

Объектно-ориентированное программирование на C++

С.Ю. Скоробогатов

Весна 2016

Список литературы по модулю

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

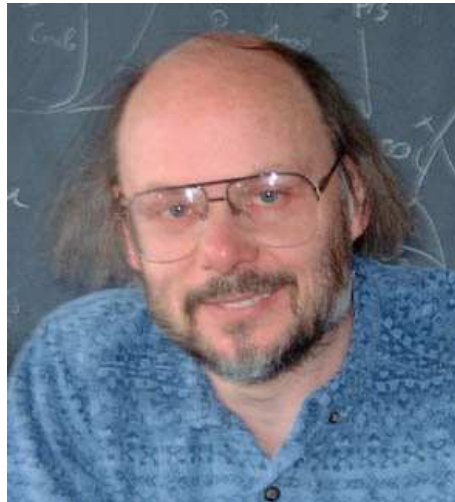
Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

1. Слайды лекций.
www.dropbox.com/s/sls69hmat8wli36/module_cpp.pdf
2. Б. Страуструп. Язык программирования C++. Специальное издание. – М.: Бином, 2011.
www.dropbox.com/s/xg7ax2qpbhwbw58/cppbook.djvu
3. Б. Страуструп. Дизайн и эволюция C++. – М.: ДМК Пресс; Спб.: Питер, 2006.
www.dropbox.com/s/cflejuy2wx7he6r/cppdesign.djvu



Б. Страуструп

Основные этапы:

Начало 80-х: C with classes, транслятор cfront.

1983: переименование в C++, виртуальные функции, перегрузка функций и операторов, ссылки.

1985: выход книги Страуструпа
The C++ Programming Language.

1989: C++ 2.0 (множественное наследование, абстрактные классы, обобщённые типы, статические функции-члены, функции-константы и защищённые члены).

1998: стандарт ISO/IEC 14882:1998 – C++98.

2003: стандарт ISO/IEC 14882:2003.

2011: стандарт ISO/IEC 14882:2011 – C++11 (выведение типов, расширенный for, замыкания).

Совместимость с C: C++ содержит в себе язык C как подмножество (с незначительными изменениями, связанными с более строгим контролем типов);

Поддержка ООП: в C++ реализованы классы с поддержкой множественного наследования, перегрузки операций и обобщённого программирования;

Небезопасность: в C++ ошибки при работе с памятью могут привести к непредсказуемой работе программы;

Ручное управление памятью: автоматическое управление памятью в C++ не реализовано из принципиальных соображений;

Объекты не только в куче: в отличие от языка Java, в C++ объекты не обязаны размещаться на верхнем уровне кучи – они могут «жить» в локальных и глобальных переменных, в элементах массивов, в полях других объектов.

Программа «Hello, World»

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Программа на C++ (hello.cpp):

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main()
6  {
7      cout << "Hello ,_world!" << endl;
8      return 0;
9  }
```

Компиляция:

```
g++ -o hello hello.cpp
```

Запуск:

```
./hello
```

Простые расширения языка C

Базовые
сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и
уничтожение
объектов

Копирование
объектов

Одиночное
наследование

Множественное
наследование

Обобщённое
программирова-
ние

Перегрузка
операций

Исключения

Однострочные комментарии. Начинаются с `//`.

Операторы-объявления. Объявление локальной переменной может располагаться в любом месте, где допустимо располагать оператор.

Объявления в заголовке `for`. В секции инициализации оператора `for` можно объявлять переменные.

Перегрузка функций. Допустимо объявлять несколько одноимённых функций с разными сигнатурами.

Теги как имена типов. Теги структур, объединений и перечислений – полноценные имена типов, которые можно использовать без ключевых слов `struct`, `union` и `enum`.

Булевский тип. Базовый тип `bool` с двумя значениями: `true` и `false`. Впрочем, для совместимости с C в условиях оператора `if` и циклов могут применяться числовые значения.

Ссылочный тип данных. *см. следующие слайды.*

Объявление переменных ссылочного типа

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Ссылка в C++ – это типизированный указатель, к которому неприменимы арифметические операции, и который не может быть нулевым. Кроме того, недопустимы ссылки на ссылки.

Для объявления ссылки используется префиксный декларатор «&»:

```
тип &имя_переменной;
```

При этом ссылки в глобальных и локальных переменных должны быть обязательно инициализированы при объявлении (забегая вперёд: ссылки в полях объектов должны быть инициализированы в конструкторе класса).

Более того, значение, полученное ссылкой при инициализации, в дальнейшем не может быть изменено.

Инициализация и использование ссылок

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Для того чтобы присвоить ссылке адрес некоторого значения в памяти, не нужно использовать операцию «&» для получения адреса объекта. Для доступа к значению, на которое указывает ссылка, не нужно использовать операцию разыменования «*».

Например,

```
int main()
{
    int x = 10;
    int &y = x;    // y указывает на x
    cout << y << " ";
    y = 20;        // меняем значение x через y
    cout << x << endl;
    return 0;
}
```

Вывод:

```
10 20
```


Ссылки в формальных параметрах функций

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Использование ссылок в качестве параметров функции позволяет имитировать var-параметры процедур и функций языка Pascal.

Например,

```
void swap(int &a, int &b) {  
    int t = a;  
    a = b;  
    b = t;  
}  
  
int main() {  
    int x = 10, y = 20;  
    swap(x, y);  
    cout << x << "□" << y << endl;  
    return 0;  
}
```

Вывод:

20 10

Ссылки как возвращаемые значения функций

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Ссылка может быть возвращаемым значением функции. При этом надо следить, чтобы случайно не вернуть ссылку на локальную переменную.

Например,

```
char &ith(char *s, int i)
{
    return s[i];
}

int main()
{
    char s[] = "qwerty";
    ith(s, 2) = 'x';
    cout << s << endl;
    return 0;
}
```

Вывод:

qwxrty

Как мы увидим в дальнейшем, особую роль играют константные ссылки, объявляемые с модификатором `const`:

```
const тип &имя_переменной;
```

Например,

```
int main()
{
    int x = 10;
    const int &y = x;
    cout << y << endl;
    y = 20; // Ошибка!
    return 0;
}
```

Сообщение об ошибке:

```
test.cpp: In function 'int main()':
test.cpp:11:6: error: assignment of
read-only reference 'y'
```

В языке C++ любая структура de facto является классом. Однако, классы принято объявлять с помощью конструкции

```
class Имя
{
    ...
};
```

Отличие этой конструкции от объявления структуры заключается в том, что по умолчанию все члены структуры (поля, методы и т.д.) доступны извне структуры, а члены класса – недоступны.

В теле класса располагаются объявления полей, прототипы методов и конструкторов, а также в некоторых случаях определения методов и конструкторов (злоупотреблять определениями не стоит, чтобы не мешать отдельной компиляции).

Секции в объявлении класса

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Для управления доступом к членам класса в C++ предусмотрены `public`-, `private`- и `protected`-секции внутри объявления класса:

```
class Имя
{
public:
    ...
private:
    ...
protected:
    ...
};
```

Секции могут быть перечислены внутри объявления класса в любом порядке и количестве. Члены класса, попадающие в `public`-секцию, видны отовсюду, члены из `private`-секции видны только из методов данного класса, а члены `protected`-секции – из методов данного класса и его классов-наследников.

Пример: секции в объявлении класса

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

```
class Person
{
public:
    string name;
    int year_of_birth;

    Person(string, int, string);
    string get_address();

private:
    string address;
};
```

В примере конструктор класса `Person`, а также метод `get_address` представлены прототипами. Их определения могут находиться в другом файле.

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

В языке C++ экземплярные поля класса объявляются также, как и поля структур в языке C. При этом объявления статических полей начинаются с модификатора `static`. По умолчанию поля недоступны извне класса, но доступны извне структуры или объединения.

Например,

```
class Point
{
public:
    int x, y;           // Координаты точки
    static int count;   // Общее количество точек
};
```

Отметим, что каждое статическое поле должно быть дополнительно *определено* в одном из единиц компиляции проекта (cpp-файле).

Определения статических полей

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Особенностью языка C++ является то, что он рассчитан на ту же схему компиляции, что и язык C. То есть программа на C++ состоит из набора сpp-файлов, каждый из которых компилируется в объектный файл. Чтобы связать эти объектные файлы, применяется линкер.

Так как класс в общем случае может использоваться в нескольких сpp-файлах, то в каждом файле должно быть его объявление. На практике объявление класса выносится в отдельный h-файл, который включается в нужные сpp-файлы.

Так как статическое поле – это фактически глобальная переменная, то она должна быть явно помещена в один из объектных файлов. Для этого в соответствующем сpp-файле должно быть дано её определение (имя статического поля выглядит как `Имя_класса::Имя_поля`).

