# Перегрузка операций

Перегрузка – переопределение действия стандартных операций для работы с собственными типами данных, в том числе объектами. Можно перегружать любые операции кроме “.”, “::”, “.*”, “?”.

Перегрузка операций осуществляется с помощью методов специального вида – *функций-операций*. При перегрузке операций сохраняются количество аргументов, приоритеты операций и правила ассоциации, используемые в стандартных типах данных.

Функция-операция содержит ключевое слово operator, за которым следует знак переопределяемой операции:

```
тип operator операция (список параметров) {тело функции}
```

## Перегрузка унарных операций

Унарная функция-операция, определяемая *внутри класса*, должна быть представлена с помощью нестатического метода без параметров, при этом операндом является вызвавший ее объект, например:

```
class monstr{
   …
   monstr & operator ++() {++health; return *this;}
}
monstr Vasia;
cout << (++Vasia).get_health();
```

Если функция определяется *вне класса*, она должна иметь один параметр типа класса:

```
class monstr{
   …
   friend monstr & operator ++( monstr &M);
};
monstr& operator ++(monstr &M) {++M.health; return M;}
```

Операции постфиксного инкремента и декремента должны иметь первый параметр типа int. Он используется только для того, чтобы отличить их от префиксной формы:

```
class monstr{
   …
   monstr operator ++(int){
      monstr M(*this); health++;
      return M;
   }
};
monstr Vasia;
cout << (Vasia++).get_health();
```

## Перегрузка бинарных операций

Бинарная функция-операция, определяемая *внутри класса*, должна быть представлена с помощью нестатического метода с параметрами, при этом вызвавший ее объект считается первым операндом:

```
class monstr{
   …
   bool operator >(const monstr &M){
      if( health > M.health) return true;
      return false;
   }
};
```

Если функция определяется *вне класса*, она должна иметь два параметра типа класса:

```
bool operator >(const monstr &M1, const monstr &M2){
   if( M1.get_health() > M2.get_health()) return true;
   return false;
}
```

## Перегрузка операции присваивания

По умолчанию операция присваивания объектов копирует содержимое всех полей. Если класс содержит поля, память под которые выделяется динамически, необходимо определить оператор присваивания:

```
const monstr& operator = (const monstr &M){
   // Проверка на самоприсваивание:
   if (&M == this) return *this;
   if (name) delete [] name;
   if (M.name){
      name = new char [strlen(M.name) + 1];
      strcpy(name, M.name);}
   else name = 0;
   health = M.health; ammo = M.ammo; skin = M.skin;
return *this;
}
```

## Перегрузка операций new и delete

Чтобы обеспечить альтернативные варианты управления памятью, можно определять собственные варианты операций new и new[] для выделения динамической памяти под объект и массив объектов соответственно, а также операции delete и delete [] для ее освобождения.

Им не требуется передавать параметр типа класса. Первым параметром функциям new и delete должен передаваться размер объекта, который имеет тип size_t (размер возвращается оператором `sizeof тип`):

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

### Указатели на элементы классов

К элементам классов можно обращаться с помощью указателей. Для этого определены операции .* и ->*. Указатели на поля и методы класса определяются по-разному.

Формат *указателя на метод класса*:

```
возвр_тип (имя_класса::*имя_указателя)(параметры);
```

Например, описание указателя на методы класса monstr

```
int get_health() {return health;}
int get_ammo() {return ammo;}
```

(а также на другие методы этого класса с такой же сигнатурой) будет иметь вид:

```
int (monstr:: *pget)();
```

Формат *указателя на поле класса*:

```
тип_данных(имя_класса::*имя_указателя);
```

В определение указателя можно включить его инициализацию в форме:

```
&имя_класса::*имя_поля;  // Поле должно быть public
```

Если бы поле health было объявлено как public, определение указателя на него имело бы вид:

```
int (monstr::*phealth) = &monstr::health;
cout << Vasia.*phealth;  // Обращение через операцию .*
cout << p->*phealth;     // Обращение через операцию ->*
```

## Наследование

Механизм наследования классов позволяет строить иерархии, в которых производные классы получают элементы родительских, или базовых, классов и могут дополнять их или изменять их свойства. Множественное наследование позволяет одному классу обладать свойствами двух и более родительских классов.

### Ключи доступа

При описании класса в его заголовке перечисляются все классы, являющиеся для него базовыми. Возможность обращения к элементам этих классов регулируется с помощью *ключей доступа* **private, protected** и **public:**

**class имя : [private | protected | public] базовый_класс { тело класса };**

Если базовых классов несколько, они перечисляются через запятую. Ключ доступа может стоять перед каждым классом, например:

```
class A { ... };
class B { ... };
class C { ... };
class D: A, protected B, public C { ... };
```

По умолчанию для классов используется ключ доступа **private,** а для структур — **public**. Помимо спецификаторов доступа private и public для любого элемента класса может также использоваться спецификатор protected, который для одиночных классов, не входящих в иерархию, равносилен private. Разница между ними проявляется при наследовании, что можно видеть из таблицы:

| Ключ доступа | Спецификатор в базовом классе | Доступ в производном классе |
|---|---|---|
| private | private | нет |
| | protected | private |
| | public | private |
| protected | private | нет |
| | protected | protected |
| | public | protected |
| public | private | нет |
| | protected | protected |
| | public | public |

Элементы protected при наследовании с ключом private становятся в производном классе private, в остальных случаях права доступа к ним не изменяются. Доступ к элементам publ ic при наследовании становится соответствующим ключу доступа.

```
// ------------  Класс daemon ------------
class daemon : public monstr{
      int brain;
   public:
      // ------------ Конструкторы:
      daemon(int br = 10){brain = br;};
      daemon(color sk) : monstr (sk) {brain = 10;}
      daemon(char * nam) : monstr (nam) {brain = 10;}
      daemon(daemon &M) : monstr (M) {brain = M.brain;}
      // ------------ Операции:
      const daemon& operator = (daemon &M){
         if (&M == this) return *this;
         brain = M.brain;
         monstr::operator = (M);
         return *this;
      }
      // ------------ Методы, изменяющие значения полей:
      void think();
      // ------------ Прочие методы:
      void draw(int x, int y, int scale, int position);
};
// ------------ Реализация класса daemon ------------
void daemon::think(){ /* … */ }
void daemon::draw(int x, int y, int scale, int position)
{ /* … Отрисовка daemon */ }
```