№ 6 Перегрузка операций

Создать заданный в варианте класс. Определить в классе необходимые функции и заданные перегруженные операции. Написать программу тестирования, в которой проверяется использование перегруженных операций.

Вариант 1	Класс – **Одномерный массив**. Дополнительно перегрузить следующие операции: * – умножение массивов; [] – доступ по индексу, int() – размер массива; = = – проверка на равенство; <= – сравнение.
Вариант 2	Класс – **Одномерный массив**. Дополнительно перегрузить следующие операции: [] – доступ по индексу; > – проверка на вхождение; != – проверка на неравенство; + – объединение массивов
Вариант 3	Класс – множество **Set**. Дополнительно перегрузить следующие операции: + – добавить элемент в множество (типа set+item); + – объединение множеств; * – пересечение множеств; int()– мощность множества.
Вариант 4	Класс – множество **Set**. Дополнительно перегрузить следующие операции: - – удалить элемент из множества (типа set-item); * – пересечение множеств; < – сравнение множеств; > – проверка на подмножество; int()– мощность множества.
Вариант 5	Класс – множество **Set**. Дополнительно перегрузить следующие операции: - – удалить элемент из множества (типа set-item); > – проверка на подмножество; != – проверка множеств на неравенство; + – добавить элемент в множество (типа set+item); * – пересечение множеств.
Вариант 6	Класс – однонаправленный список **List**. Дополнительно перегрузить следующие операции: () – удалить элемент в заданной позиции, например: int i; list L; L(i); () – добавить элемент в заданную позицию, например: int i; Type c; list L; L(c,i); != – проверка на неравенство.

Вариант 7	Класс – множество **Set**. Дополнительно перегрузить следующие операции: () – конструктор множества (в стиле конструктора для множественного типа); + – объединение множеств; <= – сравнение множеств; int()– мощность множества; [] - доступ к элементу в заданной позиции.
Вариант 8	Класс – множество **Set**. Дополнительно перегрузить следующие операции: > – проверка на принадлежность (типа операции in множественного типа* – пересечение множеств; < – проверка на подмножество; int()– мощность множества; [] - доступ к элементу в заданной позиции.
Вариант 9	Класс – однонаправленный список **List**. Дополнительно перегрузить следующие операции: + – объединить два списка; -- – удалить элемент из начала (--list); = = – проверка на равенство; bool() – проверка, пустой ли список.
Вариант 10	Класс – двунаправленый список **List**. Дополнительно перегрузить следующие операции: + – добавить элемент в начало (item+list); -- – удалить элемент из начала (--list); != – проверка на неравенство; * - объединение двух списков.
Вариант 11	Класс – однонаправленный список **List**. Дополнительно перегрузить следующие операции: + – добавить элемент в конец (list+item); -- – удалить элемент из конца (типа list--); != – проверка на неравенство; [] - доступ к элементу в заданной позиции.
Вариант 12	Класс - однонаправленный список **List**. Дополнительно перегрузить следующие операции: [] - доступ к элементу в заданной позиции; + - объединить два списка; = = - проверка на равенство; < - добавление одного списка к другому.
Вариант 13	Класс - стек **Stack**. Дополнительно перегрузить следующие операции: + - добавить элемент в стек; -- - извлечь элемент из стека; bool() - проверка, пустой ли стек; > - копирование одного стека в другой с сортировкой в возрастающем порядке.
Вариант 14	Класс - очередь **Queue**. Дополнительно перегрузить следующие операции: + - добавить элемент; -- - извлечь элемент; bool() - проверка, пустая ли очередь; < - копирование одной очереди в другую с сортировкой в убывающем порядке; int()– мощность.
Вариант 15	Класс - **Вектор**. Дополнительно перегрузить следующие операции: + - сложение векторов; ()- доступ по индексу V(i); > - сравнение векторов; = = - копирование вектора.