```
int Point::count = 0;
```


Объявление метода на C++ состоит из прототипа и определения. Прототип метода помещается в тело класса, а определение – в нужный `cpp`-файл (тут та же история, что и со статическими полями). Прототипы статических методов объявляются с модификатором `static`.

Пример (объявление класса с прототипом метода `dist`):

```
class Point
{
public:
    double x, y;
    double dist();
};
```

Пример (определение метода `dist`):

```
double Point::dist()
{
    return sqrt(x*x + y*y);
}
```

Виртуальные и абстрактные методы

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

По умолчанию для вызова методов в C++ используется раннее связывание.

Чтобы включить позднее связывание, в объявлении прототипа метода надо указать модификатор `virtual`:

```
class Point
{
public:
    virtual double dist();
};
```

Прототип абстрактного метода заканчивается на «= 0»:

```
class Point
{
public:
    virtual double dist() = 0;
};
```

Класс, в котором есть абстрактный метод, автоматически становится абстрактным.

Объявление конструкторов

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

В C++ объявление экземплярного конструктора, как и объявление обычного экземплярного метода, состоит из прототипа конструктора, записываемого в теле класса, и определения конструктора, размещённого в одном из `cpp`-файлов. В отличие от объявления метода, в объявлении конструктора имя конструктора совпадает с именем класса, и для конструктора не указывается тип возвращаемого значения. Чтобы запретить создание экземпляра класса, достаточно объявить конструктор в `private`- или `protected`-секции класса. При этом, естественно, в `public`-секции ни одного конструктора не должно быть объявлено.

Специфика реализации конструкторов в языке C++ диктует следующий крайне нетипичный для современных языков принцип: никогда не вызывайте из конструктора виртуальные методы – это может привести к проблемам в случае переопределения этих методов в классах-наследниках.

Пример: объявление конструкторов

Базовые сведения
Введение
Ссылки
Классы
Поля и методы
Конструкторы
Создание и уничтожение объектов
Копирование объектов
Одиночное наследование
Множественное наследование
Обобщённое программирование
Перегрузка операций
Исключения

```
1  class Point
2  {
3  private:
4      double x, y;
5  public:
6      Point(double x, double y);
7  };
8
9  Point::Point(double x, double y)
10 {
11     this->x = x;  this->y = y;
12 }
```

Ключевое слово `this` в теле метода (в том числе и конструктора) обозначает указатель на объект, для которого вызван конструктор. Мы вынуждены использовать `this` для доступа к полям объекта, так как имена параметров конструктора совпадают с именами полей.

Создание объектов в динамической памяти

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Рассмотрим, как в C++ принято создавать объекты классов. При этом мы на данном этапе изучения C++ ограничимся созданием объектов в динамической памяти.

Вообще, динамическое выделение памяти под значение любого типа выполняется в C++ операцией `new`, имеющей следующий синтаксис:

```
new имя_примитивного_типа
```

ИЛИ

```
new имя_примитивного_типа (начальное_значение)
```

ИЛИ

```
new имя_класса (фактические_параметры_конструктора)
```

Например,

```
int *x = new int;  
float *y = new float(3.14);  
Point *p = new Point(1.5, 2.3);
```

Создание массивов в динамической памяти

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Операция `new` совмещает выделение памяти с инициализацией (с вызовом конструктора) и возвращает указатель на созданный и инициализированный объект.

В C++ вообще не принято использовать функцию `malloc`, потому что для создания массивов также используется специальная форма операции `new`:

```
new тип_элемента [размер]
```

Например,

```
int *a = new int [10];  
Point **pa = new Point* [20];
```

Обратите внимание на то, что массив объектов можно выделять только в случае, если в классе определён конструктор по умолчанию. При этом этот конструктор будет вызван для каждого объекта в массиве.

```
Point *b = new Point [20]; // ошибка
```

Удаление объектов и массивов

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Так как в C++ не предусмотрено автоматическое управление памятью (нет сборщика мусора), то все объекты, созданные операцией `new`, должны быть в какой-то момент явно освобождены.

Для этого существует оператор `delete`, имеющий следующий синтаксис:

```
delete указатель_на_объект;      // для объектов  
delete [] указатель_на_массив;  // для массивов
```

Например,

```
Point *p = new Point(10.0, 20.0);  
delete p;  
int *a = new int [20];  
delete [] a;
```

Объявление деструктора

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Внутреннее состояние объекта может содержать указатели (ссылки) на другие объекты, созданные в его конструкторе. В языке без сборки мусора освобождение такого объекта может привести к утечке памяти.

Деструктор – это экземплярный метод, предназначенный для освобождения ресурсов, принадлежащих объекту, непосредственно перед уничтожением этого объекта.

В языке C++ деструкторы жизненно необходимы. Прототип деструктора имеет вид

```
~имя_класса();
```

Деструктор вызывается при уничтожении объекта. В частности, его автоматически вызывает оператор `delete`.

Пример: объявление деструктора

Базовые
сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и
уничтожение
объектов

Копирование
объектов

Одиночное
наследование

Множественное
наследование

Обобщённое
программирова-
ние

Перегрузка
операций

Исключения

```
1  class IntArray {
2  private:
3      int *a;
4      std::size_t n;
5  public:
6      IntArray(std::size_t n);
7      int &at(int i);
8      virtual ~IntArray();
9  };
10
11  IntArray::IntArray(std::size_t n): n(n) {
12      a = new int [n];
13  }
14
15  int &IntArray::at(int i) {
16      return a[i];
17  }
18
19  IntArray::~~IntArray() {
20      delete [] a;
21  }
```

Объекты в автоматической памяти

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

В отличие от языка Java, в C++ объекты могут размещаться в глобальных и локальных переменных, в параметрах методов, в элементах массивов и полях других объектов. Синтаксис объявления переменной, содержащей объект, выглядит как

```
имя_класса имя_переменной (параметры_конструктора);
```

Такие объявления, кстати, можно применять и для переменных, тип которых не является классом.

Примеры:

```
IntArray a(100);  
Point p(10.5, 15.0);  
int i(666);  
const char *s("qwerty");
```

Объявление переменной, содержащей объект, вызывает экземплярный конструктор класса. Более того, при выходе из блока, где такая переменная объявлена, автоматически вызывается деструктор.

Пример: автоматический вызов деструктора

Базовые сведения	1	<code>#include <iostream></code>
Введение	2	
Ссылки	3	<code>using namespace std;</code>
Классы	4	
Поля и методы	5	<code>class Demo {</code>
Конструкторы	6	<code>private:</code>
Создание и уничтожение объектов	7	<code>int x;</code>
Копирование объектов	8	<code>public:</code>
Одиночное наследование	9	<code>Demo(int x);</code>
Множественное наследование	10	<code>virtual ~Demo();</code>
Обобщённое программирование	11	<code>};</code>
Перегрузка операций	12	
Исключения	13	<code>Demo::Demo(int x)</code>
	14	<code>{</code>
	15	<code> this->x = x;</code>
	16	<code> cout << "cons:" << x << "␣";</code>
	17	<code>}</code>
	18	
	19	<code>Demo::~~Demo() { cout << "destr:" << x << "␣"; }</code>

Пример: автоматический вызов деструктора (продолжение)

Базовые сведения
Введение
Ссылки
Классы
Поля и методы
Конструкторы
Создание и уничтожение объектов
Копирование объектов
Одиночное наследование
Множественное наследование
Обобщённое программирование
Перегрузка операций
Исключения

```
21  int main()  
22  {  
23      Demo d(100);  
24      for (int i = 0; i < 2; i++) {  
25          Demo d2(i);  
26      }  
27      return 0;  
28  }
```

Вывод:

```
cons:100  cons:0  destr:0  cons:1  destr:1  destr:100
```

Объекты в глобальной памяти

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Если объект размещается в глобальной переменной или в статическом поле класса, то конструктор для него вызывается до передачи управления в функцию `main`, а деструктор – после завершения функции `main`.

Пример:

