкове значення завжди дорівнює 0. По-друге, у C++ існує особливий тип даних `bool` і булеві константи `true` і `false`. У ході виконання операторів порівняння значення 0 автоматично перетвориться в константу `false`, а будь-яке ненульове значення — у константу `true`. І навпаки, константа `true` перетвориться в значення 1, а константа `false` — у число 0.

- Оператори порівняння мають властивість лівої асоціативності, отже, порядок их обчислення такий: спочатку обчислюється крайній ліворуч оператор `a1 > a2`, а потім його результат порівнюється із змінною `a3`. Відзначимо, що змінні `a2` і `a3` не порівнюються. Зрозуміло, цей порядок можна змінити, застосувавши дужки: `a1 > (a2 > a3)`.
- Оператори порівняння мають більш низький пріоритет, ніж арифметичні оператори. Таким чином, вираз `a1 > a2 + a3` обчислюється так: спочатку складаються змінні `a2 + a3`, а потім результат порівнюється з змінної `a1`.
- У мові C++ використовується три логічних оператори: && (AND), || (OR) і ! (XOR).
- Вищий пріоритет має оператор `!`, за ним у порядку убування пріоритету слідують оператори `>`, `>=`, `<`, `<=`, `==`, `!=`, && і ||.
- У мові C++ передбачені побітові операції, що дозволяють перевіряти і встановлювати задані розряди або множини розрядів у цілочисельних змінних.
- Порозрядний оператор ~ інвертує всі розряди свого операнда. Він називається *оператором побітового заперечення*, чи *доповненням до одиниці*. Цей оператор є унарним і префіксним. Він обнуляє кожен одиничний біт і заміняє одиницею кожен нульовий біт.
- Порозрядний оператор зсуву бітів ліворуч << переносить усі біти на задану кількість позицій ліворуч, при цьому вивільнювані позиції заповнюються нулями, а старші біти зникають. Цей оператор є бінарним: його перший операнд — змінювана цілочисельна змінна, а другий — цілочисельна змінна, що задає величину зсуву.
- Порозрядний оператор зсуву бітів праворуч >> переносить усі біти на задану кількість позицій праворуч, при цем вивільнювані позиції праворуч заповнюються нулями, а молодші біти зникають. Цей оператор також бінарний: його першим операндом є змінювана цілочисельна змінна, а другим — цілочисельна змінна, що задає величину зсуву.
- Операція & може виявитися корисною для фільтрування одиниць, що входять у двійкового подання вихідного числа. У цьому випадку другий операнд називається *маскою*. При виконанні операції & у результат копіюються всі нульові розряди вихідного числа, а одиниці копіюються, тільки якщо вони відповідають одиницям заданої маски.
- Операція | дозволяє вставляти в результат необхідні розряди. При виконанні даної операції в результат копіюються всі одиничні розряди вихідного числа, а нулі копіюються, тільки якщо вони відповідають нулям заданої маски.
- Поразрядна операція ^ (XOR) є бінарною. Якщо два розряди операндів дорівнюють один одному, то результуючий біт дорівнює нулю, а якщо відрізняються — одиниці. Таким чином, одиниця в одному з операндів інвертує розряд в іншому.
- Найпростіша форма оператора присвоювання виглядає так: `<lvalue> = <rvalue>;` У лівій частині оператора присвоювання, як правило, стоїть змінна, а в правої — вираз. У принципі, будь-яке присвоювання являє собою копіювання значення змінної `rvalue` у змінну `lvalue`. У найпростішому випадку саме так всі і відбувається. Однак якщо вираз, що стоїть в правій частині, описує деяке обчислення, то його результат записується в тимчасовий змінну, значення якої і копіюється в комірку `rvalue`. Отже, оператор присвоювання повинний виконуватися останнім — після обчислення правої частини. Саме тому цей оператор має найменший пріоритет.
- Якщо типи змінних — `lvalue` і `rvalue`, вступають у дію правила приведення типів.
- Значенням оператора присвоювання є значення змінної `rvalue` (чи відповідного виразу). Ця особливість дозволяє застосовувати *множинне присвоювання*: `var1 = var2 = var3 = 100;`
- У мові C++ передбачена *скорочена форма* операторів присвоювання. Особливо важливою властивістю цієї форми є те, що вона дозволяє сполучити оператор присвоювання з будь-яким арифметичним і побітовим оператором.

4.7. Контрольні питання

- Що таке вираз?
- Як оператори класифікуються за кількістю операндів?
- Назвіть різновиди арифметичних операторів.
- Що є характерним для постфіксної форми операторів інкремента і декремента?
- Що визначає пріоритет оператора?
- Як приводяться типи лівого і правого операндів оператора присвоювання?
- Які перетворення використовуються при цьому?
- Опишіть звужуюче перетворення інтегральних типів.
- Опишіть перетворення типу `int` у тип `float`.
- Сформулюйте правило перетворення типів під час обчислення змішаних арифметичних виразів.
- Назвіть шість операторів порівняння і опишіть їхні особливості.

- Назвіть логічні оператори і опишіть їх особливості.
- Назвіть порозрядні оператори у мові C++.
- Опишіть порозрядний оператор ~.
- Опишіть порозрядний оператор зсуву бітів ліворуч <<.
- Опишіть порозрядний оператор зсуву бітів праворуч >>.
- Опишіть оператор &.
- Опишіть оператор |.
- Опишіть оператор ^.
- Опишіть оператор присвоювання.
- Опишіть скорочені форми операторів присвоювання. Назвіть їх особливості.