Вариант 16	Класс - **Марица**. Дополнительно перегрузить следующие операции: + - сложение матриц; ()- доступ по индексу V(i) к заданной строке; > - сравнение матриц по модулю; = = - копирование матрицы.
Вариант 17	Класс - **Марица**. Дополнительно перегрузить следующие операции: - - вычистания числа из всех элемнтов матрицы; ++ инкремнт всех элементов матрицы; != - сравнение матриц по модулю; int() – количество нулевых элемнтов в матрице.
Вариант 18	Класс - **Марица**. Дополнительно перегрузить следующие операции: + - сложение матриц; -- обнуление всех элементов матрицы; == - сравнение матриц по нулевому столбцу; int() – количество отрицательных элемнтов в матрице.
Вариант 19	Класс - **Марица**. Дополнительно перегрузить следующие операции: < - сравнения матриц; >= приведение матрицы к единичному виду; == - сравнение матриц по первому элементу; * – инверсия всех элемнтов матрицы.
Вариант 20	Класс - стек **Stack**. Дополнительно перегрузить следующие операции: * - добавить элемент в стек; /- извлечь элемент из стека; bool() - проверка, есть ли в стеке отрицатеьные элменты; == - сравнения стеков.
Вариант 21	Класс - стек **Stack**. Дополнительно перегрузить следующие операции: - - извлечение всех элементов равных заданному; ++ - дублирование верхнего элемента; <= коприрование неповторяющихся элементов из второго стека.
Вариант 22	Класс - очередь **Queue**. Дополнительно перегрузить следующие операции: / - добавить элемент; ++ - извлечь элемент; bool() - проверка, на содержание четных элемнтов в очереди; int()– количество положительных эелементов в очереди
Вариант 23	Класс - **Строка**. Дополнительно перегрузить следующие операции: < - сравнения строк в лексиграфическои порядке; + добавления числа к строке; - удаление последнего символа в строке; * – замена всех символов в строке на заданный.
Вариант 24	Класс - **Строка**. Дополнительно перегрузить следующие операции: < - удаление всех символов равных заданному; + удаление нечентых символов; != сравнение длин строк; [] – доступ к символу строки по индексу.
Вариант 25	Класс – **Строка**. Дополнительно перегрузить следующие операции: - – удалить элемент из строки из заданной позиции (типа set-item); > – проверка на вхождение подстроки; != – проверка строк на неравенство; + – добавить элемент в строку

	на заданную позицию (типа string+item).
Вариант 26	Класс – **Пароль**. Дополнительно перегрузить следующие операции: - – замена последнего символа (типа password-item); > – сравнение длин паролей; != – проверка паролей на неравенство; ++ – сброс пароля на значение по умолчанию; bool() - проверка на стройкость.

Пример

```
#include<iostream>
class Complex
{
  public:
  double re, im;
  Complex(double r = 0.0, double i = 0.0) { re = r; im = i; }

  friend Complex operator + (const Complex& x, const Complex& y)
      { return Complex(x.re + y.re, x.im + y.im); }

  friend Complex operator - (const Complex& x, const Complex& y)
      { return Complex(x.re - y.re, x.im - y.im); }

  friend Complex operator * (const Complex& x, const Complex& y)
      { return Complex(x.re * y.re - x.im * y.im, x.re * y.im +
y.re * x.im); }

  friend Complex operator / (const Complex& x, const Complex& y)
      {
        double z = y.re * y.re + y.im * y.im;
        return Complex((x.re * y.re + x.im * y.im) / z,
                       (y.re * x.im - x.re * y.im) / z);
      }

  friend ostream& operator << (ostream& output, const Complex&
x)
      { return output<<x.re<<"+i"<<x.im; }

  friend istream& operator >> (istream& input, Complex& x)
      { return input>>x.re>>x.im; }

  friend int operator == (const Complex& x, const Complex& y)
      { return (x.re == y.re) && (x.im == y.im); }

  friend int operator != (const Complex& x, const Complex& y)
      { return (x.re != y.re) || (x.im != y.im); }
};

void main()
{
  Complex a, b;
  cin>>a>>b;
```

```
  cout<<a<<endl<<b<<endl<<a+b<<endl<<a-
b<<endl<<a*b<<endl<<a/b<<endl;
  if (a == b) cout<<"equals "<<endl;
  if (a != b) cout<<"not equals"<<endl;

}
```

Теория

В языке C++ определены множества операций над переменными стандартных типов, такие как +, -, *, / и т.д. Каждую операцию можно применить к операндам определенного типа.