```
Demo d(100);
```

```
int main()
{
    cout << "main_";
    return 0;
}
```

Вывод:

```
cons:100 main destr:100
```

Объекты в полях других объектов

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Если объект класса X содержится в поле x объекта Y , то конструктор для этого поля вызывается из конструктора Y посредством следующей синтаксической конструкции:

```
Y::Y(формальные_параметры_конструктора_Y)
    : x(фактические_параметры_конструктора_X) {
    ...
}
```

Между прочим, любое поле объекта может быть инициализировано таким образом, даже если тип этого поля не является классом. Например, конструктор класса `Demo` может быть переписан как

```
Demo::Demo(int x): x(x) {
    cout << "cons:" << x << "␣";
}
```

Пример: объекты в полях других объектов

Базовые
сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и
уничтожение
объектов

Копирование
объектов

Одиночное
наследование

Множественное
наследование

Обобщённое
программирова-
ние

Перегрузка
операций

Исключения

```
class Cont {  
private:  
    Demo d;  
public:  
    Cont(int x);  
};  
  
Cont::Cont(int x): d(x) {}  
  
int main()  
{  
    Cont c(100);  
    return 0;  
}
```

Вывод:

```
cons:100 destr:100
```

Обратите внимание: деструктор класса Demo был автоматически вызван при уничтожении объекта класса Cont.

Проблема копирования объектов

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Следует понимать, что передача объекта в качестве параметра при вызове метода автоматически влечёт создание копии объекта. Смысл копирования заключается в том, что изменение копии объекта внутри метода не приводит к изменению объекта-оригинала (семантика копирования).

Кроме этого, копирование объектов осуществляется при инициализации объявляемых переменных.

По умолчанию создаётся побитовая копия внутреннего состояния объекта. Однако, во многих случаях для обеспечения семантики копирования побитового копирования недостаточно.

Например, для рассмотренного ранее класса `IntArray` побитовое копирование объекта приводит к тому, что возникает два объекта, которые содержат в поле `a` указатели на один и тот же массив целых чисел. Поэтому после уничтожения одного из них второй объект «потеряет» массив.

Объявление конструктора копий

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Для создания «правильных» копий объектов необходимо объявить конструктор копий, прототип которого выглядит как

```
имя_класса(const имя_класса &obj);
```

Этот конструктор будет автоматически вызываться всякий раз, когда необходимо создать копию объекта.

Пример:

```
class IntArray {  
    ...  
    IntArray(const IntArray &obj);  
    ...  
};  
  
IntArray::IntArray(const IntArray &obj): n(obj.n) {  
    a = new int [n];  
    std::copy(obj.a, obj.a + n, a);  
}
```

Перегруженная операция присваивания

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Проблема копирования объектов возникает также при присваивании. Поэтому, забегаая вперёд, покажем на примере класса `IntArray`, как определить свою операцию присваивания:

```
class IntArray {
    ...
    IntArray& operator= (const IntArray &obj);
    ...
};

IntArray& IntArray::operator= (const IntArray &obj) {
    if (this != &obj) {
        int *new_a = new int [obj.n];
        std::copy(obj.a, obj.a + obj.n, new_a);
        delete [] a;
        n = obj.n;
        a = new_a;
    }
    return *this;
}
```

Основной способ наследования в C++

Базовые
сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и
уничтожение
объектов

Копирование
объектов

Одиночное
наследование

Множественное
наследование

Обобщённое
программирова-
ние

Перегрузка
операций

Исключения

Синтаксис одиночного наследования в C++ выглядит как

```
class ИмяПроизводногоКласса: public ИмяБазовогоКласса
{
    ...
};
```

Если конструктор базового класса не имеет параметров (является конструктором по умолчанию), его вызов добавляется компилятором C++ в конструктор производного класса автоматически.

В противном случае необходимо явно вызывать конструктор базового класса в конструкторе производного класса.

В языке C++ вызов конструктора базового класса X из конструктора производного класса Y выполняется аналогично вызовам конструкторов полей:

```
Y::Y(формальные_параметры_конструктора_Y)
    : X(фактические_параметры_конструктора_X)
{ ... }
```

Пример: одиночное наследование

Базовые сведения	1	<code>class Animal</code>
Введение	2	<code>{</code>
Ссылки	3	<code>private:</code>
Классы	4	<code> string species;</code>
Поля и методы	5	<code>public:</code>
Конструкторы	6	<code> Animal(const string &species);</code>
Создание и уничтожение объектов	7	<code>};</code>
Копирование объектов	8	
Одиночное наследование	9	<code>Animal::Animal(const string &species):</code>
Множественное наследование	10	<code> species(species) {}</code>
Обобщённое программирование	11	
Перегрузка операций	12	<code>class Dog: public Animal</code>
Исключения	13	<code>{</code>
	14	<code>private:</code>
	15	<code> string breed;</code>
	16	<code>public:</code>
	17	<code> Dog(const string &bread);</code>
	18	<code>};</code>
	19	
	20	<code>Dog::Dog(const string &bread):</code>
	21	<code> Animal("Canis_lupus_familiaris"), breed(breed) {}</code>

Переопределение методов

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

В C++, как и в языке Java, для переопределения метода, унаследованного от базового класса, достаточно объявить и определить этот метод в производном классе.

В качестве примера рассмотрим базовый класс `Animal`:

```
1  class Animal
2  {
3  private:
4      string species;
5  public:
6      Animal(const string &species);
7      virtual void draw();
8  };
9
10 Animal::Animal(const string &s): species(s) { }
11
12 void Animal::draw() {
13     cout << "    @ @    \n"
14            "    ('o')    \n"
15            "    o(____)o  \n";
16 }
```

Пример: переопределение методов

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Переопределим метод `draw` в производном классе `Dog`:

```
18 class Dog: public Animal
19 {
20 private:
21     string breed;
22 public:
23     Dog(const string &bread);
24     virtual void draw();
25 };
26
27 Dog::Dog(const string &bread):
28     Animal("Canis lupus familiaris"), breed(bread) {}
29
30 void Dog::draw() {
31     cout << "      / |__      \n"
32            "      (      @' ___ \n"
33            "      /              0 \n"
34            "      /      (_____/ \n"
35            "      /_____/      U   \n";
36 }
```

Динамическое приведение типов

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Операция динамического приведения объекта `obj` к типу `T` проверяет, является ли тип `T` одним из типов объекта `obj`, и возвращает `obj`, если является. В противном случае операция либо возвращает нулевой указатель, либо порождает исключение.

В C++ операция динамического приведения типа может применяться к указателям и ссылкам на объекты и записывается как

```
dynamic_cast <T> (obj)
```

Так как нулевых ссылок не существует, то при неудачном приведении порождается исключение `std::bad_cast`.

Пример: динамическое приведение типов не всегда допустимо

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

```
1  class Animal{};
2  class Dog: public Animal{};
3  class Cat: public Animal{};
4
5  int main()
6  {
7      Animal *a = new Dog();
8      Dog *dog = dynamic_cast<Dog*> (a);
9      cout << dog;
10     return 0;
11 }
```

g++ выдаёт ошибку:

```
cannot dynamic_cast 'a' (of type 'class Animal*') to
type 'class Dog*' (source type is not polymorphic)
```

Т.к. в классе Dog нет виртуальных методов, то для экономии памяти объект этого класса не содержит информации о типе, тем самым делая невозможным динамическое приведение типов.

Пример: динамическое приведение типов

Базовые сведения	1	<code>class Animal</code>
Введение	2	<code>{</code>
Ссылки	3	<code>public:</code>
Классы	4	<code>virtual void f() {}</code>
Поля и методы	5	<code>};</code>
Конструкторы	6	<code>class Dog: public Animal{};</code>
Создание и уничтожение объектов	7	<code>class Cat: public Animal{};</code>
Копирование объектов	8	
Одиночное наследование	9	<code>int main()</code>
Множественное наследование	10	<code>{</code>
Обобщённое программирование	11	<code>Animal *a = new Dog(), *b = new Cat();</code>
Перегрузка операций	12	<code>Dog *x = dynamic_cast<Dog*> (a),</code>
Исключения	13	<code>*y = dynamic_cast<Dog*> (b);</code>
	14	<code>cout << x << ", " << y;</code>
	15	<code>return 0;</code>
	16	<code>}</code>

Вывод:

0x603010 , 0

private- и protected-наследование

Базовые
сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и
уничтожение
объектов

Копирование
объектов

Одиночное
наследование

Множественное
наследование

Обобщённое
программирова-
ние

Перегрузка
операций

Исключения

Наследование от нескольких базовых классов

Базовые
сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и
уничтожение
объектов

Копирование
объектов

Одиночное
наследование

Множественное
наследование

Обобщённое
программирова-
ние

Перегрузка
операций

Исключения

С++ поддерживает множественное наследование. Если класс Y – производный от нескольких базовых классов X_1 , X_2 , ..., X_n , то объявление класса Y выглядит на С++ как

