Лекція 8

Перевантаження операторів

У цій лекції...

8.1. Операторні функції
8.2. Перевантаження унарних операторів за допомогою функцій-членів
8.3. Перевантаження бінарних операторів за допомогою функцій-членів
8.4. Перевантаження унарних операторів за допомогою дружніх функцій
8.5. Перевантаження бінарних операторів за допомогою дружніх функцій
8.6. Перетворення типів

У математиці зміст операції залежить від її операндів. Якщо `a` і `b` є цілими числами, їх сума `a+b` обчислюється по одним правилах, якщо матрицями — по іншим. Людина, що знає природу операндів, не випробує утруднень, інтерпретуючи зміст операції `+`.

Як відомо, кожен убудований тип даних зв'язаний з визначеним набором операцій, які можна до нього застосовувати. Ці операції позначаються відповідними символами, зміст яких відомий компілятору заздалегідь. Однак цілком природно спробувати розширити застосування операторів на класи, створювані програмістом. Наприклад, матричні обчислення стають набагато наочніше, якщо сума двох матриць записується як `a+b`, а не як `summa(a,b)`. Для цього в мові C++ існує механізм *перевантаження операторів*.

8.1. ОПЕРАТОРНІ ФУНКЦІЇ

Спочатку необхідно відзначити, що не всі символи операцій можна перевантажити. Нижче перераховані оператори, дозволені до перевантаження.

```
+       -       *       /       %       ^       &
|       ~       !       =       <       >       +=
-=      *=      /=      %=      ^=      &=      |=
<<      >>      >>=     <<=     ==      !=      <=
>=      &&      ||      ++      --      ->*     ,
->      []      ()      new     new[]   delete  delete[]
```

Існують також оператори, заборонені до перевантаження. Зміна їх змісту зруйнувало би логіку програми. До таких операторів належать `::` (оператор дозволу області видимості), `.` (“точка” — оператор доступу до члена класу), `?:` (тернарний оператор), `.*` (доступ до розіменованого вказівника-члена класу), `sizeof`, `typeid`, `static_cast`, `dynamic_cast`, `const_cast` і `reinterpret_cast`. Крім того, не рекомендується перевантажувати логічні оператори `&&` і `||`, оскільки на їхні перевантажені версії не поширюється правило скорочених обчислень логічних виразів. (Нагадаємо, що це правило полягає в наступному: якщо на деякому етапі значення усього вираження стає визначеним, подальші обчислення припиняються.)

Синтаксис операторних функцій виглядає в такий спосіб.

```
тип_значення_що_повертається operator символ_операції(параметри)
{
   ...
}
```

Наприклад, операторна функція, що перевантажує операцію `+`, називається `operator+()`.

Операторні функції повинні мати прямий доступ до членів класу. Отже, необхідно, щоб вони були або членами класу, або дружніми функціями.

Необхідно пам'ятати про обмеження, що супроводжують застосування перевантажених операторів.

1. Перевантажені функції не можуть змінити пріоритет операторів.
2. Кількість операндів фіксована: жодного, один чи два.
3. Значення операндів не можна задавати за замовчуванням.

8.1. ПЕРЕВАНТАЖЕННЯ УНАРНИХ ОПЕРАТОРІВ ЗА ДОПОМОГОЮ ФУНКЦІЙ-ЧЛЕНІВ

Оператори можуть бути унарними і бінарними. Унарний оператор має один операнд, а бінарний — два. Нагадаємо, що до унарних операторів, що перевантажуються, належать такі оператори, як `+`, `-`, `++`, `--`, `&`, `~` і `!`. До бінарних операторів, що перевантажуються, належать всі інші оператори, перераховані в приведеній вище таблиці.

Операторні функції-члени, що перевантажують унарний оператор, мають одну особливість: їх операнди передаються неявно за допомогою вказівника `this`. Отже, така функція-член класу не має явних параметрів.

8.2.1. Унарні оператори “плюс” і “мінус”

Розглянемо приклад перевантаження унарного мінуса. (Операція унарного плюса є “порожній”. Вона включена в мову для симетрії.)

Перевантаження унарних операторів “плюс” і “мінус”

```
  #include <stdio.h>

 class TComplex
 {
    double Re;
    double Im;
 public:
    TComplex(double x, double y):Re(x),Im(y){ }
    TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
    ~TComplex(){}
    void print();
    TComplex operator-() {Re = -Re; Im = -Im; return *this;}
 };

 int main()
 {
    TComplex z(1,1),u(0,0);
    z.print();
    u=-z;
    u.print();
    return 0;
 }

 void TComplex::print()
 {
  printf("%lf + i*%lf\n", Re, Im);
 }
```

Помітимо, що операторна функція може повертати об'єкт класу (точніше, розіменований вказівник `this`) чи посилання на об'єкт, а може і нічого не повертати. Вибираючи тип значення, що повертається, необхідно керуватися здоровим глуздом. Якщо унарний оператор повинний використовуватися усередині виразів, то операторна функція повинна повертати чи об'єкт посилання. Якщо ж оператор використовується ізольовано, функція може нічого не повертати.

Рядок програми

```
u = - z;
```

можна переписати в еквівалентному виді.

```
u = z.operator-();
```

Цей рядок демонструє, що виклик операторної функції `operator-()` здійснюється об'єктом `z`.

Варто мати на увазі, що хоча зміст оператора перевантажувати можна, його природу змінювати заборонено. Це значить, що унарні оператор не можна перевантажити як бінарний і навпаки.

Розглянемо найбільш важливі приклади унарних операторів.

8.2.2. Оператори інкремента і декремента

У мові C++ передбачено дві форми операторів інкремента і декремента: префіксна і постфіксна. Для того щоб розрізняти їх, використовується звичайний механізм перевантаження функцій — вводиться фіктивний цілочисловий параметр. Наприклад, для класу `TComplex` прототипи відповідних операторних функцій можуть виглядати в такий спосіб.

```
   TComplex operator++();          // Префіксна форма
 TComplex operator++(int x);    // Постфіксна форма
 TComplex operator--();         // Префіксна форма
TComplex operator--(int x);     // Постфіксна форма
```

Спробуємо розібратися на конкретному прикладі.

