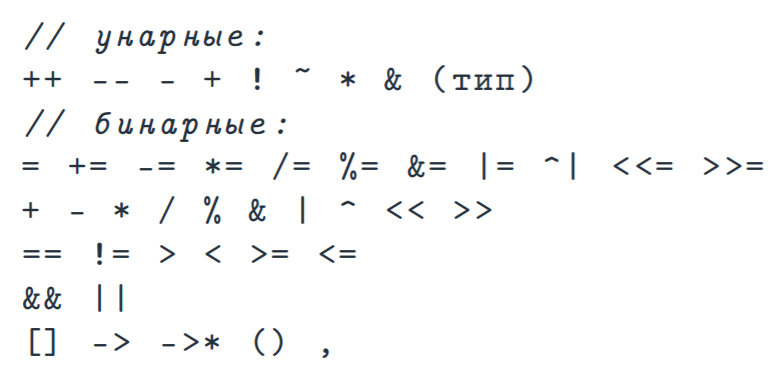
# Общая информация

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.

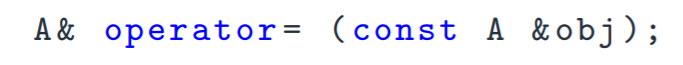
В языке C++ допускается перегрузка следующего набора операций:



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.

# Операция присваивания

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

Схема работы перегруженной операции присваивания:

1. проверить, не присваивается ли объект сам себе (если так, то перейти к пункту 5);

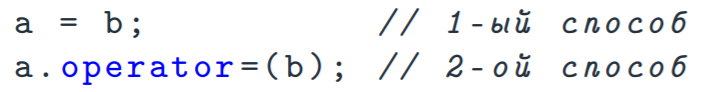
2. выделить память, в которую будет скопировано содержимое obj;

3. освободить память, используемую внутри объекта;

4. выполнить копирование данных;

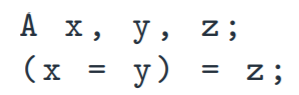
5. вернуть \*this.

Вызов перегруженной операции присваивания можно выполнять двумя способами:

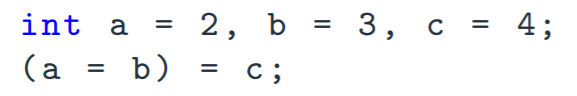


(Второй способ для извращенцев, ИМХО)

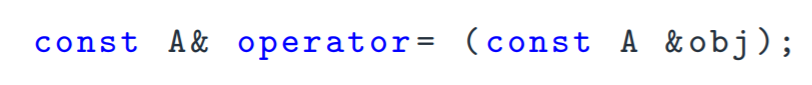
Форма перегрузки операции присваивания в C++ наводит на мысль, что она возвращает левое значение, т. е. результату, возвращаемому операцией присваивания, можно чего-то присвоить:



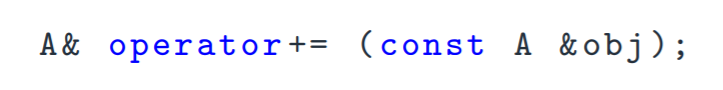
Этот код действительно работает. **Более того, в отличие от языка C, язык C++ допускает** такое использование операции присваивания и для встроенных типов данных. Например



Поэтому не стоит пытаться запретить такое поведение путём, например, возвращения константой ссылки(Типо, вы можете так сделать, но только если у вас есть на то причина. Иначе вы извращенец):



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



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

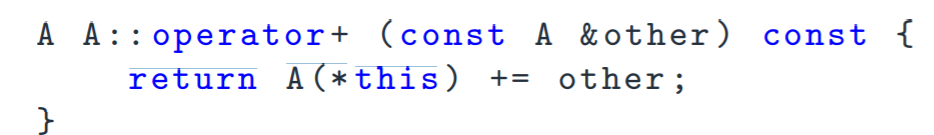


# Перегрузка арифметических операторов

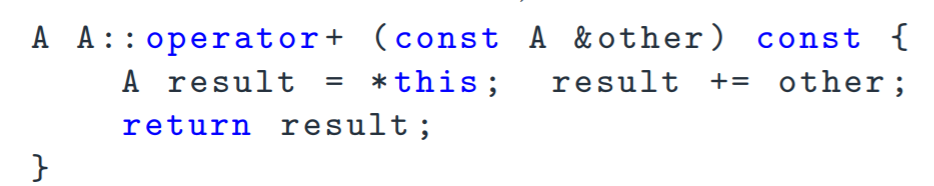
Прототип перегруженной бинарной арифметической операции может выглядеть как