```
class Y: мод X1, мод X2, ..., мод Xn
{
    ...
};
```

Здесь «мод» – это `public`, `protected` или `private`.

При этом в конструкторе класса Y должны выполняться вызовы конструкторов всех базовых классов, кроме, возможно, тех из них, которые имеют конструкторы по умолчанию:

```
Y::Y(...):
    X1(...),
    X2(...),
    ...,
    Xn(...) {
    ...
}
```

Противоречия в именах наследуемых членов класса

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

При множественном наследовании в производном классе могут возникать два или более методов с совпадающими именами и сигнатурами. То же самое справедливо и для полей. Пример:

```
1  class A { public: virtual void m() { } };
2  class B { public: virtual void m() { } };
3  class C: public A, public B { };
4
5  int main()
6  {
7      C c;
8      c.m();
9      return 0;
10 }
```

g++ выдаёт ошибку в строке 8:

```
error: request for member 'm' is ambiguous
```

Разрешение противоречий в именах

Базовые
сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и
уничтожение
объектов

Копирование
объектов

Одиночное
наследование

Множественное
наследование

Обобщённое
программирова-
ние

Перегрузка
операций

Исключения

Для разрешения противоречий в именах наследуемых членов класса необходимо использовать квалифицированные имена.

Пример:

```
1  class A { public: virtual void m() { } };
2  class B { public: virtual void m() { } };
3  class C: public A, public B { };
4
5  int main()
6  {
7      C c;
8      c.A::m();
9      return 0;
10 }
```

Иерархия наследования и классы противоречия

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Иерархия наследования – это ориентированный ациклический граф, множеством узлов которого является множество классов программы. При этом если класс Y является непосредственным базовым классом для класса X , то из узла X исходит дуга, входящая в Y .

Мы будем говорить, что класс B является *классом противоречия*, если существует такой класс A , что в иерархии наследования можно провести не менее двух непересекающихся путей из A в B .

Иерархия наследования называется *противоречивой*, если в ней существуют классы противоречия.

Основная проблема противоречивых иерархий

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Основной проблемой противоречивых иерархий является возможность многократного включения полей базового класса в производный класс.

Пример:

```
1  class A
2  {
3  private:
4      int x;
5  };
6  class B: public A {};
7  class C: public A {};
8  class D: public B, public C {};
```

Поле `x` будет содержаться в объекте класса `D` дважды.

Виртуальное наследование

Базовые сведения

Введение

Ссылки

Классы

Поля и методы

Конструкторы

Создание и уничтожение объектов

Копирование объектов

Одиночное наследование

Множественное наследование

Обобщённое программирование

Перегрузка операций

Исключения

Избежать многократного включения базового класса в производный класс позволяет виртуальное наследование.

Виртуальное наследование – это способ реализации наследования, гарантирующий, что базовый класс не будет включён ни в один из производных классов более чем в одном экземпляре.

Если класс В виртуально наследует классу А, то говорят, что А – *виртуальный базовый класс* для класса В. Синтаксически это выражается как

```
class B: мод virtual A
{
    ...
};
```

Здесь «мод» – это `public`, `protected` или `private`.

Пример: виртуальное наследование

Базовые сведения	1	<code>class R</code>
Введение	2	<code>{</code>
Ссылки	3	<code>public:</code>
Классы	4	<code>int q;</code>
Поля и методы	5	<code>R() { q = 13; }</code>
Конструкторы	6	<code>};</code>
Создание и уничтожение объектов	7	
Копирование объектов	8	<code>class A: virtual R { public: void setq(int x); };</code>
Одиночное наследование	9	<code>class B: virtual R { public: int getq(); };</code>
Множественное наследование	10	<code>class C: public A, public B { };</code>
Обобщённое программирование	11	
Перегрузка операций	12	<code>void A::setq(int x) { q = x; }</code>
Исключения	13	<code>int B::getq() { return q; }</code>
	14	
	15	<code>int main(void)</code>
	16	<code>{</code>
	17	<code>C c;</code>
	18	<code>c.setq(666);</code>
	19	<code>cout << c.getq(); // Выведет 666</code>
	20	<code>return 0;</code>
	21	<code>}</code>

Базовые
сведения

Обобщённое
программирова-
ние

Объявление
шаблонов

Порождение
кода

Специализация
шаблонов

Перегрузка
операций

Исключения

В отличие от языка Java, обобщённые классы в C++ в строгом смысле отсутствуют. Вместо них в C++ используется развитый язык макроопределений, предназначенный для порождения кода во время компиляции программы. Ключевым элементом этого языка является понятие *шаблона*, который является рецептом для генерации кода класса или функции (метода).

Объявление шаблона начинается с ключевого слова `template`, за которым следует список формальных параметров шаблона:

```
template <формальные параметры>  
определение класса или функции
```

Тело шаблона представляет собой определение класса или функции, тем самым позволяя объявлять *шаблон класса* и *шаблон функции*.

Типовые формальные параметры шаблона

Базовые
сведения

Обобщённое
программирова-
ние

Объявление
шаблонов

Порождение
кода

Специализация
шаблонов

Перегрузка
операций

Исключения

Формальные параметры шаблона – это идентификаторы, областью видимости которых является тело шаблона. Они могут обозначать типы, значения или, в свою очередь, другие шаблоны. При применении шаблона формальные параметры в его теле заменяются на конкретные типы, значения и шаблоны, и полученный код компилируется.

Типовые параметры шаблона обозначают типы: синтаксически имя типового параметра в списке формальных параметров шаблона предваряется ключевым словом `typename` (или `class`).

Например, шаблон функции:

```
template <typename T>
void do_swap(T& a, T& b)
{
    T c = a;
    a = b;
    b = c;
}
```

Нетиповые формальные параметры шаблона

Базовые
сведения

Обобщённое
программирова-
ние

Объявление
шаблонов

Порождение
кода

Специализация
шаблонов

Перегрузка
операций

Исключения

Нетиповые параметры шаблона представляют значения, а не типы. Их синтаксис похож на объявление переменных. Внутри тела шаблона имя такого параметра обозначает константу указанного типа.

Например, шаблон для матриц:

```
template <typename T, int M, int N>
class Matrix
{
private:
    T a[M][N];
public:
    T& at(int i, int j);
};
```

```
template <typename T, int M, int N>
T& Matrix<T,M,N>::at(int i, int j)
{
    assert(i >= 0 && i < M && j >= 0 && j < N);
    return a[i][j];
}
```

Шаблонные формальные параметры шаблона

Базовые
сведения

Обобщённое
программирова-
ние

Объявление
шаблонов

Порождение
кода

Специализация
шаблонов

Перегрузка
операций

Исключения

Шаблонные параметры шаблона позволяют параметризовать шаблон другим шаблоном. Синтаксически такой параметр записывается как объявление шаблона без тела.

Например, умножение матриц:

```
template<template <typename , int , int> class M,  
        typename T, int P, int Q, int R>  
void multiply(M<T,P,R>& c, M<T,P,Q>& a, M<T,Q,R>& b)  
{  
    M<T,P,R> res;  
    for (int i = 0; i < P; i++) {  
        for (int j = 0; j < R; j++) {  
            res.at(i,j) = 0;  
            for (int k = 0; k < Q; k++) {  
                res.at(i,j) += a.at(i,k)*b.at(k,j);  
            }  
        }  
    }  
    c = res;  
}
```

Значения параметров шаблона по умолчанию

Базовые
сведения

Обобщённое
программирова-
ние

Объявление
шаблонов

Порождение
кода

Специализация
шаблонов

Перегрузка
операций

Исключения

Формальные параметры шаблона могут иметь значения по умолчанию. Синтаксически это оформляется путём добавления после имени параметра знака «=», за которым следует значение:

```
template <typename T = double, int M = 3, int N = 3>
class Matrix
{
private:
    T a[M][N];
public:
    T& at(int i, int j);
};
```

Если формальный параметр шаблона имеет значение по умолчанию, то все следующие за ним параметры также должны иметь значения по умолчанию.

Значения по умолчанию используются в случае, если при применении (инстанциации) шаблона соответствующие фактические параметры не указаны.

Зависимые идентификаторы в шаблоне

Базовые
сведения

Обобщённое
программирова-
ние

Объявление
шаблонов

Порождение
кода

Специализация
шаблонов

Перегрузка
операций

Исключения

Зависимый идентификатор – это любое имя внутри определения шаблона, которое зависит от формальных параметров шаблона. Смысл зависимых идентификаторов становится ясен компилятору только при инстанциации шаблона, когда известны его фактические параметры. Поэтому по умолчанию компилятор считает, что зависимые идентификаторы обозначают поля и методы. Чтобы указать, что некоторый зависимый идентификатор обозначает тип, перед ним нужно указывать ключевое слово `typename`.