К сожалению, лишь ограниченное число типов непосредственно поддерживается любым языком программирования. Например, C и C++ не позволяют выполнять операции с комплексными числами, матрицами, строками, множествами. Однако, все эти операции можно выполнить через классы в языке C++.

Рассмотрим пример.

Пусть заданы множества A и B:

A = { a1, a2, a3 };

B= { a3, a4, a5 },

и мы хотим выполнить операции объединения (+) и пересечения (*) множеств.

A + B = { a1, a2, a3, a4, a5 }

A * B = { a3 }.

Можно определить класс Set - "множество" и определить операции над объектами этого класса, выразив их с помощью знаков операций, которые уже есть в языке C++, например, + и *. В результате операции + и * можно будет использовать как и раньше, а также снабдить их дополнительными функциями (объединения и пересечения). Как определить, какую функцию должен выполнять оператор: старую или новую? Очень просто – по типу операндов. А как быть с приоритетом операций? Сохраняется определенный ранее приоритет операций. Для распространения действия операции на новые типы данных надо определить специальную функцию, называемую "операция-функция" (operator-function). Ее формат:

тип_возвр_значения operator знак_операции(специф_параметров)
{операторы_тела_функции}

При необходимости может добавляться и прототип:

тип_возвр_значения operator знак_операции(специф_параметров);

Если принять, что конструкция operator знак_операции есть имя некоторой функции, то прототип и определение операции-функции подобны прототипу и определению обычной функции языка C++. Определенная таким образом операция называется перегруженной (overload).

Чтобы была обеспечена явная связь с классом, операция-функция должна быть либо компонентом класса, либо она должна быть определена в классе как дружественная и у нее должен быть хотя бы один параметр типа класс (или ссылка на класс). Вызов операции-функции осуществляется так же, как и любой другой функции C++: operator @. Однако разрешается использовать сокращенную форму ее вызова: a @ b, где @ - знак операции.

Перегрузка унарных операций.

Любая унарная операция @ может быть определена двумя способами: либо как компонентная функция без параметров, либо как глобальная (возможно дружественная) функция с одним параметром. В первом случае выражение @ Z означает вызов Z.operator @(), во втором - вызов operator @(Z).

Унарные операции, перегружаемые в рамках определенного класса, могут перегружаться только через нестатическую компонентную функцию без параметров. Вызываемый объект класса автоматически воспринимается как операнд.

Унарные операции, перегружаемые вне области класса (как глобальные функции), должны иметь один параметр типа класса. Передаваемый через этот параметр объект воспринимается как операнд.

Синтаксис:

а) в первом случае (описание в области класса):

тип_возвр_значения operator знак_операции

б) во втором случае (описание вне области класса):

тип_возвр_значения operator знак_операции (идентификатор_типа)

Пример.

```
class Person {
public:
  void operator ++()
  {    ++age;    }
protected:
  int age;
  ...
};
```

```
int main(void)
{  Person John;
  ++John;
  ...
}    class Person {
protected:
  int age;
  ...
friend void operator ++(Person &);
};
void operator ++(Person &ob)
{  ++ob.age;}
int main (void)
{  Person John;
  ++John;
  ...
}
```

Перегрузка бинарных операций.

Любая бинарная операция @ может быть определена двумя способами: либо как компонентная функция с одним параметром, либо как глобальная (возможно дружественная) функция с двумя параметрами. В первом случае x @ y означает вызов x.operator @(y), во втором – вызов operator @(x, y).

Операции, перегружаемые внутри класса, могут перегружаться только нестатическими компонентными функциями с параметрами. Вызываемый объект класса автоматически воспринимается в качестве первого операнда. Операции, перегружаемые вне области класса, должны иметь два операнда, один из которых должен иметь тип класса.

