

# Лекция 4

Перегрузка операций

# Перегрузка операций

## Операция выбора элемента

Операция выбора элемента '->' так же может быть перегружена. Она относится к унарным операциям своего левого операнда. Этот левый операнд должен быть либо объектом класса, либо ссылкой на объект класса, для которого операция перегружается. Функция `operator->` обязательно должна быть составной нестатической функцией класса.

# Перегрузка операций

В качестве результата операция должна возвращать либо объект, либо указатель на объект. К этому указателю затем автоматически будет применена предопределенная операция выбора (по указателю ->).

Если операция -> перегружена для класса Name, тогда выражение name-> m, где name – объект класса Name, интерпретируется как (name.operator->())->m.

# Перегрузка операций

Обычно данную операцию имеет смысл перегружать в том случае, когда иерархия классов сильно разветвлена.

Рассмотрим пример.

```
struct N  
{  
    int a;  
};
```

# Перегрузка операций

```
struct L1
{
    N *target;
    N *operator->() const
        { return target; }
};
```

```
struct L2
{
    L1 *target;
    L1 &operator->() const
        { return * target; }
};
```

# Перегрузка операций

Использование перегруженных операций:

```
int main()
{
    N x = { 3 };
    L1 y = { & x };
    L2 z = { & y };
    cout << x.a << y->a << z->a << endl;    // print
    "333"
    return 0;
}
```

# Перегрузка операций

Если изменить перегруженную операцию класса struct L2 следующим образом:

```
L1 &operator->() const
{
    if(target == 0)
    {
        cout << " Список пустой " << endl;
        exit(-1);
    }
    return * target;
}
```

# Перегрузка операций

Тогда при объявлении

$N\ x = \{ \};$

$L1\ y = \{ \};$

$L2\ z = \{ \};$

Получим сообщение о пустоте списка.



# Перегрузка операций

## Перегрузка операции приведения типа

Существует возможность определить функцию-операцию, которая будет осуществлять преобразование объектов класса к другому типу. Общий формат:  
operator имя\_типа ();

Тип возвращаемого значения и параметры указывать не требуется. Можно переопределять методы как виртуальные, которые можно использовать в иерархии классов.

# Перегрузка операций

Рассмотрим несложный пример:

```
class Test
{
    int test;
public:
    Test(){};
    Test(int t):test(t){};
    operator int()
    { return test; }
};
```

# Перегрузка операций

Использование данного оператора:

```
Test t(190);
```

```
int i = int(t);
```

```
cout << i << endl;
```

Или более в «экзотическом» виде:

```
cout << t.operator int() << endl;
```

# Перегрузка операций

## **Перегрузка операции вызова функции**

Класс в котором определена операция вызова функции, называется функциональным классом. При этом, от такого класса не требуется наличия каких-либо полей и других методов.

Рассмотрим классический пример функционального класса.

# Перегрузка операций

```
class if_greater
{
public:
    int operator()(int a, int b)
    { return a>b; }
};
```

# Перегрузка операций

Использование данной операции несколько необычно по сравнению с другими операциями:

```
int main()
{
    if_greater x;
    cout << x(1,5) << endl;
    cout << if_greater()(5,1) << endl;
    return 0;
}
```

# Перегрузка операций

Перегруженная операция вызова функции приводит к важному понятию, используемому во многих языках программирования – *функциональный объект* или *функтор*.

В терминах C++ функции не являются объектами, поэтому они функторами не считаются, хотя имена функций, не функции, а именно имена функций входят в категорию функциональных объектов.

# Перегрузка операций

Имя функции — это идентификатор, который неявно приводится к указателю (как и имя массива).

Поскольку указатели в терминах C++ объектами считаются, то и имя функции, приведённое к указателю, считается объектом, а поскольку оно объект, то можно говорить о функциональности объекта.



# Перегрузка операций

Посмотрим еще один пример использования функтора, когда в классе перегружается операция (), а потом объект класса используется подобно функции, вбирая в себя аргументы.