Перевантаження постфіксної і префіксних форм інкрементації

```
  #include <stdio.h>

 class TComplex
 {
    double Re;
    double Im;
 public:
    TComplex(double x, double y):Re(x),Im(y){ }
    TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
    ~TComplex(){}
    void print();
    TComplex& operator++()
    {
```

```
        ++Re;
        ++Im;
        printf("Префіксна форма ++ \n");
        return *this;
    }
    const TComplex operator++(int i)
    {
        ++Re;
        ++Im;
        printf("Постфіксна форма ++ %d\n",i);
        return *this;
    }
    TComplex& operator--()
    {
        --Re;
        --Im;
        printf("Префіксна форма -- \n");
        return *this;
    }
    const TComplex operator--(int i)
    {
        --Re;
        --Im;
        printf("Постфіксна форма -- %d\n",i);
        return *this;
    }
};

int main()
{
   TComplex z(1,1);
   ++z;
   z.print();
   z++;
   z.print();
   --z;
   z.print();
   z--;
   z.print();
   return 0;
}

void TComplex::print()
{
 printf("%lf + i*%lf\n", Re, Im);
}
```

У функції `main()` до об'єкта `z` послідовно застосовуються префіксна і постфіксна форми операторів `++` і `--`.

```
   Префіксна форма ++
2.000000 + i*2.000000
Постфіксна форма ++ 0
3.000000 + i*3.000000
Префіксна форма --
2.000000 + i*2.000000
Постфіксна форма -- 0
1.000000 + i*1.000000
```

Якщо символ операції `++` стоїть перед операндом, викликається операторна функція `operator++()`, якщо після — операторна функція `operator++(int i)`. Змінна `i` відіграє роль прапора, що повідомляє компілятору, що дана функція перевантажує постфіксну форму оператора інкремента і декремента.

Незважаючи на те що програміст вільний вільно трактувати перевантажений оператор, прагнучи зберегти аналогію з його убудованими аналогами, необхідно дотримувати визначені правила. Наприклад, як відомо, оператор інкрементації цілих чисел повертає посилання на неконстантний об'єкт. Ця властивість повинна зберігатися і при перевантаженні.

```
   TComplex& operator++()
{
  ++Re;
```

```
  ++Im;
  printf("Префиксна форма ++ \n");
  return *this;
 }
```

У той же час постфиксний оператор повертає константне значення. Саме тому в C++ неможливі вираження x++++. Цілком природно зажадати, щоб перевантажений оператор мав таку ж властивість.

```
  const TComplex operator++(int i)
{
  ++Re;
  ++Im;
  printf("Постфиксна форма ++ %d\n",i);
  return *this;
 }
```

Крім того, оскільки у вихідному варіанті постфиксний оператор іінкрементації реалізується через префиксний, це також бажано врахувати при перевантаженні. У цьому випадку реалізація операторної функції ++ для постфиксної форми може виглядати так.

```
  const TComplex operator++(int i)
{
  TComplex Z;
  ++*this;
  printf("Постфиксна форма ++ %d\n",i);
  return Z;
 }
```

8.2.3. Унарні оператори !, & і ~

Оператори заперечення (!), узяття адреси (&) і побітового заперечення (~) допускають перевантаження, але не мають універсальних альтернатив, що варто було б реалізувати. Їх можна перевантажувати, наприклад, для підвищення наочності програми. Скажемо, за допомогою оператора ! можна позначати операцію звертання матриці, а за допомогою символу ~ — її транспонування. Щоправда, застосування тильди закріплене за деструкторами, тому варто виявляти обережність, щоб не створити плутанину. У будь-якому випадку зміст перевантаження операторів залежить від конкретної задачі.

8.2.4. Перевантаження оператора ->

Оператор *посилання на член об'єкта* є унарним. Операторна функція, що перевантажуєйого, виглядає в такий спосіб.

об'єкт `->` *елемент*

Цей запис еквівалентний наступному вираженню.

об'єкт`.operator->(`*елемент*`);`

Функція `operator- >()` повинна бути нестатичним членом класу. Як параметр вона одержує об'єкт чи класу посилання на нього, повертаючи вказівник `this` на об'єкт, де виконується виклик, або посилання на об'єкт будь-якого іншого класу, у якому визначений оператор `->`. її зручно використовувати в контейнерних класах, що містять усередині себе вказівник на інший клас. Основний зміст перевантаження оператора `->` полягає в додатковій функціональності, що розширює можливості звичайних вказівників.

Наприклад, у приведеній нижче програмі функція `operator- >()` веде підрахунок посилань на кожен об'єкт класу.

Перевантаження оператора ->

```
  #include <stdio.h>

 class TClass
 {
    int n;
    int counter;
 public:
    TClass(int x):n(x),counter(0) { }
    TClass* operator->();
    int get(void) { return n;}
    int ref(void) { return counter; }
 };

TClass* TClass::operator ->()
 {
    counter++;
    return this;
 }
```

```
int main()
{
   TClass a(1), b(2);
   printf("n = %d \n",a->get());
   printf("n = %d \n",b->get());
   printf("n = %d \n",a->get());
   printf("counter = %d \n",a->ref());
   printf("counter = %d \n",b->ref());
   return 0;
}
```

У результаті роботи програми на екран виводяться наступні рядки.

```
n = 1
n = 2
n = 1
counter = 3
counter = 2
```

Зверніть увагу на те, що функція `operator- >()` у даному прикладі повертає значення типу `TClass*`.

Розглянемо тепер програму, у якій створюється контейнер, що містить масив вказівників на об'єкти класу `TClass`.

```
#include <stdio.h>

class TClass
{
   int n;
   int counter;
public:
   TClass(int x):n(x),counter(0) { }
   TClass* operator->();
   int get(void) { return n;}
   int ref(void) { return counter; }
};

TClass* TClass::operator ->()
{
   counter++;
   return this;
}

class TContainer
{
public:
   int m;
   TClass* p[];
   TContainer(int k, TClass* q[])
   {
      m = k;
      for(int i=0; i<m; i++)p[i]=q[i];
   }
   TContainer* operator->();
};

TContainer* TContainer::operator ->()
{
   return this;
};

int main()
{
   int m = 2;
   TClass a(1), b(2);
   TClass* p[]={&a, &b};
   TContainer c(m,p);
   for(int i=0; i<m; i++) printf("%ld \n",c->p[i]->get());
   return 0;
}
```

Оскільки й у класі `TContainer`, і в класі `TClass` міститься перевантажена версія оператора `->`, ми можемо скористатися ланцюжком операторів `c->p[i]->get()`, що забезпечує перегляд умісту контейнера.

8.2. ПЕРЕВАНТАЖЕННЯ БІНАРНИХ ОПЕРАТОРІВ ЗА ДОПОМОГОЮ ФУНКЦІЙ-ЧЛЕНІВ

Бінарний оператор має два операнди. Його виклик виконується об'єктом, розташованим у лівій частині оператора. Отже, бінарний оператор

```
a+b
```

еквівалентний такому оператору.

```
a.operator+(b)
```

Таким чином, бінарна операторна функція-член повинна має тільки один параметр, що задає другий операнд Вказівник `this` на перший операнд вона одержує неявно.