Пример.

```
class Person { ... };
class AdressBook
{
public:
  Person &operator [](int);// доступ к i
protected:
  // содержит в качестве компонентных данных множество объектов типа
  // Person представляемых как динамический массив, список или дерево
  ...
};
Person &AdressBook::operator [](int i)
{ ... }
int main(void)
{
  AdressBook persons;
  Person record;
  ...
  record = persons[3];
  ...
}
```

```
class Person { ... };
class AdressBook
{
protected:
  // содержит в качестве компонентных данных множество объектов типа
  // Person представляемых как динамический массив, список или дерево
  ...
friend Person &operator [](const AdressBook &, int); // доступ к i
};
Person &AdressBook::operator [](const AdressBook &, int i)
{ ... }


int main(void)
{  AdressBook persons;
  Person record;
  ...
  record = persons[3];
  ...
}
```

Перегрузка операций ++ и --.

Унарные операции инкремента ++ и декремента -- существуют в двух формах: префиксной и постфиксной. В современной спецификации C++ определен способ, по которому компилятор может различить эти две формы. В соответствии с этим способом задаются две версии функции operator ++() и operator --(). Они определены следующим образом:

Префиксная форма:

operator ++();
operator --();

Постфиксная форма:

operator ++(int);
operator --(int);

Указание параметра int для постфиксной формы не специфицирует второй операнд, а используется только для отличия от префиксной формы.

Пример.

```
class Person
{public:
...
```

```
void operator ++()
{++age; }
void operator ++(int)
{age++; }
protected:
int age;
...
};
int main(void)
{Person John;
John++;
++John;
}
```

Перегрузка операции вызова функции.

Это операция '()'. Она является бинарной операцией. Первым операндом обычно является объект класса, вторым – список параметров. Пример.

```
class Matrix // двумерный массив вещественных чисел
{
public:
...
double operator ()(int, int); //доступ к элементам матрицы по индексам
};
double Matrix::operator() (int i, int j)
{ ... }
int main (void)
{
Matrix a;
double k;
...
k = a(5, 6);
...
}
```

Перегрузка операции присваивания.

Операция отличается тремя особенностями:

- ✓ операция не наследуется;
- ✓ операция определена по умолчанию для каждого класса в качестве операции поразрядного копирования объекта, стоящего справа от знака операции, в объект, стоящий слева.
- ✓ операция может перегружаться только в области определения класса. Это гарантирует, что первым операндом всегда будет леводопустимое выражение.

Если вас устраивает поразрядное копирование, нет смысла создавать собственную функцию operator =(). Однако бывают случаи, когда поразрядное копирование нежелательно. Например, использование предопределенной операции присваивания для классов, содержащих указатели в качестве компонентных данных, чаще всего приводит к ошибкам. Покажем это на примере.

Пользовательский класс - строка String:

```
class String
{public:
  String(char *);
  ~String();
  void show();
protected:
  char *p; // указатель на строку
  int len; // текущая длина строки
};
String::String(char *ptr)
{
  len = strlen(ptr);
  p = new chat[len + 1];
  if (!p)
  {    cout << "Ошибка выделения памяти\n");
    exit(1);
  }
  strcpy(p, ptr);
}
String::~String(){  delete [] p;}
void String::show(void){  cout << *p& lt;< "\n";}
int main(void)
{
  String s1("Это первая строка"),
         s2("А это вторая строка");
  s1.show();
  s2.show()
  s2 = s1; // Это ошибка
  s1.show();
  s2.show();
  return 0;
}
```

В чем здесь ошибка? Когда объект s1 присваивается объекту s2, указатель p объекта s2 начинает указывать на ту же самую область памяти, что и указатель p объекта s1. Таким образом, когда эти объекты удаляются, память, на которую указывает указатель p объекта s1, освобождается дважды, а память, на которую до присваивания указывал указатель p объекта s2, не освобождается вообще.

Хотя в данном примере эта ошибка и не опасна, в реальных программах с динамическим распределением памяти она может вызвать крах программы.

В этом случае необходимо самим перегружать операцию присваивания. Покажем как это сделать для нашего класса String.

```
class String
{public:
  ...
  String &operator =(String &);

protected:
  char *p; // указатель на строку
  int len; // текущая длина строки
};
String &String::operator =(String &ob);
{  if (this == &ob) return *this;
  if (len < ob.len)
  {    // требуется выделить дополнительную память
    delete [] p;
    p = new char[ob.len + 1];
    if (!p)
    {
      cout << "Ошибка выделения памяти\n");
      exit(1);
    }
  }
  len = ob.len;
  strcpy(p, ob.p);
  return *this;
}
```

В этом примере выясняется, не происходит ли самоприсваивание (типа ob = ob). Если имеет место самоприсваивание, то просто возвращается ссылка на объект.