# Перегрузка операций

```
class Less
{
public:
    bool operator() (const int left, const int right)
    { return left < right; }
};

int main()
{
    int arr[] = {1,2,3,4,5};
    Less less;    //создаётся функциональный объект
    sort(begin(arr), end(arr), less); //передаётся функтор в алгоритм
    сортировки
}
```

# Перегрузка операций

## Перегрузка операций `new` и `delete`

При перегрузке операций `new` и `delete` следует руководствоваться

Международным стандартом по языку C++:

ISO/IEC 14882,

[https://en.cppreference.com/w/cpp/memory/new/operator\\_new](https://en.cppreference.com/w/cpp/memory/new/operator_new)

<http://www.cplusplus.com/reference/cstdint/>

# Перегрузка операций

Чтобы обеспечить альтернативные варианты управления памятью, можно определить свои собственные варианты операций `new` и `delete`.

Перегрузить можно и глобальные операции, описав их вне всякого блока (структуры, класса).

# Перегрузка операций

Пример перегрузки глобальных операций:

```
#include <new>
```

```
void * operator new ( size_t sz )
```

```
{
```

```
    cout << " новая глобальная операция new " << sz << endl;
```

```
    if ( sz == 0 )
```

```
        ++ sz ; // избегайте malloc (0), который может вернуть nullptr в случае  
успеха
```

```
    if ( void * ptr = malloc ( sz ) )
```

```
        return ptr;
```

```
    throw bad_alloc();
```

```
}
```

# Перегрузка операций

```
void operator delete ( void * ptr )  
{  
    cout << " новая глобальная операция delete" << endl;;  
    free ( ptr ) ;  
}
```

# Перегрузка операций

Использование операций:

```
int main ( )  
{  
    int * p1 = new int ;  
    delete p1 ;  
  
    int * p2 = new int [ 10 ] ;  
    delete [ ] p2 ;  
  
    return 0;  
}
```

# Перегрузка операций

В этом примере при объявлении массивов в динамической памяти сработают стандартные. Для полноты программы перегрузим операции и для массивов:

```
void * operator new []( size_t sz , int n)
{
    cout << " новая глобальная операция new []" << sz << endl;
    if ( sz == 0 )
        ++ sz ; //

    if ( void * ptr = malloc ( sz ) )
        return ptr;
    throw bad_alloc();
}
```



# Перегрузка операций

```
void operator delete[] ( void * ptr )  
{  
    cout << " новая глобальная операция delete"  
    << endl;  
    free ( ptr ) ;  
}
```

Теперь вызовы:

```
int * p2 = new int [ 10 ] ;  
delete [ ] p2 ;
```

получат отклик от программы

# Перегрузка операций

Эти примеры учебные и подобные перегрузки глобальных операций в практическом программировании обычно не делают.

# Перегрузка операций

## Перегрузка операций `new` и `delete` в теле класса

При перегрузке данных операций должны соответствовать следующим правилам:

- им не требуется передавать параметр типа класса;
- первым параметром функции `new` и `new[]` должен передаваться размер объекта типа `size_t` (возвращается операцией `sizeof`, содержится в файле `<stddef.h>`); при вызове функции передается неявным образом;

# Перегрузка операций

- операции `new` и `new[]` должны возвращать в качестве результата тип `void*`, даже если оператор `return` возвращает указатель на другие типы (чаще всего на тип класса);
- операции `delete` и `delete[]` должны возвращать тип `void` и первый аргумент типа `void*`;
- операции выделения и освобождения памяти являются статическими компонентами класса.