Для того щоб бінарну операторну функцію можна було застосовувати усередині виразів, необхідно, щоб вона повертала об'єкт свого класу. Розглянемо приклад.

Перевантаження бінарного оператора +

```
#include <stdio.h>

class TComplex
{
   double Re;
   double Im;
public:
   TComplex(double x, double y):Re(x),Im(y){ }
   TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
   ~TComplex(){}
   void print();
   TComplex operator+(TComplex z)
   {
       TComplex w(0,0);
       w.Re = Re+z.Re;
       w.Im = Im+z.Im;
       return w;
   }
};


int main()
{
   TComplex u(1,1),v(2,2),z(0,0);
   z=u+v;
   z.print();
   return 0;
}

void TComplex::print()
{
   printf("%lf + i*%lf\n", Re, Im);
}
```

У цій програмі сума комплексних чисел `u` і `v` привласнюється об'єкту `z`. Виклик операторної функції `operator+()` здійснюється з об'єкта `u`.

Оскільки сума двох об'єктів класу `TComplex` є об'єктом цього ж класу, не обов'язково створювати новий об'єкт і записувати в нього результат підсумовування. Можна скористатися наступною синтаксичною конструкцією.

```
(u+v).print();
```

У цьому випадку функція `print()` буде викликатися тимчасовим об'єктом, створеним операторної функцією `operator+()`. Цікаво, що цей механізм допускає вживання безглуздих виразів. Наприклад, компілятор не заперечує проти такого оператора.

```
(u+v)=z;
```

Для того щоб запобігти це, оператор + повинний повертати константний об'єкт. Крім того, бажано, щоб він не змінював другий операнд. Отже, параметр повинний бути константним. І останнє: тому що об'єкт може бути досить громіздким, його варто передавати за значенням. Отже, одержуємо наступну реалізацію.

```
const TComplex operator+(const TComplex& z)
{
    TComplex w(0,0);
    w.Re = Re+z.Re;
    w.Im = Im+z.Im;
```

```
    return w;
}
```

8.3.1. Перевантаження арифметичних операторів

Продемонструємо, як здійснюються ланцюжки обчислень над комплексними числами.

Арифметичні дії над комплексними числами

```
   #include <stdio.h>

class TComplex
{
   double Re;
   double Im;
public:
   TComplex(double x, double y):Re(x),Im(y){ }
   TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
   ~TComplex(){}
   void print();
   TComplex operator+(const TComplex& z)  // Додавання
   {
       TComplex w(0,0);
       w.Re = Re+z.Re;
       w.Im = Im+z.Im;
       printf("Operator + \n");
       return w;
   }
   TComplex operator-(const TComplex& z)  // Вирахування
   {
       TComplex w(0,0);
       w.Re = Re-z.Re;
       w.Im = Im-z.Im;
       printf("Operator - \n");
       return w;
   }
   TComplex operator*(const TComplex& z)  // Множення
   {
       TComplex w(0,0);
       w.Re = Re*z.Re-Im*z.Im;
       w.Im = Re*z.Im+Im*z.Re;
       printf("Operator * \n");
       return w;
   }
   TComplex operator-()            // Унарний мінус
   {
       Re = -Re;
       Im = -Im;
       printf("Unary operator - \n");
       return *this;
   }
   TComplex operator~()            // Сполучене число
   {
       TComplex w(0,0);
       w.Re = Re;
       w.Im = -Im;
       printf("Unary operator ~ \n");
       return w;
   }
   TComplex operator/(TComplex& z)
   {
       TComplex w(0,0);
       w=(*this)*(~z);
       w.Re = w.Re/modul2(z);
       w.Im = w.Im/modul2(z);
       printf("Operator / \n");
       return w;
   }
   double modul2(const TComplex& z)
   {
```

```
        return z.Re*z.Re+z.Im*z.Im;
    }
};

int main()
{
    TComplex u(1,1),v(2,2),z(0,0);
    z=(u+v)*u/v;
    z.print();
    return 0;
}

void TComplex::print()
{
 printf("%lf + i*%lf\n", Re, Im);
}
```

У класі `TComplex` операція розподілу двох комплексних чисел реалізована через обчислення сполученого числа (операція ~) і розподіл дійсної і мнимої частин на квадрат модуля дільника (функція `modul2()`).

От у якому порядку виконувалися зазначені оператори.

```
Operator +
Operator *
Unary operator ~
Operator *
Operator /
```

У підсумку, як і випливало очікувати, одержуємо наступне число.

```
1.500000 + i*1.500000
```

Реалізуючи ланцюжок обчислень, необхідно обережно використовувати ключове слово `const`. Безперечно, захист параметра від необережної модифікації необхідний. У той же час повернення константних об'єктів не завжди виправданий.

8.3.2. Перевантаження оператора присвоювання

У програмах, розглянутих вище, ми вільно маніпулювали об'єктами, привласнюючи їх один одному. Це відбувалося завдяки убудованому оператору присвоювання, що виконує побітове копіювання. Як і у випадку з конструктором копіювання, це — не краще рішення. Якщо об'єкт містить вказівник на деяку область пам'яті, його копія також буде посилатися на неї. Для того щоб цього не відбулося, перевантажений оператор присвоювання повинний виконувати глибоке копіювання, уникаючи подвійної адресації.

Розглянемо приклад.

Перевантаження оператора присвоювання

```
  #include <stdio.h>

class TArray
{
    int *p;
    int size;
public:
    TArray(long n, int x);
    TArray(TArray&);
    TArray& operator=(TArray& X);
    void view();
};

int main()
{
    TArray x(10,1),y(10,0);
    x.view();
    y = x;
    y.view();
    return 0;
}

TArray::TArray(long n, int x)
{
    size = n;
    p = new int[size];
    for (long i=0; i<size; i++)p[i] = x;
    printf("Адреса = %p\n",p);
```

```
}

TArray::TArray(TArray& X)
{
   size=X.size;
   p = new int[size];     // Глибоке копіювання
   for (long i=0; i<X.size; i++) p[i] = X.p[i];
   printf("\nАдрес = %p",p);
}

void TArray::view()
{
    for(long i=0; i<size; i++) printf("%d ",p[i]);
}

TArray& TArray::operator=(TArray& X)
{
   if(this == &X) return *this; // Перевірка самоприсвоювання.
   if(size==X.size)             //  Глибоке копіювання
                                // (вказівник не копіюється!)
   for (long i=0; i<X.size; i++) p[i] = X.p[i];
   else printf(" Size ! \n");
   printf("\n");
   return *this;
}
```

На екрані будуть виведені наступні рядки.

```
Адреса = 008803A0
Адреса = 00880340
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
```

Зверніть увагу на два моменти. По-перше, у перевантаженому операторі присвоювання виконується не побітове, а глибоке копіювання. Адреса об'єкта y залишається незмінним, міняється лише зміст. По-друге, у функції `operator=()` передбачена перевірка самоприсвоювання. Це дозволяє коректно обробляти безглузді вираження такого виду.

```
x = x;
```

Незважаючи на зовнішню схожість перевантаженого оператора присвоювання і конструктора копіювання, між ними існує принципова різниця. При виклику конструктора копіювання створюється новий об'єкт, що іініціалізується раніше існуючим об'єктом. При присвоюванні обидва об'єкти уже існують.