Затем проверяется, достаточно ли памяти в объекте, стоящем слева от знака присваивания, для объекта, стоящего справа от знака присваивания. Если не достаточно, то память освобождается и выделяется новая, требуемого размера. Затем строка копируется в эту память.

Отметим две важные особенности функции operanor =(). Во-первых, в ней используется параметр-ссылка. Это необходимо для предотвращения создания копии объекта, передаваемого через параметр по значению. В случае создания копии, она удаляется вызовом деструктора при завершении работы функции. Но деструктор освобождает память, на которую указывает p. Однако эта память все еще необходима объекту, который является аргументом. Параметр-ссылка помогает решить эту проблему.

Во-вторых, функция operator =() возвращает не объект, а ссылку на него. Смысл этого тот же, что и при использовании параметра-ссылки. Функция возвращает временный объект, который удаляется после завершения ее работы. Это означает, что для временной переменной будет вызван

деструктор, который освобождает память по адресу p. Но она необходима для присваивания значения объекту. Поэтому, чтобы избежать создания временного объекта, в качестве возвращаемого значения используется ссылка.

Другой путь решения проблем, описанных выше - это создание конструктора копирования. Но конструктор копирования может оказаться не столь эффективным решением, как ссылка в качестве параметра и ссылка в качестве возвращаемого значения функции. Это происходит потому, что использование ссылки исключает затраты ресурсов, связанных с копированием объектов в каждом из двух указанных случаев.

Перегрузка операции new.

Синтаксис

Операция new, заданная по умолчанию, может быть в двух формах:
1) new тип <инициализирующее выражение>
2) new тип [];

Первая форма используется не для массивов, вторая - для массивов. Перегруженную операцию new можно определить в следующих формах, соответственно для не массивов и для массивов:

void *operator new (size_t t[,остальные аргументы]);
void *operator new [] (size_t t[,остальные аргументы]);

Первый и единственный обязательный аргумент t всегда должен иметь тип size_t. Если аргумент имеет тип size_t, то в операцию-функцию new автоматически подставляется аргумент sizeof(t), т.е. она получает значение, равное размеру объекта t в байтах.
Например, пусть задана следующая функция:

```
void *operator new(size_t t, int n)
{ return new char [t * n]; }
```

и она вызывается следующим образом:

```
double *d = new(5) double;
Здесь t = double, n = 5.
```

В результате после вызова значение t в теле функции будет равно sizeof(double).

При перегрузке операции new появляется несколько глобальных операций new, одна из которых определена в самом языке по умолчанию, а

другие являются перегруженными. Возникает вопрос: как различить такие операции? Это делается путем изменения числа и типов их аргументов. При изменении только типов аргументов может возникнуть неоднозначность, являющаяся следствием возможных преобразований этих типов друг к другу. При возникновении такой неоднозначности следует при обращении к new задать тип явно, например:

```
new((double) 5) double;
```

Одна из причин, по которой перегружается операция new, состоит в стремлении придать ей дополнительную семантику, например, обеспечение диагностической информацией или устойчивости к сбоям. Кроме того класс может обеспечить более эффективную схему распределения памяти чем та, которую предоставляет система.

В соответствии со стандартом C++ в заголовочном файле <new> определены следующие функции-операции new, позволяющие передавать, наряду с обязательным первым size_t аргументом и другие.

```
void *operatop new(size_t t) throw (bad_alloc);
void *operatop new(size_t t, void *p) throw ();
void *operatop new(size_t t, const nothrow &) throw ();
void *operatop new(size_t t, allocator &a);
void *operatop new [](size_t t) throw (bad_alloc);
void *operatop new [](size_t t, void *p) throw ();
void *operatop new [](size_t t, const nothrow &) throw ();
```

Эти функции используют генерацию исключений (throw) и собственный распределитель памяти (allocator).