```
struct Outer
{
    struct Inner { int x; };
};

template <typename T> class Sample
{
    T::Inner x;           // Ошибка!
    typename T::Inner y;  // Правильное объявление
};
```

Инстанциация шаблона – это порождение кода по шаблону и списку фактических параметров. Инстанциация осуществляется при первом использовании конструкции

```
имя_шаблона <список фактических параметров >
```

Например, в момент, когда компилятор обрабатывает объявление переменной

```
Matrix<int, 4, 5> m;
```

он порождает код, получаемый путём подстановки в тело шаблона `Matrix<typename, int, int>` фактических параметров `int`, 4 и 5.

Отметим, что если последний фактический параметр шаблона – типовый, и к тому же представляет собой инстанциацию другого шаблона, то нужно ставить пробел между заключительными угловыми скобками. Например,

```
do_swap <Matrix<int, 4, 5> >(m, m2);
```


Выведение фактических параметров при инстанциации шаблона функции

Базовые
сведения

Обобщённое
программирова-
ние

Объявление
шаблонов

Порождение
кода

Специализация
шаблонов

Перегрузка
операций

Исключения

Если параметры шаблона функции используются в списке формальных параметров этой функции, то компилятор может вывести значения фактических параметров шаблона самостоятельно.

Например, рассмотрим следующей фрагмент кода:

```
Matrix<int, 3, 1> a;  
Matrix<int, 1, 3> b;  
Matrix<int, 3, 3> c;  
a.at(0, 0) = 1;  
a.at(1, 0) = 2;  
a.at(2, 0) = 3;  
b.at(0, 0) = 1;  
b.at(0, 1) = 2;  
b.at(0, 2) = 3;  
multiply<Matrix, int, 3, 1, 3>(c, a, b);
```

Автоматическое выведение фактических параметров шаблона функции позволяет упростить вызов функции `multiply`:

```
multiply(c, a, b);
```

Требования к фактическим параметрам шаблонов

Базовые сведения

Обобщённое программирование

Объявление шаблонов

Порождение кода

Специализация шаблонов

Перегрузка операций

Исключения

Тело шаблона накладывает требования на его фактические параметры. Например, в теле шаблона функции `multiply` подразумевается, что значения типа `M` – это объекты, имеющие метод `at`, который возвращает ссылку на значение, к которому могут быть применены операции сложения и умножения.

Если фактические параметры шаблона не удовлетворяют требованиям, инстанциация шаблона с этими фактическими параметрами приведёт к ошибке времени компиляции.

Например,

```
template <typename T, int M, int N>
struct Array2D { T a[M][N]; };
...
Array2D<int, 3, 1> x;
Array2D<int, 1, 3> y;
Array2D<int, 3, 3> z;
multiply(z, x, y); // error: struct Array2D<int, 3, 3>
                  // has no member named 'at'
```

Специализация шаблона функции

Базовые
сведения

Обобщённое
программирова-
ние

Объявление
шаблонов

Порождение
кода

Специализация
шаблонов

Перегрузка
операций

Исключения

Специализация шаблона – это разработка отдельной версии порождаемого шаблоном кода для конкретного набора фактических параметров.

Пусть имеется шаблон функции вида

```
template <формальные_параметры >  
тип имя (...)  
{ ... }
```

Специализированная версия функции для конкретного набора фактических параметров записывается как

```
template <>  
тип имя<фактические_параметры> (...)  
{ ... }
```

Следует иметь в виду, что специализация шаблона функции не даёт особенных преимуществ над обычной перегрузкой функций.

Пример: специализация шаблона функции

Базовые сведения	1 <code>template <typename T> void do_swap(T& a, T& b)</code>
	2 <code>{</code>
Обобщённое программирование	3 <code> T c = a;</code>
	4 <code> a = b;</code>
Объявление шаблонов	5 <code> b = c;</code>
Порождение кода	6 <code>}</code>
	7
Специализация шаблонов	8 <code>template <> void do_swap<int>(int& a, int& b)</code>
	9 <code>{</code>
Перегрузка операций	10 <code> a += b;</code>
	11 <code> b = a-b;</code>
Исключения	12 <code> a = a-b;</code>
	13 <code>}</code>
	14
	15 <code>int main()</code>
	16 <code>{</code>
	17 <code> int x = 10, y = 20;</code>
	18 <code> do_swap(x, y); // будет вызвана функция</code>
	19 <code> // в строчках 8..13</code>
	20 <code> return 0;</code>
	21 <code>}</code>

Пример: перегрузка имеет больший приоритет, чем специализация

Базовые сведения

Обобщённое программирование

Объявление шаблонов

Порождение кода

Специализация шаблонов

Перегрузка операций

Исключения

```
1  template <typename T>
2  T t_max(T a, T b)
3  {
4      cout << "1_";  return a > b ? a : b;
5  }
6
7  template <>
8  int t_max<int>(int a, int b)
9  {
10     cout << "2_";  return a > b ? a : b;
11 }
12
13 int t_max(int a, int b) {
14     cout << "3_";  return a > b ? a : b;
15 }
16
17 int main()
18 {
19     int x = 10, y = 20;
20     cout << t_max(x,y); // Вывод: 3 20
21     return 0;
22 }
```

Специализация шаблона класса

Базовые
сведения

Обобщённое
программирова-
ние

Объявление
шаблонов

Порождение
кода

Специализация
шаблонов

Перегрузка
операций

Исключения

Специализированные версии шаблонов классов создаются аналогично специализации шаблонов функций.

Например, рассмотрим шаблон класса, представляющего кортежи фиксированной длины:

```
1  template <typename T, size_t N>
2  class Tuple
3  {
4  private:
5      T a[N];
6  public:
7      T get(int i);
8      void set(int i, T x);
9  };
10
11  template <typename T, size_t N>
12  T Tuple<T,N>::get(int i) { return a[i]; }
13
14  template <typename T, size_t N>
15  void Tuple<T,N>::set(int i, T x) { a[i] = x; }
```

Пример: специализация шаблона класса

Базовые сведения	17	<code>template <></code>
Обобщённое программирование	18	<code>class Tuple<bool, 8></code>
	19	<code>{</code>
Объявление шаблонов	20	<code>private:</code>
Порождение кода	21	<code> unsigned char a;</code>
Специализация шаблонов	22	<code>public:</code>
	23	<code> bool get(int i);</code>
Перегрузка операций	24	<code> void set(int i, bool x);</code>
	25	<code>};</code>
Исключения	26	
	27	<code>bool Tuple<bool,8>::get(int i)</code>
	28	<code>{</code>
	29	<code> return (a >> i) & 1;</code>
	30	<code>}</code>
	31	
	32	<code>void Tuple<bool,8>::set(int i, bool x)</code>
	33	<code>{</code>
	34	<code> a = (unsigned char)x << i;</code>
	35	<code>}</code>

Частичная специализация шаблона класса

При частичной специализации фиксируется только часть фактических параметров шаблона. Например,

```
37  template <typename T>
38  class Tuple<T,1>
39  {
40  private:
41      T a;
42  public:
43      T get(int i);
44      void set(int i, T x);
45  };
46
47  template <typename T>
48  T Tuple<T,1>::get(int i) { return a; }
49
50  template <typename T>
51  void Tuple<T,1>::set(int i, T x) { a = x; }
```


Вычисления во время компиляции

Базовые
сведения

Обобщённое
программирова-
ние

Объявление
шаблонов

Порождение
кода

Специализация
шаблонов

Перегрузка
операций

Исключения

Специализация шаблонов позволяет заставить компилятор выполнить некоторые вычисления во время компиляции. Например, можно попросить компилятор вычислить факториал числа:

```
5  template <int N>
6  struct Fact {
7      static const unsigned long val =
8      N * Fact<N-1>::val;
9  };
10
11  template <>
12  struct Fact<0> {
13      static const unsigned long val = 1;
14  };
15
16  int main()
17  {
18      cout << Fact<3>::val << '\n';
19      return 0;
20  }
```

Пример: возведение числа в степень

Базовые сведения	5	template <typename T, int N>
	6	struct Power {
Обобщённое программирование	7	static T eval(T x) {
	8	if (N % 2 == 0) {
	9	return Power<T,N/2>::eval(x*x);
Объявление шаблонов	10	}
Порождение кода	11	return x * Power<T,N-1>::eval(x);
Специализация шаблонов	12	}
	13	};
Перегрузка операций	14	
	15	template <typename T>
Исключения	16	struct Power<T,0> {
	17	static T eval(T x) { return (T)1; }
	18	};
	19	
	20	int main()
	21	{
	22	int x;
	23	cin >> x;
	24	cout << Power<int,10>::eval(x) << '\n';
	25	return 0;
	26	}

Понятие перегрузки операций

Базовые сведения

Обобщённое программирование

Перегрузка операций

Введение

Присваивание

Бинарные операции

Унарные операции

Операция ()

Исключения

Вообще, de facto практически в любом языке программирования операции перегружены. Например, в языке Pascal смысл операции «+» меняется в зависимости от типов операндов вплоть до того, что «сложение» строк означает их конкатенацию.

Язык C++ позволяет распространить принцип перегрузки на пользовательские типы данных, т.е. на классы. При этом, в отличие от языков, в которых пользователь может сам составлять знаки операций и задавать их приоритет и ассоциативность (например, ML или Haskell), ассортимент операций и их характеристики в C++ жёстко заданы.

Технически перегрузка операций в C++ реализована через определение функций со специальными именами вида «operator знак_операции», осуществляющих выполнение операций.

Ассортимент перегружаемых операций

Базовые сведения

Обобщённое программирование

Перегрузка операций

Введение

Присваивание

Бинарные операции

Унарные операции

Операция ()

Исключения

В языке C++ допускается перегрузка следующего набора операций:

```
// унарные :  
++ -- - + ! ~ * & (тип)  
// бинарные :  
= += -= *= /= %= &= |= ^| <<= >>=  
+ - * / % & | ^ << >>  
== != > < >= <=  
&& ||  
[] -> ->* () ,
```

Кроме того, допускается перегружать операции «new» и «delete».

Перегруженные операции имеют те же приоритеты и ассоциативность, что и соответствующие операции над базовыми типами языка C++.

Нельзя перегружать операции «.», «.*», «? :», «sizeof», «dynamic_cast» и некоторые другие.

Перегрузка операции присваивания

Базовые сведения

Обобщённое программирование

Перегрузка операций

Введение

Присваивание

Бинарные операции

Унарные операции

Операция ()

Исключения

Операция присваивания для некоторого класса `A` перегружается путём объявления в классе метода, прототип которого выглядит как

```
A& operator= (const A &obj);
```

Схема работы перегруженной операции присваивания:

1. проверить, не присваивается ли объект сам себе (если так, то перейти к пункту 5);
2. выделить память, в которую будет скопировано содержимое `obj`;
3. освободить память, используемую внутри объекта;
4. выполнить копирование данных;
5. вернуть `*this`.

Вызов перегруженной операции присваивания можно выполнять двумя способами:

```
a = b;                // 1-ый способ  
a.operator=(b);       // 2-ой способ
```

Операция присваивания возвращает левое значение

Базовые сведения

Обобщённое программирование

Перегрузка операций

Введение

Присваивание

Бинарные операции

Унарные операции

Операция ()

Исключения

Форма перегрузки операции присваивания в C++ наводит на мысль, что она возвращает *левое значение*, т.е. результату, возвращаемому операцией присваивания, можно чего-то присвоить:

```
A x, y, z;  
(x = y) = z;
```

Этот код действительно работает. Более того, в отличие от языка C, язык C++ допускает такое использование операции присваивания и для встроенных типов данных. Например,

```
int a = 2, b = 3, c = 4;  
(a = b) = c;
```

Поэтому не стоит пытаться запретить такое поведение путём, например, возвращения константой ссылки:

```
const A& operator= (const A &obj);
```

Составные операции присваивания

Базовые
сведения

Обобщённое
программирова-
ние

Перегрузка
операций

Введение

Присваивание

Бинарные
операции

Унарные опе-
рации

Операция ()

Исключения

Составные операции присваивания перегружаются по той же схеме, что и обычная операция присваивания:

```
A& operator+= (const A &obj);
```

Как правило, в случае сложного внутреннего состояния объекта реализация перегруженной составной операции присваивания также должна предусматривать возможность присваивания объекта самому себе.

Составные операции присваивания также возвращают левое значение, обеспечивая работоспособность конструкций вида

```
(x += y) *= z;
```

Перегрузка бинарных арифметических операций

Базовые
сведения

Обобщённое
программирова-
ние

Перегрузка
операций

Введение

Присваивание

Бинарные
операции

Унарные опе-
рации

Операция ()

Исключения

Прототип перегруженной бинарной арифметической операции может выглядеть как

```
const A& operator+ (const A &other) const;
```

Отметим, что ключевое слово `const` в конце прототипа метода означает, что метод не меняет внутреннее состояние объекта, для которого он вызван. Если объект – константный, то к нему применимы только константные методы!

Удобно реализовать бинарные арифметические операции через уже реализованные операции составного присваивания:

```
const A& A::operator+ (const A &other) const {  
    return A(*this) += other;  
}
```

А вот так делать не надо (лишний вызов конструктора копий + вызов деструктора):

```
const A A::operator+ (const A &other) const {  
    A result = *this;  result += other;  
    return result;  
}
```


Перегрузка операций сравнения

Базовые сведения

Обобщённое программирование

Перегрузка операций

Введение

Присваивание

Бинарные операции

Унарные операции

Операция ()

Исключения

Операции сравнения при перегрузке имеют следующие прототипы:

```
bool operator== (const A &other) const;  
bool operator!= (const A &other) const;  
bool operator<  (const A &other) const;  
bool operator>  (const A &other) const;  
bool operator<= (const A &other) const;  
bool operator>= (const A &other) const;
```

При этом удобно при реализации одних операций сравнения использовать уже реализованные другие операции. Например, реализовав операцию «==», мы можем использовать её отрицание в коде операции «!=»:

```
const bool A::operator!= (const A &other) const {  
    return !(*this == other);  
}
```

Этот приём, в частности, автоматически обеспечивает непротиворечивость реализации операций сравнения.

Перегрузка операций с помощью функций

Базовые сведения

Обобщённое программирование

Перегрузка операций

Введение

Присваивание

Бинарные операции

Унарные операции

Операция ()

Исключения

Перегруженные операции могут иметь операнды разных типов. Это полезно, например, при реализации операции умножения матрицы на скалярное значение:

```
const Matrix& operator* (int k) const;
```

Однако, использование такой операции возможно только в случае, когда матрица является первым операндом. Чтобы справиться с этой проблемой, в C++ разрешена перегрузка операций с помощью функций. Например, в нашем случае функция умножения скалярного значения на матрицу может выглядеть как

```
const Matrix& operator* (int k, const Matrix &m) {  
    return m * k;  
}
```

Следует иметь в виду, что хотя бы один операнд функции, осуществляющей перегрузку операций, должен быть объектом класса. Это гарантирует невозможность перегрузки операций над встроенными типами.

Перегрузка операций «&&» и «||»

Базовые
сведения

Обобщённое
программирова-
ние

Перегрузка
операций

Введение

Присваивание

Бинарные
операции

Унарные опе-
рации

Операция ()

Исключения

Операции «&&» и «||» могут быть перегружены точно так же, как и другие бинарные арифметические операции:

```
const A& operator&& (const A &other) const;  
const A& operator|| (const A &other) const;
```

Однако, перегрузка операций «&&» и «||» не рекомендуется. Дело в том, что эти операции применительно к встроенным типам данных вычисляют свой второй операнд не всегда. Эта их особенность широко используется в программировании, т.е. любой программист ожидает, что они будут работать именно по этой схеме.

Перегруженные версии этих операций всегда будут вычислять оба операнда. Тем самым, их использование будет контринтуитивно, что обязательно приведёт к ошибкам в программах.

Перегрузка операции «[]»

Базовые
сведения

Обобщённое
программирова-
ние

Перегрузка
операций

Введение

Присваивание

Бинарные
операции

Унарные опе-
рации

Операция ()

Исключения

Операция «[]» перегружается следующим образом:

```
тип1& operator [] (тип2 index);
```

Операция возвращает ссылку, что даёт возможность использовать её результат в левой части операции присваивания. К сожалению, таким образом перегруженную операцию невозможно применять к константным объектам. Поэтому принято определять ещё и специальную версию операции «[]» для константных объектов:

```
const тип1& operator [] (тип2 index) const;
```

Естественно, в этом случае необязательно, чтобы операция возвращала ссылку. Поэтому, если «тип1» допускает дешёвое копирование, от ссылки можно избавиться. Например,

```
double operator [] (int index) const;
```

Отметим, что операция «[]» не может быть перегружена с помощью функции.

Перегрузка унарных операций

Базовые сведения

Обобщённое программирование

Перегрузка операций

Введение

Присваивание

Бинарные операции

Унарные операции

Операция ()

Исключения

Унарные операции перегружаются по одной из приведённых схем (далее «@» – некоторый знак операции):

```
class A
{
    ...
    тип operator@ ();    // с помощью метода
    ...
};
тип operator@ (A &obj); // с помощью функции
```

Исключение составляют постфиксные ++ и --. Чтобы их отличать от префиксных инкремента и декремента, они имеют дополнительный фиктивный параметр типа `int`:

```
class A
{
    ...
    A& operator++ (int);
    ...
};
A& operator++ (A &obj, int);
```

Перегрузка операций «*» и «->»

Базовые сведения

Обобщённое программирование

Перегрузка операций

Введение

Присваивание

Бинарные операции

Унарные операции

Операция ()

Исключения

Перегрузка операций «*» и «->» позволяет использовать объекты класса так, как если бы они были указателями:

```
class A
{
    ...
    тип& operator* ();
    тип operator-> ();
    ...
};
тип& operator* (A &a);
```

Особенностью операции «->» является тот факт, что она последовательно применяется к своему же возвращаемому значению до тех пор, пока не получится указатель. Т.е., например, если в классе А операция «->» возвращает ссылку на объект класса В, а в классе В операция «->» возвращает указатель на объект класса С, то применение «->» к переменной *x* типа А открывает доступ к полям и методам объекта класса С.

Пример: перегрузка операции «->»

Базовые сведения	1	<code>struct C { int field; };</code>
	2	
Обобщённое программирование	3	<code>struct B {</code>
	4	<code> C c;</code>
	5	<code> C* operator-> ();</code>
Перегрузка операций	6	<code>};</code>
Введение	7	
Присваивание	8	<code>C* B::operator-> () { return &c; }</code>
Бинарные операции	9	
Унарные операции	10	<code>struct A {</code>
	11	<code> B b;</code>
Операция ()	12	<code> B& operator-> ();</code>
Исключения	13	<code>};</code>
	14	
	15	<code>B& A::operator-> () { return b; }</code>
	16	
	17	<code>int main()</code>
	18	<code>{</code>
	19	<code> A x;</code>
	20	<code> x->field = 666;</code>
	21	<code> return 0;</code>
	22	<code>}</code>

Перегрузка операции «&»

Базовые сведения

Обобщённое программирование

Перегрузка операций

Введение

Присваивание

Бинарные операции

Унарные операции

Операция ()

Исключения

Перегрузка операции «&» позволяет вместо адреса объекта возвращать всё, что угодно (например, «умный» указатель, т.е. объект с перегруженными операциями «*» и «->», осуществляющий подсчёт ссылок на объект и автоматическое его удаление при уничтожении последней ссылки):

```
class A
{
    ...
    тип operator& ();
    ...
};
тип operator& (A &a);
```

Впрочем, перегрузка этой операции не рекомендуется, т.к. может привести к трудноуловимым ошибкам в коде. Вместо её перегрузки рекомендуется определить метод, который делает то же самое, но не экранирует стандартную реализацию «&»:

```
тип address_of ();
```


Перегрузка операции приведения типа

Базовые сведения

Обобщённое программирование

Перегрузка операций

Введение

Присваивание

Бинарные операции

Унарные операции

Операция ()

Исключения

Перегрузка операции приведения объекта к некоторому типу осуществляется путём объявления метода

```
operator тип() const;
```

Например,

```
struct A {  
    int x;  
    operator int() const;  
};  
  
A::operator int() const { return x; }
```

Обратите внимание на то, что, хотя операция приведения типа возвращает значение, в её объявлении тип возвращаемого значения не указывается. В этом она напоминает конструктор.

Перегрузка операции «()»

Базовые
сведения

Обобщённое
программирова-
ние

Перегрузка
операций

Введение

Присваивание

Бинарные
операции

Унарные опе-
рации

Операция ()

Исключения

Перегрузка операции () даёт возможность «вызывать» объекты класса, как если бы они были функциями. Прототип перегруженной операции () выглядит как

тип `operator()`(список формальных параметров);

Можно сказать, что операция () – n -арная, потому что внутри скобок может располагаться произвольное количество формальных параметров.

Естественно, в классе можно определить несколько перегруженных операций (), если они будут различаться сигнатурами.

Базовые
сведения

Обобщённое
программирова-
ние

Перегрузка
операций

Исключения

Порождение

Перехват и
обработка

Спецификация

Класс exception

В отличие от языка Java, в качестве исключения в C++ может выступать любое значение (объект, строка, значение базового типа и т.п.).

Для порождения исключения предназначен оператор `throw`, который может вызываться как с указанием значения, описывающего исключительную ситуацию, так и без его указания:

```
throw значение;  
throw; // вариант без указания значения
```

Механизм исключений в C++ не предусматривает запоминание информации о стеке вызовов в момент порождения исключения. Поэтому сообщение, выводимое при аварийном завершении программы по причине необработанного исключения, менее информативно.

Пример: оператор throw

Базовые
сведения

Обобщённое
программирова-
ние

Перегрузка
операций

Исключения

Порождение

Перехват и
обработка

Спецификация

Класс exception

```
1  int main()  
2  {  
3      throw;  
4      return 0;  
5  }
```

Вывод:

```
terminate called without an active exception
```

```
1  int f(int x) { return 2*x; }  
2  int main()  
3  {  
4      throw f;  
5      return 0;  
6  }
```

Вывод:

```
terminate called after throwing an instance  
of 'int (*)(int)'
```

Синтаксис try- и catch-блоков

Базовые
сведения

Обобщённое
программирова-
ние

Перегрузка
операций

Исключения

Порождение

Перехват и
обработка

Спецификация

Класс exception

Как и в языке Java, участки кода, в которых ожидается возникновение исключительной ситуации, обрамляются try-блоками. С try-блоками связаны один или несколько catch-блоков, осуществляющих перехват исключений по типу:

```
try {  
    /* ... */  
}  
catch (тип_исключения1 переменная) {  
    /* ... */  
}  
catch (тип_исключения2 переменная) {  
    /* ... */  
}
```

Catch-блок, который может перехватить любое исключение, записывается как

```
catch (...) {  
    /* ... */  
}
```

Пример: перехват исключения `std::bad_alloc`

Базовые сведения

Обобщённое программирование

Перегрузка операций

Исключения

Порождение

Перехват и обработка

Спецификация

Класс exception

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      try {
7          size_t n;
8          cin >> n;
9          int *a = new int[n];
10         cout << "success\n";
11         delete [] a;
12     } catch (bad_alloc) {
13         cout << "out_of_memory\n";
14     }
15     return 0;
16 }
```

Исключение `std::bad_alloc` порождается операцией `new` при невозможности выделить блок динамической памяти.