8.3.3. Перевантаження скорочених операторів присвоювання

В арифметичних обчисленнях дуже корисні скорочені оператори присвоювання. Розумно спробувати перевантажити їх для знову створюваних класів. При цьому варто врахувати обмеження, що накладаються на оператор присвоювання й арифметичних операторів. Причому, як правило, перевантажені арифметичні оператори реалізуються за допомогою скорочених операторів присвоювання.

Перевантаження скорочених операторів присвоювання

```
#include <stdio.h>

class TComplex
{
   double Re;
   double Im;
public:
   TComplex(double x, double y):Re(x),Im(y){ }
   TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
   ~TComplex(){}
   void print();
   const TComplex operator+=(const TComplex& z)  // Додавання
   {
       Re = Re+z.Re;
       Im = Im+z.Im;
       printf("Operator += \n");
       return *this;
   }
   const TComplex operator+(const TComplex& z)  // Додавання
   {
       TComplex w=*this;
```

```
        w+=z;
        printf("Operator + \n");
        return w;
    }
};

int main()
{
    TComplex u(1,1),v(2,2),z(3,3);
    u+=v;
    u.print();
    z=u+v;
    z.print();
    return 0;
}

void TComplex::print()
{
 printf("%lf + i*%lf\n", Re, Im);
}
```

8.3.4. Перевантаження оператора послідовного обчислення

Як випливає з визначення стандартної операції послідовного обчислення, її значенням є результат обчислення останнього вираження серед перерахованих у списку. Отже, головним фактором, що впливає на результат застосування цього оператора, є порядок перерахування його операндів. Однак коли оператор “,” перевантажується за допомогою операторної функції, операнди стають параметрами *функції*. Це створює проблему, тому що порядок обчислення аргументів функції стандартом мови не визначений. Таким чином, перевантаження оператора послідовного обчислення може привести до хитливої роботи програми і непередбачених ефектів.

Функція `operator,()` повинна бути нестатичним членом класу.

Продемонструємо програму, що ілюструє потенційну можливість перевантажувати оператор “,”.

Перевантаження оператора послідовного обчислення

```
  #include <stdio.h>

class TIndex
{
   int i;
public:
   TIndex(int init):i(init){}

   void print(){ printf(" Index = %d\n",i);}
   TIndex operator++(){ ++i; return *this;}
   TIndex operator--(){ --i; return *this;}
   TIndex operator,(TIndex last);
   bool operator>(TIndex rhs)
   {
     if ((*this).i > rhs.i)
         return true;
     else return false;
   }
};

TIndex TIndex::operator,(TIndex last)
{
   TIndex tmp(0);
   tmp.i = last.i;
   return tmp;
}

int main()
{
   TIndex last(0);
   for(TIndex i(0),j(5); j>0; ++i,--j)
      i.print();
   return 0;
}
```

Як і слід було очікувати, на екран виводяться такі рядки.

```
 Індекс = 0
Індекс = 1
Індекс = 2
Індекс = 3
Індекс = 4
```

Лівий операнд передається операторної функції неявно — за допомогою вказівника this. Функція ігнорує його значення, повертаючи значення правого операнда.

8.3.5. Перевантаження операторів new і delete

Програміст може керувати виділенням пам'яті, перевантажуючи оператори `new` і `delete`. Перевантажена операторна функція має наступний вид.

```
void* operator new(size_t size);
```

Вона виділяє `size` байт пам'яті і повертає адресу виділеної пам'яті. Конструктор і деструктор об'єктів викликаються автоматично. Тип `size_t` є цілочисловим.

Перевантажений оператор `delete` звільняє пам'ять, виділену перевантаженим оператором `new`.

Розглянемо програму, у якій оператор `new` реалізує виділення пам'яті за допомогою функції `malloc()`. Таке перевантаження може виявитися корисної при сполученні декількох модулів, написаних на мовах C і C++. Як відомо, одночасне використання механізмів розподілу пам'яті, передбачених у мовах C і C++, тобто використання пар `new` — `free()` чи `malloc()` — `delete`, може привести до непередбачених результатів. Отже, перевантаження оператора `new`, зазначена вище, може привести різні модулі "до загального знаменника".

Розглянемо програму, у якій перевантажені оператори `new` і `delete`, що використовують функції `malloc()` і `free()`. У класі передбачений статична змінна — індикатор зайнятої пам'яті. Як тільки обсяг виділеної пам'яті перевищує припустимий, програма видає повідомлення на екран.

Перевантаження операторів new і delete

```
   #include <iostream.h>
#define MAX_SIZE 1024

class TClass
{
   char array[500];
public:
   TClass(){cout << "Ctor" << endl;}
   ~TClass(){cout << "Dtor" << endl;}
   static void* operator new(size_t);
   static void operator delete(void*);
   static long counter;
};

void* TClass::operator new(size_t size)
{
   if( counter >= MAX_SIZE - sizeof(TClass))
   {
       cout << "Пам'ять вичерпана" << endl;
       return 0;
   }
   else
   {
       counter += sizeof(TClass);
       cout << "New " << endl;
       cout << "Зайняте: " << counter << " байт" << endl;
       return malloc(size);
   }
}

void TClass::operator delete(void* p)
{
   free(p);
   counter-=sizeof(TClass);
   cout << "Delete  " << endl;
   cout << "Зайнято: " << counter << " байт" << endl;
};

long TClass::counter = 0;
```

```
int main()
{
   TClass* q[3];
   for(int i = 1; i<=3; i++) { q[i] = new TClass;}
   for(int i = 1; i<=3; i++) { delete q[i];}
   return 0;
}
```

Ця програма веде облік зайнятої пам'яті, збільшує розмір лічильника, а потім звільняє пам'ять і зменшує лічильник на відповідну величину. Зверніть увагу на те, що перевантажені оператори `new` і `delete` повинні бути статичними членами класу.

```
New
Зайнято: 500 байт
Ctor
New
Зайнято: 1000 байт
Ctor
Пам'ять вичерпана
Dtor
Delete
Зайнято: 500 байт
Dtor
Delete
Зайнято: 0 байт
```

Тепер застосування операторів `new` і `delete` до об'єктів класу `TClass` приведе до виклику перевантажених версій, а для убудованих типів використовуються звичайні варіанти цих операторів.