Версия с nothrow выделяет память как обычно, но если выделение заканчивается неудачей, возвращается 0, а не генерируется bad_alloc. Это позволяет нам для выделения памяти использовать стратегию обработки ошибок до генерации исключения.

Правила использования операции new

- ✓ Объекты, организованные с помощью new имеют неограниченное время жизни. Поэтому область памяти должна освобождаться оператором delete.
- ✓ Если резервируется память для массива, то операция new возвращает указатель на первый элемент массива.
- ✓ При резервировании памяти для массива все размерности должны быть выражены положительными величинами.
- ✓ Массивы нельзя инициализировать.
- ✓ Объекты классов могут организовываться с помощью операции new, если класс имеет конструктор по умолчанию.

- ✓ Ссылки не могут организовываться с помощью операции new, так как для них не выделяется память.
- ✓ Операция new самостоятельно вычисляет потребность в памяти для организуемого типа данных, поэтому первый параметр операции всегда имеет тип size_t.

Обработка ошибок операции new.

Обработка ошибок операции new происходит в два этапа:

Устанавливается, какие предусмотрены функции для обработки ошибок. Собственные функции должны иметь тип new_handler и создаются с помощью функции set_new_handler. В файле new.h объявлены

```
typedef void (*new_handler)();
new_handler set_new_handler(new_handler new_p);
```

Вызывается соответствующая new_handler функция. Эта функция должна:

- ✓ либо вызвать bad_alloc исключение;
- ✓ либо закончить программу;
- ✓ либо освободить память и попытаться распределить ее заново.

Диагностический класс bad_alloc объявлен в new.h.

В реализации C++ включена специальная глобальная переменная _new_handler, значением которой является указатель на new_handler функцию, которая выполняется при неудачном завершении new. По умолчанию, если операция new не может выделить требуемое количество памяти, формируется исключение bad_alloc. Изначально это исключение называлось xalloc и определялось в файле except.h. Исключение xalloc продолжает использоваться во многих компиляторах. Тем не менее, оно вытесняется определенным в стандарте C++ именем bad_alloc.

Рассмотрим несколько примеров.

Пример 1. В примере использование блока try ... catch дает возможность проконтролировать неудачную попытку выделения памяти.

```
#include <iostream>
#include <new>
int main(void)
{
  double *p;
  try {
    p = new double[1000];
    cout << "Память выделилась успешно" << endl;
  }
  catch (bad_alloc xa) {
    cout << "Ошибка выделения памяти\n";
```

```
    cout << xa.what();
    return 1;
  }
  return 0;
}
```

Пример 2. Поскольку в предыдущем примере при работе в нормальных условиях ошибка выделения памяти маловероятна, в этом примере ошибка выделения памяти достигается принудительно. Процесс выделения памяти длится до тех пор, пока не произойдет ошибка.

```
#include <iostream>
#include <new>
int main(void)
{
  double *p;
  do
  {    try {
       p = new double[1000];
       cout << "Память выделилась успешно" << endl;
    }
    catch (bad_alloc xa) {
      cout << "Ошибка выделения памяти\n";
      cout << xa.what();
    }
  }
  while (p);
  return 0;
}
```

Пример 3. Демонстрируется перегруженная форма операции new - операция new (nothow).

```
#include <iostream>
#include <new>
int main(void)
{  double *p;
  struct nothrow noth_ob;
  do  {
    p = new(noth_ob) double[1000];
    if (!p) cout << "Ошибка выделения памяти\n";
    else cout << "Память выделилась успешно\n";
  }
  while (p);
  return 0;
}
```