Жизненный цикл объектов-исключений

Базовые
сведения

Обобщённое
программирова-
ние

Перегрузка
операций

Исключения

Порождение

Перехват и
обработка

Спецификация

Класс exception

Исследуем жизненный цикл объектов исключений с помощью класса A, который протоколирует факты создания, копирования и уничтожения своих экземпляров:

```
struct A {  
    bool is_copy;  
    A();  
    A(const A &obj);  
    ~A();  
    void report();  
};
```

```
A::A(): is_copy(false)  
{ cout << "new" << '\n'; }
```

```
A::A(const A &obj): is_copy(true)  
{ cout << "copy" << '\n'; }
```

```
A::~~A() { cout << "destroy_" << is_copy << '\n'; }
```

```
void A::report() { cout << "report\n"; }
```

Пример 1: жизненный цикл объектов-исключений

Базовые сведения

Обобщённое программирование

Перегрузка операций

Исключения

Порождение

Перехват и обработка

Спецификация

Класс exception

Порождение объекта-исключения в куче и перехват, при котором он передаётся в catch-блок по указателю. В конце catch-блока необходимо явное уничтожение объекта.

```
int main()
{
    try {
        throw new A();
    } catch (A *a) {
        a->report();
        delete a;
    }
    return 0;
}
```

Вывод:

```
new
report
destroy 0
```


Пример 2: жизненный цикл объектов-исключений

Базовые сведения

Обобщённое программирование

Перегрузка операций

Исключения

Порождение

Перехват и обработка

Спецификация

Класс exception

Порождение временного объекта-исключения и перехват, при котором создаётся его копия.

```
int main()
{
    try {
        throw A();
    } catch (A a) {
        a.report();
    }
    return 0;
}
```

Вывод:

```
new
copy
report
destroy 1
destroy 0
```

Пример 3: жизненный цикл объектов-исключений

Базовые сведения

Обобщённое программирование

Перегрузка операций

Исключения

Порождение

Перехват и обработка

Спецификация

Класс exception

Порождение временного объекта-исключения и перехват, при котором он передаётся в catch-блок по ссылке, т.е. без создания копии.

```
int main()
{
    try {
        throw A();
    } catch (A &a) {
        a.report();
    }
    return 0;
}
```

Вывод:

```
new
report
destroy 0
```

Пример: уничтожение автоматических объектов в процессе передачи исключения в catch-блок

Базовые сведения

Обобщённое программирование

Перегрузка операций

Исключения

Порождение

Перехват и обработка

Спецификация

Класс exception

```
1  #include <iostream>
2
3  struct Dummy { ~Dummy(); };
4
5  Dummy::~~Dummy() { std::cout << "#_"; }
6
7  void recur(int n) {
8      if (n == 0) throw "base_of_recursion";
9      Dummy d;
10     recur(n-1);
11     std::cout << "should_never_be_reached!";
12 }
13
14 int main()
15 {
16     try { recur(10); } catch (...) { }
17     return 0;
18 }
```

Вывод:

```
# # # # # # # # # #
```

Перехват исключений в инициализаторах конструкторов

Базовые сведения

Обобщённое программирование

Перегрузка операций

Исключения

Порождение

Перехват и обработка

Спецификация

Класс exception

Напомним, что конструкторы в C++ могут иметь списки инициализаторов, в которых вызываются конструкторы объектов, расположенных в полях, и конструкторы базовых классов.

Так как конструкторы полей и базовых классов могут порождать исключения, нужно уметь эти исключения перехватывать, например, для протоколирования неудачного создания объектов в логах. Для этой цели служат специальные try-блоки, обрамляющие тело конструктора, которые синтаксически выглядят как

```
Класс :: Класс (параметры_конструктора) try
    : список_инициализации
{ тело_конструктора }
catch (исключение) { тело_обработчика_исключения }
...
```

Отметим, что если catch-блок не оканчивается порождением другого исключения, старое исключение не уничтожается и передаётся дальше.

Пример: перехват исключений в инициализаторах конструкторов

Базовые сведения
Обобщённое программирование
Перегрузка операций
Исключения
Порождение
Перехват и обработка
Спецификация
Класс exception

```
1  #include <iostream>
2  using namespace std;
3
4  struct A { A(int x); };
5  struct B: public A { B(int x); };
6
7  A::A(int x) { if (x < 0) throw "negative"; }
8
9  B::B(int x) try: A(x) { }
10 catch (const char *s) { cout << "1_"; }
11
12 int main()
13 {
14     try {
15         B b(-10);
16     } catch (...) {
17         cout << "2\n";
18     }
19     return 0;
20 }
```

Спецификатор `throw` в заголовках функций

Базовые
сведения

Обобщённое
программирова-
ние

Перегрузка
операций

Исключения

Порождение

Перехват и
обработка

Спецификация

Класс `exception`

Как и в языке Java, в C++ можно перечислить типы исключений, которые может породить вызов функции. Для этого после списка формальных параметров следует разместить спецификатор `throw`:

`throw` (список типов исключений)

Семантика этого спецификатора несколько отличается от семантики соответствующей конструкции языка Java, а именно:

- вызов функции, для которой разрешённые исключения не указаны, может породить любые исключения;
- вызов функции со спецификатором `throw()` не порождает никаких исключений;
- компилятор не следит за тем, какие исключения порождаются внутри функции: вместо этого неразрешённые исключения просто не могут выйти за пределы функции и приводят к аварийному завершению программы.

Пример 1: спецификатор throw

Базовые сведения

Обобщённое программирование

Перегрузка операций

Исключения

Порождение

Перехват и обработка

Спецификация

Класс exception

```
1  #include <iostream>
2  using namespace std;
3
4  void f() throw(int)
5  {
6      throw 2;
7  }
8
9  int main()
10 {
11     try {
12         f();
13     } catch (int e) {
14         cout << "int:_" << e << '\n';
15     }
16     return 0;
17 }
```

Вывод (исключение 2 «пропущено» наружу из функции f):

```
int: 2
```

Пример 2: спецификатор throw

Базовые сведения	1	<code>#include <iostream></code>
	2	<code>using namespace std;</code>
Обобщённое программирование	3	
	4	<code>void f() throw()</code>
	5	<code>{</code>
Перегрузка операций	6	<code> throw 2;</code>
	7	<code>}</code>
Исключения	8	
Порождение	9	<code>int main()</code>
Перехват и обработка	10	<code>{</code>
Спецификация	11	<code> try {</code>
Класс exception	12	<code> f();</code>
	13	<code> } catch (int e) {</code>
	14	<code> cout << "int:_" << e << '\n';</code>
	15	<code> }</code>
	16	<code> return 0;</code>
	17	<code>}</code>

Вывод (исключение 2 «не выпущено» из функции `f`):

```
terminate called after throwing an instance
of 'int'
```


Спецификатор `throw` и переопределение виртуальных методов

Базовые сведения

Обобщённое программирование

Перегрузка операций

Исключения

Порождение

Перехват и обработка

Спецификация

Класс exception

По очевидным соображениям, связанным с поддержкой полиморфизма, переопределение виртуальных методов не должно ослаблять ограничения, накладываемые спецификатором `throw`, т.е. переопределённому методу не разрешается породить исключения, не порождаемые методом базового класса.

```
struct A {  
    virtual void f() throw(int) {}  
};  
  
struct B: public A {  
    void f() throw(float) {}  
};
```

Сообщение об ошибке:

```
error: looser throw specifier for  
'virtual void B::f() throw(float)'  
overriding 'virtual void A::f() throw(int)'
```

Создание классов исключений

Базовые сведения

Обобщённое программирование

Перегрузка операций

Исключения

Порождение

Перехват и обработка

Спецификация

Класс exception

Рекомендуемым способом создания классов исключений в языке C++ является наследование их от библиотечного класса `exception` (его объявление расположено в заголовочном файле `<exception>`):

```
class exception {
public:
    exception () throw();
    exception (const exception&) throw();
    exception& operator= (const exception&) throw();
    virtual ~exception() throw();
    virtual const char* what() const throw();
};
```

В производном от `exception` классе нужно переопределить виртуальный метод `what`, возвращающий ASCIIZ-строку с описанием исключительной ситуации.

Пример: создание классов исключений

Базовые сведения
Обобщённое программирование
Перегрузка операций
Исключения
Порождение
Перехват и обработка
Спецификация
Класс exception

```
1  #include <exception>
2  using namespace std;
3
4  struct MyException: public exception
5  {
6      const char *what() const throw()
7      { return "This is my exception"; }
8  };
9
10 int main()
11 {
12     throw MyException();
13     return 0;
14 }
```

Вывод:

```
terminate called after throwing an instance
of 'MyException'
  what():  This is my exception
Aborted
```