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:

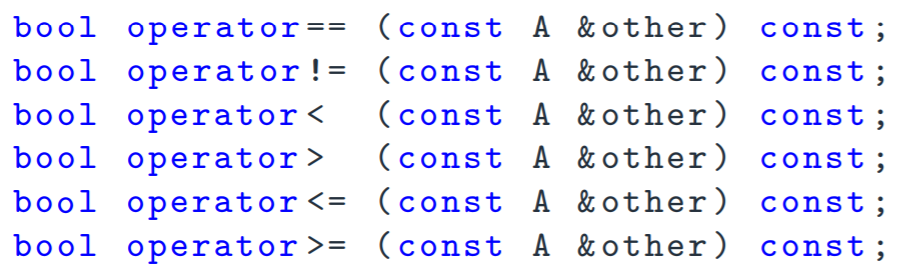


**А вот так делать не надо** (лишний вызов конструктора копий + вызов деструктора):

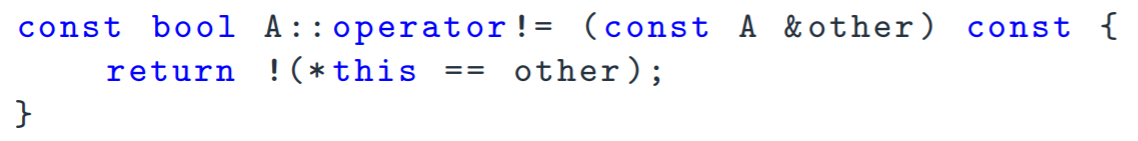


# Перегрузка операторов сравнения

Тут в целом все то же самое, что и с другими бинарными операторами, только они операторы сравнения возвращают bool. Операции сравнения при перегрузке имеют следующие прототипы:



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



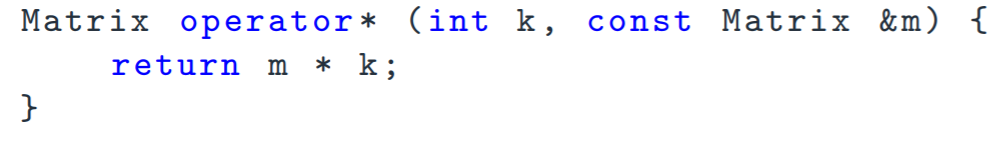
Этот приём, в частности, автоматически обеспечивает непротиворечивость реализации операций сравнения.

# Перегрузка операторов с помощью функций

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



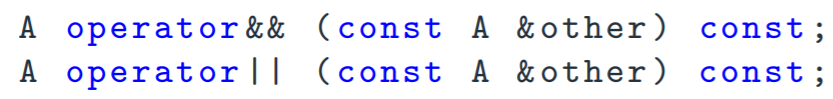
Однако, использование такой операции возможно только в случае, когда матрица является первым операндом. Чтобы справиться с этой проблемой, в C++ разрешена перегрузка операций с помощью функций. Например, в нашем случае функция умножения скалярного значения на матрицу может выглядеть как



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 операторов «&&» и «||»

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



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 «[ ]»

Операция «[ ]» перегружается следующим образом:



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:



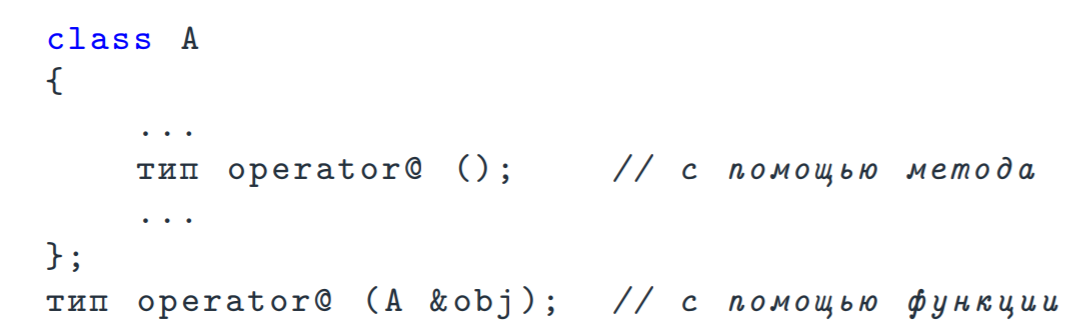
Естественно, в этом случае необязательно, чтобы операция возвращала ссылку. Поэтому, если «тип1» допускает дешёвое копирование, от ссылки можно избавиться. Например,