Оператор `new` має особливу форму перевантаження, що називається синтаксисом розміщення. Вона дозволяє створювати об'єкт, розміщаючи його в осередку з заданою адресою. Нагадаємо, що саме в цьому випадку необхідно явно викликати деструктор.

Синтаксис розміщення

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

class TClass
{
public:
   int a;
   TClass() {cout << "Ctor TClass " << endl;}
   ~TClass(){cout << "Dtor TClass " << endl;}
};

int main()
{
   char memory[sizeof(TClass)];
   void *p=memory;
   TClass* q = new(p) TClass;
   cout << q << endl;
   q->~TClass();
   return 0;
}
```

Результат роботи цієї програми виглядає так.

```
Ctor TClass
0x0065fdc4
Dtor TClass
```

8.3.6. Перевантаження операторів new[ ] і delete[ ]

При створенні власного механізму розподілу пам'яті для масивів, можна перевантажити оператори `new[]` і `delete[]`, керуючись тими ж правилами.

Перевантаження операторів new[ ] і delete[ ]

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

void* operator new[](size_t size)
{
   void *p;
   cout << "New " << endl;
```

```
    p = malloc(size);
    return p;
 }

 void operator delete[](void* p)
{
    cout << "Delete\n";
    free(p);
 }

 int main()
 {
    char* q;
    q = new char[5];
    delete[] q;
    return 0;
 }
```

Ця програма виділяє пам'ять для масиву, що складає з 5 символів, а потім звільняє її.

```
  New
 Delete
```

Як правило, перевантажені оператори `new` і `delete` генерують виняткові ситуації. Однак існують версії цих операторів, у яких генерація виняткової ситуації скасована. Технічні деталі ми розглянемо, коли вивчимо механізм виняткових ситуацій.

8.3.8. Перевантаження оператора [ ]

Бинарний оператор доступу до члена масиву перевантажується за допомогою наступної операторної функції, що повинна бути нестатичним членом класу.

```
   тип_ щоповертається_значення& ім'я_класу::operator[](інтегральний_тип i)
 {
   // ...
}
```

Відзначимо два моменти. По-перше, оскільки операція доступу застосовується до індексованих масивів, параметр операторної функції повинний бути цілочисловим. По-друге, елементи масиву можуть стояти як у лівої, так і в правій частині оператора присвоювання. Отже, функція `operator[]()` повинна повертати посилання або вказівник.

Хоча зовні функція `operator[]()` виглядає унарною, насправді вона є бінарною. Її перший параметр явно задає індекс елемента, а другий параметр, що представляє собою вказівник `this` на об'єкт,де виконується виклик, передається неявно.

От типовий приклад використання цього оператора.

Перевантаження оператора [ ]

```
   #include <stdio.h>

 class TMatrix
 {
    int n;
    int m;
    double *p;
 public:
    TMatrix(int x,int y);
    TMatrix();
    ~TMatrix()  { if (p!=NULL) delete p; p=NULL; }
    double* operator[](int i)  { return p+i*m; }
    void output();
 };

 TMatrix::TMatrix(int x,int y):n(x),m(y)
 {
    p=new double[n*m];
    for (int i=0; i<n; i++)
    for (int j=0; j<m; j++)
    if(i==j)p[i*m+j]=1; else p[i*m+j]=0;
 }

 void TMatrix::output()
 {
```

```
    for (int i=0; i<n; i++)
    {
      for (int j=0; j<m; j++)
      {
        printf("%6.3lf ", (*this)[i][j]);
      }
    printf("\n");
    }
    printf("\n");
}

int main()
{
    TMatrix c(3,3);
    c[1][2]=2.0;
    c[2][1]=2.0;
    c.output();

    return 0;
}
```

Як відомо, у мові C++ за замовчуванням не передбачена перевірка виходу індексу масиву за припустимі межі. З цієї причини визначення класу `TMatrix` варто було б доповнити генеруванням відповідних виняткових ситуацій.

8.3.8. Перевантаження оператора ()

Об'єкти, що містять операторну функцію `operator()()`, називаються *функціями-об'єктами*, чи функторами. Ця функція може одержувати довільну кількість параметрів і повертати значення будь-яких типів. Такі об'єкти бувають корисними при виконанні операцій, зв'язаних з декількома індексами. Як відзначалася вище, операторна функція `operator()()` повинна бути нестатичним членом класу.

Проілюструємо створення функторів програмою, у якій функція `operator()()` повертає заданий рядок матриці.

Перевантаження оператора ()

```
   #include <stdio.h>

class TMatrix;

class Str
{
    int m;
    double *q;
public:
    Str(int x):m(x) {q = new double[m];}
    double& operator[](int i)  { return *(q+i); }
    void output();
    friend class TMatrix;
};

class TMatrix
{
    int n;
    int m;
     double *p;
public:
    TMatrix(int x,int y);
    ~TMatrix()  { if (p!=NULL) delete p; p=NULL; }
    Str operator()(int);
    double* operator[](int i)  { return p+i*m; }
    void output();
};

TMatrix::TMatrix(int x,int y):n(x),m(y)
{
    p=new double[n*m];
    for (int i=0; i<n; i++)
    for (int j=0; j<m; j++)
    if(i==j)p[i*m+j]=1; else p[i*m+j]=0;
```

```
}

Str TMatrix::operator()(int i)
{
   Str s(3);
   for (int j=0; j < m; j++) s[j]=(*this)[i][j];
   return s;
}

void TMatrix::output()
{
   for (int i=0; i<n; i++)
   {
     for (int j=0; j<m; j++)
     {
       printf("%6.3lf ", (*this)[i][j]);
     }
     printf("\n");
   }
   printf("\n");
}

void Str::output()
{
   for(int j = 0; j < m; j++)
   printf("%6.3lf ",q[j]);
}


int main()
{
   TMatrix c(3,3);
   Str s(3);
   c[1][2]=2.0;
   c[2][1]=2.0;
   printf("Матриця:\n");
   c.output();
   printf("Рядок:\n");
   s=c(1);
   s.output();

   return 0;
}
```

Виконавши цю програму, ми одержимо наступний результат.

```
Матриця:
1.000  0.000  0.000
  0.000  1.000  2.000
  0.000  2.000  1.000

Рядок:
  0.000  1.000  2.000
```

Зверніть увагу на те, що в класі Str операторна функція operator[]() повертає посилання на число типу double. Це дозволяє використовувати вираження s[j] у лівій частині оператора присвоювання.