Пример 4. Демонстрируются различные формы перегрузки операции new.

```
#include <iostream.h>
#include <new.h>
double *p, *q, **pp;
```

```
class Demo
{public:
  Demo()  {    value = 0;   }
  Demo(int i)  {    value = i;   }
  void *operator new(size_t, int, int);
  void *operator new(size_t, int);
  void *operator new(size_t, char *);
protected:
  int value;
};
void *Demo::operator new(size_t t, int i, int j)
{  if (j) return new(i) Demo;
  return NULL;
}
void *Demo::operator new(size_t t, int i)
{
  Demo *p = ::new Demo;
  (*p).value = i;
  return p;
}
void *Demo::operator new(size_t t, char *z)
{
  return ::new (z) Demo;
}
int main(void)
{
  Demo *p_ob1, *p_ob2;
  struct nothrow noth_ob;
  p = new double;
  pp = new double *;
  p = new double(1.2); // инициализация
  q = new double[3];    // массив
  p_ob1 = new Demo[10]; // массив объектов demo
  void (**f_ptr)(int) // указатель на указатель на функцию
  f_ptr = new(void(*[3])(int)) // массив указателей на функцию
  char z[sizeof(Demo)]; // резервируется память в соответствии с
величиной demo
  p_ob2 = new(z) Demo; // организуется demo-объект в области памяти на
                       // которую указывает переменная z
  p_ob2 = new(3) Demo; // demo-объект с инициализацией
  p_ob1 = new(3, 0) Demo; // возвращает указатель NULL
  p_ob2 = new(noth_ob) Demo[5]; // массив demo-объектов,
                                // в случае ошибки возвращает NULL
  return 0;
}
```

Перегрузка операции delete.

- Операция-функция delete бывает двух видов:

void operator delete(void *);
void operator delete(void *, size_t);

Вторая форма включает аргумент типа size_t, передаваемый delete. Он передается компилятору как размер объекта, на который указывает p.

Особенностью перегрузки операции delete является то, что глобальные операции delete не могут быть перегружены. Их можно перегрузить только по отношению к классу.

Основные правила перегрузки операций.

- ✓ Вводить собственные обозначения для операций, не совпадающие со стандартными операциями языка C++, нельзя.
- ✓ Не все операции языка C++ могут быть перегружены. Нельзя перегрузить следующие операции:

. – прямой выбор компонента,
.* – обращение к компоненту через указатель на него,
? : – условная операция,
:: – операция указания области видимости,
sizeof, # и ## – препроцессорные операции.

- ✓ Каждая операция, заданная в языке, имеет определенное число операндов, свой приоритет и ассоциативность. Все эти правила, установленные для операций в языке, сохраняются и для ее перегрузки, т.е. изменить их нельзя.
- ✓ Любая унарная операция @ определяется двумя способами: либо как компонентная функция без параметров, либо как глобальная (возможно дружественная) функция с одним параметром. Выражение @z означает в первом случае вызов z.operator @(), во втором - вызов operator @(z).
- ✓ Любая бинарная операция @ определяется также двумя способами: либо как компонентная функция с одним параметром, либо как глобальная (возможно дружественная) функция с двумя параметрами. В первом случае x @ y означает вызов x.operator @(y), во втором – вызов operator @(x, y).
- ✓ Перегруженная операция не может иметь аргументы (операнды), заданные по умолчанию.
- ✓ В языке C++ установлена идентичность некоторых операций, например, ++z – это тоже, что и z += 1. Эта идентичность теряется для перегруженных операций.
- ✓ Функцию operator можно вызвать по ее имени, например, z = operator * x, y) или z = x.operator *(y). В первом случае вызывается глобальная функция, во втором – компонентная функция класса X, и x – это объект класса X. Однако, чаще всего функция operator вызывается косвенно, например, z = x * y.