# Перегрузка операций

Рассмотрим пример перегрузки операций в теле класса:

```
struct X
```

```
{  
    static void * operator new (size_t sz )  
    {  
        cout << "custom new for size" << ' ' << sz << endl;  
        return :: operator new ( sz );  
    }  
    static void * operator new [ ] (size_t sz )  
    {  
        cout << "custom new [] для размера" << ' ' << sz << endl ;  
        return :: operator new ( sz );  
    }  
};
```

# Перегрузка операций

Вариант использования:

```
int main ( )  
{  
    X * p1 = new X ;  
    delete p1 ;  
    X * p2 = new X [ 10 ] ;  
    delete [ ] p2 ;  
    return 0 ;  
}
```

# Перегрузка операций

Перегрузка `operator new` и `operator new[]` с дополнительными определяемыми пользователем параметрами («формы размещения») также могут быть определены как составные функции класса.

# Перегрузка операций

Рассмотрим пример:

```
struct X
{
    X() { throw runtime_error(""); }
    static void* operator new(std::size_t sz, bool b)
    {
        cout << "custom placement new called, b = " << b << '\n';
        return ::operator new(sz);
    }
    static void operator delete(void* ptr, bool b)
    {
        cout << "custom placement delete called, b = " << b << '\n';
        ::operator delete(ptr);
    }
};
```



# Перегрузка операций

Использование объявлений:

```
int main()
{
    try
    {
        X* p1 = new (true) X;
    }
    catch(const exception&) { }
    return 0;
}
```

Посмотрите и оцените работу данной программы.

# Перегрузка операций

При переопределении операций `new` и `delete` рекомендуется использовать обработчик исключительных ситуаций.

# Указатели компонентов класса

## Указатели полей

Существует возможность создания указателя на нестатическую компоненту класса. Этот указатель отличается от обычного тем, что в его описании присутствует идентификатор класса. В частности, указатель типа 'int' относится к типу 'int \*', то указатель на целочисленную нестатическую компоненту класса Class, относится к типу 'int Class:: \*'.

# Указатели компонентов класса

Примеры:

Объявление компоненты      Тип указателя

int Fix;

int Class:: \*

float \*Num;

float \*Class:: \*

long (Ref \*)[2];

long (\*Class:: \*)[2]

void Sub(int);

void( Class::\*)(int)

Благодаря связи компонента класса с идентификатором класса возможно осуществление контроля правильности обращений к компонентам класса через указатели.

# Указатели компонентов класса

С точки зрения синтаксиса обращений применяется следующая нотация: если Ptr- выражение, указывающее на компонент класса, то \*Ptr – имя этого компонента.

Если Obj является именем объекта класса, а Ref – указателем на объект, то справедливы следующие записи:

Obj. \*Ptr

Ref-> \*Ptr

# Указатели компонентов класса

Рассмотрим простой класс:

```
class Fixed  
{  
public:  
    int Fix;  
};
```

# Указатели компонентов класса

Объявим указатель на единственное поле класса:

```
int Fixed::*Ref = &Fixed::Fix;
```

Имея объект класса `Fixed` и указатель на него можно обратиться к компоненте класса `Fix`.

```
Fixed Num;
```

```
Fixed *Ptr_Fixed = &Num; и
```

```
Ptr_Fixed->Fix = 12; // обращение через  
обычный указатель на класс или Num.Fix =  
12;
```

# Указатели компонентов класса

Обращение через объект класса:

```
cout << Num.*Ref << endl;
```

Обращение через указатель на класс:

```
cout << Ptr_Fixed->*Ref << endl;
```

Указатель на приватную или защищенную компоненту создать можно, но обратиться – нельзя.



# Указатели компонентов класса

## Указатели составляющих

Рассмотрим возможность создания указателя на составляющую функцию класса.

```
class Fixed
{
public:
    int Fix;
    int *FixPtr()
    {
        return &Fix;
    }
};
```

Указатель на составляющую функцию  
класса:

```
int *(Fixed:: *Ref)() = &Fixed::FixPtr;
```

Вызов функции через указатель:

```
Fixed Num, *Ptr = &Num;
```

```
Num.Fix = 13;
```

```
cout << *(Ptr->*Ref)() << endl;
```

Нельзя создать указатель на статическую  
функцию класса и на конструктор  
классаю

