8.3. ПЕРЕВАНТАЖЕННЯ УНАРНИХ ОПЕРАТОРІВ ЗА ДОПОМОГОЮ ДРУЖНІХ ФУНКЦІЙ

У деяких ситуаціях перевантажені оператори повинні бути членами класу. До них належать оператори =, [], ->, new і delete. В інших випадках програміст повинний керуватися здоровим глуздом.

Перевантаження операторів у виді дружніх функцій нічим не відрізняється від перевантаження у виді членів класу, за одним виключенням — функція тепер не одержує неявний вказівник *this. Отже, всі операнди повинні бути зазначені явно.

8.4.1. Перевантаження унарного мінуса

Розглянемо приклад, у якому унарний оператор “мінус” перевантажується дружньою функцією.

Перевантаження унарного оператора “мінус” дружньою функцією

```
#include <stdio.h>

class TComplex
```

```
 {
    double Re;
    double Im;
 public:
    TComplex(double x, double y):Re(x),Im(y){ }
    TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
    ~TComplex(){}
    friend void print(TComplex z);
    friend TComplex operator-(TComplex z);
 };

 int main()
 {
    TComplex z(1,1),u(0,0);
    print(z);
    u=-z;
    print(u);
    return 0;
 }

 void print(TComplex z)
 {
    printf("%lf + i*%lf\n", z.Re, z.Im);
 }

 TComplex operator-(TComplex z)
 {
    TComplex w(0,0);
    w.Re = -z.Re;
    w.Im = -z.Im;
    return w;
 }
```

Як бачимо, на відміну від попередніх варіантів, унарний мінус перевантажується операторною функцією, що має один операнд. Отже, у тілі функції необхідно створити локальний об'єкт, а потім змінити його зміст відповідно до визначення оператора. До недоліку такого підходу варто віднести додаткові витрати пам'яті і часу при створенні локальних об'єктів.

8.4.2. Перевантаження операторів інкремента і декремента

Оскільки дружні функції не одержують вказівник `this`, операнди операторів `++` і `--` необхідно передавати по посиланню. Якщо зневажити цим правилом і передати операнд за значенням, то інкрементація і декрементація торкнуться лише копії параметра, а не оригіналу. Можна було б створити локальний об'єкт, скопіювати туди значення параметра, а потім повернути його в модуль, звідки виконується виклик, але це занадто неефективно. Передача параметра по посиланню усі спрощує.

Перевантаження операторів інкремента і декремента

```
  #include <stdio.h>

class TComplex
{
    double Re;
    double Im;
public:
    TComplex(double x, double y):Re(x),Im(y){ }
    TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
    ~TComplex(){}
    void print();
    friend TComplex& operator++(TComplex& z);
    friend const TComplex operator++(TComplex& z, int i);
    friend TComplex& operator--(TComplex& z);
    friend const TComplex operator--(TComplex& z, int i);
    friend void print(const TComplex& z);
};

int main()
{
    TComplex z(1,1);
    ++z;
```

```
    print(z);
    z++;
    print(z);
    --z;
    print(z);
    z--;
    print(z);
    return 0;
}

void print(const TComplex& z)
{
  printf("%lf + i*%lf\n", z.Re, z.Im);
}

TComplex& operator++(TComplex& z)
{
    ++z.Re;
    ++z.Im;
    printf("Префиксная форма ++ \n");
    return z;
 }

 const TComplex operator++(TComplex& z, int i)
{
    ++z.Re;
    ++z.Im;
    printf("Постфиксна форма ++ %d\n",i);
    return z;
 }

 TComplex& operator--(TComplex& z)
{
    --z.Re;
    --z.Im;
    printf("Префиксна форма -- \n");
    return z;
 }

 const TComplex operator--(TComplex& z, int i)
{
    --z.Re;
    --z.Im;
    printf("Постфиксна форма -- %d\n",i);
    return z;
 }
```

Як і колись, щоб розрізнити префіксну і постфіксну форми операторів інкрементації і декрементації, застосовується фіктивний целочисловий параметр.

Результат цієї програми легко пророчити.

```
 Префиксна форма ++
2.000000 + i*2.000000
Постфиксна форма ++ 0
3.000000 + i*3.000000
Префиксна форма --
2.000000 + i*2.000000
Постфиксна форма -- 0
1.000000 + i*1.000000
```

8.4. ПЕРЕВАНТАЖЕННЯ БІНАРНИХ ОПЕРАТОРІВ ЗА ДОПОМОГОЮ ДРУЖНІХ ФУНКЦІЙ

Якщо стандарт не накладає особливих обмежень, програміст може сам вибирати спосіб перевантаження. Однак у деяких ситуаціях функції-члени можуть виявитися незручними. Наприклад, у нашому класі `TComplex` визначений оператор +, у якого і лівий, і правий операнди є об'єктами класу `TComplex`.

У деяких випадках перевантаження арифметичних операторів за допомогою функцій-членів може привести до небажаних ефектів. Розглянемо наступну реалізацію.

Перевантаження оператора “плюс” за допомогою функції-члена

```
   #include <stdio.h>

class TComplex
{
    double Re;
    double Im;
public:
    TComplex(double x, double y=0):Re(x),Im(y){ }
    TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
    ~TComplex(){}
    void print();
    const TComplex operator+(const TComplex& z)
    {
       TComplex w(0,0);
       w.Re = Re+z.Re;
       w.Im = Im+z.Im;
       return w;
    }
};

int main()
{
    TComplex u(1,1),v(2,2),z(0,0);
    z = u + 1.0;
    z.print();
    return 0;
}

void TComplex::print()
{
    printf("%lf + i*%lf\n", Re, Im);
}
```

Оскільки другий параметр конструктора задається за замовчуванням, відкривається можливість неявного перетворення змінної типу `double` в об'єкт класу `TComplex`. Отже, вираження

```
z = u + 1.0;
```

цілком коректно. Одержуємо результат: `2.000000 + i*1.000000`. Однак особливості перевантаження операторів за допомогою функцій-членів порушують математичні правила: вираження `1.0+u` невірно. Це порозуміється тим, що лівим і правої операндами операторної функції `operator+()` повинні бути об'єкти класу `TComplex`. Якщо як правий об'єкт зазначена змінна типу `double`, вона буде неявно перетворена в об'єкт класу `TComplex`. Однак на об'єкт, що викликає функцію `operator+()`, це не поширюється. Компілятор просто повідомить, що для цієї ситуації в класі немає придатної функції-члена. Дозволити цю ситуацію просто — оператор потрібно перевантажити за допомогою дружньої функції.