- ✓ За исключением перегрузки операций new и delete функция operator должна быть либо нестатической компонентной функцией, либо иметь как минимум один аргумент (операнд) типа "класс" или "ссылка на класс" (если это глобальная функция).
- ✓ Операции '=', '[]', '–>' можно перегружать только с помощью нестатической компонентной функции operator @. Это гарантирует, что первыми операндами будут леводопустимые выражения.
- ✓ Операция '[]' рассматривается как бинарная. Пусть a – объект класса A, в котором перегружена операция '[]'. Тогда выражение a[i] интерпретируется как a.operator [](i).
- ✓ Операция '()' вызова функции рассматривается как бинарная. Пусть a – объект класса A, в котором перегружена операция '()'. Тогда выражение a(x1, x2, x3, x4) интерпретируется как a.operator ()(x1, x2, x3, x4).
- ✓ Операция '–>' доступа к компоненту класса через указатель на объект этого класса рассматривается как унарная. Пусть a – объект класса A, в котором перегружена операция '–>'. Тогда выражение a–>m интерпретируется как (a.operator–>())–>m. Это означает, что функция operator –>() должна возвращать указатель на класс A, или объект класса A, или ссылку на класс A.
- ✓ Перегрузка операций '++' и '--', записываемых после операнда (z++, z--), отличается добавлением в функцию operator фиктивного параметра int, который используется только как признак отличия операций z++ и z-- от операций ++z и --z.
- ✓ Глобальные операции new можно перегрузить и в общем случае они могут не иметь аргументов (операндов) типа "класс". В результате разрешается иметь несколько глобальных операций new, которые различаются путем изменения числа и (или) типов аргументов.
- ✓ Глобальные операции delete не могут быть перегружены. Их можно перегрузить только по отношению к классу.
- ✓ Заданные в самом языке глобальные операции new и delete можно изменить, т.е. заменить версию, заданную в языке по умолчанию, на свою версию.
- ✓ Локальные функции operator new() и operator delete() являются статическими компонентами класса, в котором они определены, независимо от того, использовался или нет спецификатор static (это, в частности, означает, что они не могут быть виртуальными).
- ✓ Для правильного освобождения динамической памяти под базовый и производный объекты следует использовать виртуальный деструктор.
- ✓ Если для класса X операция "=" не была перегружена явно и x и y - это объекты класса X, то выражение x = y задает по умолчанию побайтовое копирование данных объекта y в данные объекта x.

- ✓ Функция operator вида operator type() без возвращаемого значения, определенная в классе A, задает преобразование типа A к типу type.
- ✓ За исключением операции присваивания '=' все операции, перегруженные в классе X, наследуются в любом производном классе Y.
- ✓ Пусть X – базовый класс, Y – производный класс. Тогда локально перегруженная операция для класса X может быть далее повторно перегружена в классе Y.

Вопросы

1. Каково назначение перегрузки операторов?
2. Как используется ключевое слово слово operator?
3. Можно ли перегрузкой отменить очередность выполнения операции?
4. Истинно ли следующее утверждение: операция >= может быть перегружена.
5. Сколько аргументов требуется для определения перегруженной унарной операции?
6. Когда вы перегружаете операцию арифметического присваивания куда передается результат?
7. Истинно ли следующее утверждение: выражение objA = objB будет причиной ошибки компилятора, если объекты разных типов?
8. Можно ли перегрузкой отменить число операндов?
9. Какие операции требуют, чтобы левый операнд был объектом класса?
10. Какие операторы нельзя перегружать?
11. Запишите приведенный ниже пример как дружественный оператор класса dat:

```
dat     dat::operator+(int n)
{
     dat     x;
     x     = *this;
     while (n-- != 0) x.next();
     return(x);
}
```

12. Наследуется ли перегрузка операции присваивания?
13. Может ли операция = переопределяться вне области определения класса?
14. Можно ли использовать операцию =, если она не определена?
15. Можно ли перегружать операцию []с использованием friend-функции ?

16. Можно ли перегружать операцию ()с использованием friend-функции ?

17. Можно ли перегружать операцию -> с использованием friend-функции ?

18. Может ли перегруженная операция delete возвращать значение void?

19. Может ли перегруженная операция delete возвращать значение int ?

20. Можно ли перегружать операцию delete с использованием friend-функции?

21. Для чего используется оператор dynamic_cast?

22. Для чего используется оператор const_cast?

23. Для чего используется оператор static_cast?

24. Для чего используется оператор reinterpret_cast?

25. Как называется вызов функции, обрабатываемый во время выполнения программы?

26. Что будет выведено на экран?

```
class A {
public:
   A(){ };
   ~A(){ };
   explicit A(int a);
   operator int(){return 1;}
};

int main(int argc, char* argv[])
{
   A foo;
   int value = foo + 1;
   std::cout << value << std::endl;
   return 0;
}
```

27. Как реализовать перегрузку инкеремнта и дикремента в постфиксной и префиксной формах?