Перевантаження оператора + за допомогою дружньої функції

```
   #include <stdio.h>

class TComplex
{
    double Re;
    double Im;
public:
    TComplex(double x, double y):Re(x),Im(y){ }
    TComplex(TComplex& z){ Re = z.Re; Im = z.Im;}
    ~TComplex(){}
    friend void print(TComplex z);
    friend TComplex operator+(TComplex x, TComplex y);
};

int main()
{
    TComplex u(1,1),v(2,2),z(0,0);
    z=u+v;
    print(z);
    return 0;
}
```

```
TComplex operator+(TComplex x, TComplex y)
{
   TComplex w(0,0);
   w.Re = x.Re+y.Re;
   w.Im = x.Im+y.Im;
   return w;
}

void print(TComplex z)
{
   printf("%lf + i*%lf \n",z.Re,z.Im);
}
```

Тепер обидва параметри дружньої функції можуть неявно перетворюватися з типу `double` у тип `TComplex`, і комутативність додавання відновлена.

8.5. ПЕРЕТВОРЕННЯ ТИПІВ

Ми уже зіштовхувалися з перетворенням типів, коли намагалися складати комплексні і дійсні числа. Нагадаємо, що основну роль у неявному перетворенні зіграв конструктор, у якого один з аргументів був заданий за замовчуванням.

Синтаксична конструкція, що реалізує явне перетворення об'єкта одного класу в об'єкт іншого класу, має наступний вид.

`operator` *`результуючий_тип`*`(void);`

Таким чином, щоб привести об'єкт `Obj` класу `TClass1` до типу `TClass2`, необхідно виконати оператор `(TClass2)obj`.

Перетворення типів

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

long base = 20;

class TLong
{
   char* pLong;
public:
   TLong(long);
   operator int();
   void print();
};

TLong::operator int()
{
   int i=0;
   for (int j = base-1; j >= 0; j--) i+=pLong[j]*pow(10,j);
   return i;
}

void TLong::print()
{
   for (int j = base-1; j >= 0; j--) printf("%d ",pLong[j]);
}

TLong::TLong(long x)
{
   pLong= new char[base];
   register int i=0;
   while(i<base)pLong[i++]=0;
   i=0;
   while(x)pLong[i++]=x%10,x/=10;
   (*this).print();
}

int main()
{
   TLong x(2498);
   int i;
   i=(int)x;
```

```
    printf("\ni = %d",i);
}
```

Ця програма спочатку створює об'єкт, що представляє собою масив, що складається 20 десяткових цифр позитивного цілого числа. Потім до цього об'єкта застосовується операторна функція `operator int()`, що перетворить його в число типу `int`.

8.8. РЕЗЮМЕ

- Не всі символи операцій можна перевантажити. Нижче перераховані оператори, дозволені до перевантаження.

```
+       -       *       /       %       ^       &
|       ~       !       =       <       >       +=
-=      *=      /=      %=      ^=      &=      |=
<<      >>      >>=     <<=     ==      !=      <=
>=      &&      ||      ++      --      ->*     ,
->      []      ()      new     new[]   delete  delete[]
```

- До операторів, які не можна перевантажити, належать `::` (оператор дозволу області видимості), `.` ("точка" — оператор доступу до члена класу), `?:` (тернарний оператор), `.*` (доступ до розіменованого вказівника-члена класу), `sizeof`, `typeid`, `static_cast`, `dynamic_cast`, `const_cast` і `reinterpret_cast`. Крім того, не рекомендується перевантажувати логічні оператори `&&` і `||`, оскільки на їхні перевантажені версії не поширюється правило скорочених обчислень логічних виразів. (Нагадаємо, що це правило полягає в наступному: якщо на деякому етапі значення усього вираження стає визначеним, подальші обчислення припиняються.)
- Синтаксис операторних функцій виглядає в такий спосіб.

```
тип_значення_що_повертається operator символ_операції(параметри)
{
   ...
}
```

Наприклад, операторна функція, що перевантажує операцію `+`, називається `operator+()`.

- Операторні функції повинні мати прямий доступ до членів класу. Отже, необхідно, щоб вони були або членами класу, або дружніми функціями.
- Необхідно пам'ятати про обмеження, що супроводжують застосування перевантажених операторів: 1) перевантажені функції не можуть змінити пріоритет операторів, 2) кількість операндів фіксована: жодного, один чи два, 3) значення операндів не можна задавати за замовчуванням.
- Оператори можуть бути унарними і бінарними. Унарний оператор має один операнд, а бінарний — два. Нагадаємо, що до унарних операторів, що перевантажуються, належать такі оператори, як `+`, `-`, `++`, `--`, `&`, `~` і `!`. До перевантажуваного бінарного належать всі інші оператори, перераховані в приведеній вище таблиці.
- Операторні функції-члени, що перевантажують унарний оператор, мають одну особливість: їх операнди передаються неявно за допомогою вказівника `this`. Отже, така функція-член класу не має явних параметрів.
- У мові C++ передбачено дві форми операторів інкремента і декремента: префіксна і постфіксна. Для того щоб розрізняти їх, використовується звичайний механізм перевантаження функцій — вводиться фіктивний цілочисловий параметр. Якщо символ операції `++` стоїть перед операндом, викликається операторна функція `operator++()`, якщо після — операторна функція `operator++(int i)`. Змінна і відіграє роль прапора, що повідомляє компілятору, що дана функція перевантажує постфіксну форму оператора інкремента і декремента.
- Незважаючи на те що програміст вільний вільно трактувати перевантажений оператор, прагнучи зберегти аналогію з його убудованими аналогами, необхідно дотримувати визначені правила. Наприклад, як відомо, оператор інкрементації цілих чисел повертає посилання на неконстантний об'єкт. Ця властивість повинна зберігатися і при перевантаженні.
- Постфіксний оператор повертає константне значення. Саме тому в C++ неможливі вираження `x++++`. Цілком природно зажадати, щоб перевантажений оператор мав таку ж властивість.
- Крім того, оскільки у вихідному варіанті постфіксний оператор іінкрементації реалізується через префіксний, це також бажано врахувати при перевантаженні. У цьому випадку реалізація операторної функції `++` для постфіксної форми може виглядати так.
- Оператори заперечення (`!`), узяття адреси (`&`) і побітового заперечення (`~`) допускають перевантаження, але не мають універсальних альтернатив, що варто було б реалізувати. Їх можна перевантажувати, наприклад, для підвищення наочності програми. Скажемо, за допомогою оператора `!` можна позначати операцію звертання матриці, а за допомогою символу `~` — її транспонування. Щоправда, застосування тильди закріплене за деструкторами, тому варто виявляти обережність, щоб не створити плутанину. У будь-якому випадку зміст перевантаження операторів залежить від конкретної задачі.
- Оператор *посилання на член об'єкта* є унарним. Операторна функція, що перевантажуєйого, виглядає в такий спосіб.

```
об'єкт -> елемент
```

Цей запис еквівалентний наступному вираженню.

```
об'єкт.operator->(елемент);
```

- Функція `operator- >()` повинна бути нестатичним членом класу. Як параметр вона одержує об'єкт чи класу посилання на нього, повертаючи вказівник `this` на об'єкт, звідки надійшов виклик, або посилання на об'єкт будь-якого іншого класу, у якому визначений оператор `->`. її зручно використовувати в контейнерних класах, що містять усередині себе вказівник на інший клас. Основний зміст перевантаження оператора `->` полягає в додатковій функціональності, що розширює можливості звичайних вказівників.
- Бінарний оператор має два операнди. Його виклик виконується об'єктом, розташованим у лівій частині оператора. Отже, бінарний оператор

```
a+b
```

еквівалентний такому оператору.

```
a.operator+(b)
```

- Таким чином, бінарна операторна функція-член повинна має тільки один параметр, що задає другий операнд. Вказівник `this` на перший операнд вона одержує неявно.
- Для того щоб бінарну операторну функцію можна було застосовувати усередині виразів, необхідно, щоб вона повертала об'єкт свого класу.
- У перевантаженому операторі присвоювання повинне виконуватися не побітове, а глибоке копіювання. Адреса об'єкта `y` залишається незмінним, міняється лише зміст.
- У функції `operator=()` повинна бути передбачена перевірка самоприсвоювання. Це дозволяє коректно обробляти безглузді вираження такого виду.

```
x = x;
```

- Незважаючи на зовнішню схожість перевантаженого оператора присвоювання і конструктора копіювання, між ними існує принципова різниця. При виклику конструктора копіювання створюється новий об'єкт, що іниціалізується раніше існуючим об'єктом. При присвоюванні обидва об'єкти уже існують.
- Як випливає з визначення стандартної операції послідовного обчислення, її значенням є результат обчислення останнього вираження серед перерахованих у списку. Отже, головним фактором, що впливає на результат застосування цього оператора, є порядок перерахування його операндів. Однак коли оператор "`,`" перевантажується за допомогою операторної функції, операнди стають параметрами *функції*. Це створює проблему, тому що порядок обчислення аргументів функції стандартом мови не визначений. Таким чином, перевантаження оператора послідовного обчислення може привести до хитливої роботи програми і непередбачених ефектів.
- Функція `operator,()` повинна бути нестатичним членом класу.
- Програміст може керувати виділенням пам'яті, перевантажуючи оператори `new` і `delete`. Перевантажена операторна функція має наступний вид.

```
void* operator new(size_t size);
```

- Вона виділяє `size` байт пам'яті і повертає адресу виділеної пам'яті. Конструктор і деструктор об'єктів викликаються автоматично. Тип `size_t` є цілочисловим.
- Перевантажений оператор `delete` звільняє пам'ять, виділену перевантаженим оператором `new`.
- Оператор `new` має особливу форму перевантаження, що називається синтаксисом розміщення. Вона дозволяє створювати об'єкт, розміщаючи його в осередку з заданою адресою. Нагадаємо, що саме в цьому випадку необхідно явно викликати деструктор.
- Бинарний оператор доступу до члена масиву перевантажується за допомогою наступної операторної функції, що повинна бути нестатичним членом класу.
-
```
тип_значення_що_повертається_значення&
ім'я_класу::operator[](інтегральний_тип i)
 {
   // ...
}
```

- Відзначимо два моменти. По-перше, оскільки операція доступу застосовується до індексованих масивів, параметр операторної функції повинний бути цілочисловим. По-друге, елементи масиву можуть стояти як у лівої, так і в правій частині оператора присвоювання. Отже, функція `operator[]()` повинна повертати посилання або вказівник.
- Хоча зовні функція `operator[]()` виглядає унарною, насправді вона є бінарною. Її перший параметр явно задає індекс елемента, а другий параметр, що представляє собою вказівник `this` на об'єкт, що здійснює виклик, передається неявно.
- Об'єкти, що містять операторну функцію `operator()()`, називаються *функціями-об'єктами*, чи функторами. Ця функція може одержувати довільну кількість параметрів і повертати значення будь-яких типів. Такі об'єкти бувають корисними при виконанні операцій, зв'язаних з декількома індексами. Як відзначалася вище, операторна функція `operator()()` повинна бути нестатичним членом класу.

- У деяких ситуаціях перевантажені оператори повинні бути членами класу. До них належать оператори `=`, `[]`, `->`, `new` і `delete`. В інших випадках програміст повинний керуватися здоровим глуздом.
- Перевантаження операторів у виді дружніх функцій нічим не відрізняється від перевантаження у виді членів класу, за одним виключенням — функція тепер не одержує неявний вказівник `*this`. Отже, всі операнди повинні бути зазначені явно.
- Синтаксична конструкція, що реалізує явне перетворення об'єкта одного класу в об'єкт іншого класу, має наступний вид.

  `operator` *`результуючий_тип`*`(void);`

  Таким чином, щоб привести об'єкт `Obj` класу `TClass1` до типу `TClass2`, необхідно виконати оператор `(TClass2)obj`.

8.8. КОНТРОЛЬНІ ПИТАННЯ

1. Які операції можна перевантажити?
2. Які операції не можна перевантажити?
3. Як виглядає синтаксис операторних функцій?
4. Які вимоги висуваються до операторних функцій?
5. Які обмеження накладаються на застосування перевантажених операторів?
6. Назвіть особливість операторних функцій-членів, що перевантажують унарний оператор.
7. Назвіть дві форми перевантаження операторів інкремента і декремента.
8. Назвіть особливості перевантаження операторів інкремента і декремента.
9. Як перевантажується оператор посилання на член об'єкта?
10. Які обмеження накладаються на функцію `operator->()`?
11. Назвіть особливості бінарних операторних функцій-членів.
12. Назвіть вимоги до перевантаженого оператору присвоювання.
13. Назвіть вимоги до перевантаженого операторної функції `operator,()`.
14. Як виглядає перевантажена операторна функція для оператору `new`.
15. Що таке синтаксис розміщення?
16. Як перевантажується бінарний оператор доступу до члена масиву.
17. Що таке функція-об'єкт, або функтом?
18. Які операторні функції обов’язково повинні бути членами класу?
19. Чим перевантаження операторів у вигляді дружніх функцій відрізняється від перевантаження у вигляді членів класу?
20. Опишіть синтаксичну конструкцію, що реалізує явне перетворення об'єкта одного класу в об'єкт іншого класу.