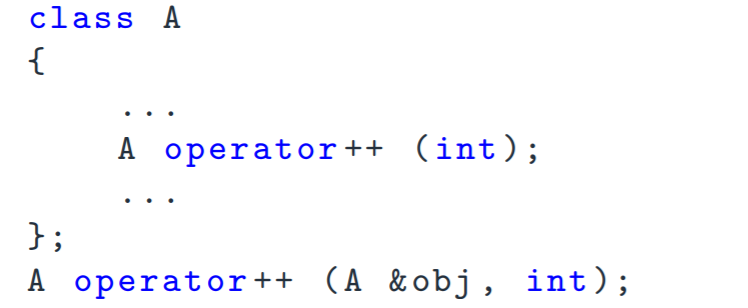
Отметим, что операция «[ ]» не может быть перегружена с помощью функции.

# Перегрузка унарных операторов

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

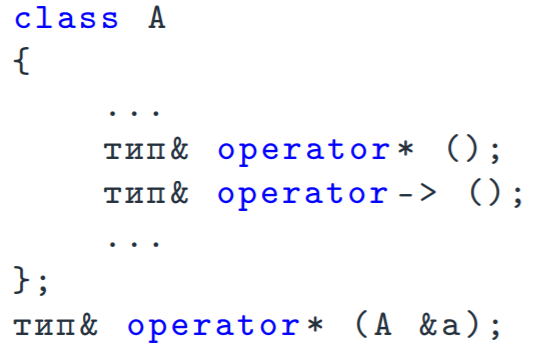


Исключение составляют постфиксные ++ и --. Чтобы их отличать от префиксных инкремента и декремента, они имеют дополнительный фиктивный параметр типа int:

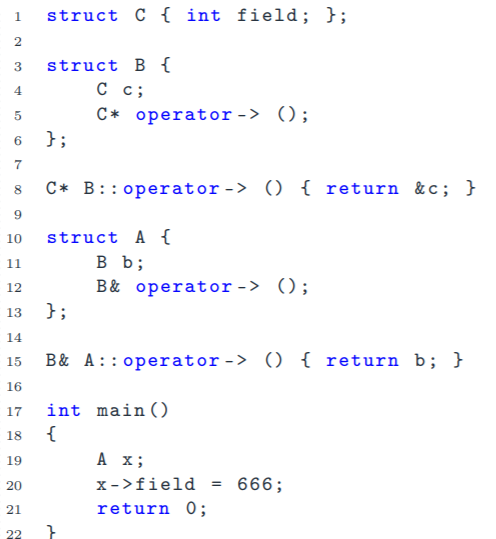


# Перегрузка операций «\*» и «->»

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

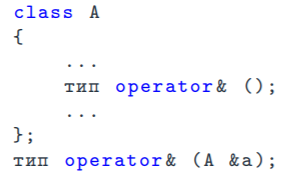


Особенностью операции «->» является тот факт, что она последовательно применяется к своему же возвращаемому значению до тех пор, пока не получится указатель. Т.е., например, если в классе A операция «->» возвращает ссылку на объект класса B, а в классе B операция «->» возвращает указатель на объект класса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



# Перегрузка операции «&»

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):



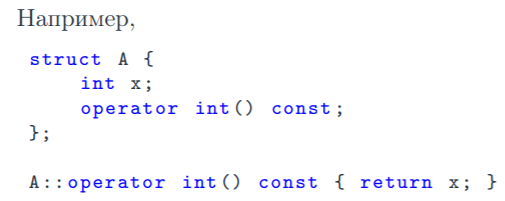
Впрочем, перегрузка этой операции не рекомендуется, т.к. может привести к трудноуловимым ошибкам в коде. Вместо её перегрузки рекомендуется определить метод, который делает то же самое, но не экранирует стандартную реализацию «&»:

тип address\_of ();

# Перегрузка операции приведения типа

Перегрузка операции приведения объекта к некоторому типу осуществляется путём объявления метода





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 «( )»

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

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

# Перегрузка операторов new и delete

Перегрузку new и delete можно использовать для следующих целей:

- Можно реализовать собственный распределитель памяти, который работает более быстро

- Либо более оптимально (кэшируя выделения, освобождения памяти и повторные выделения для создания однотипных объектов)

- Либо можно добавить трассировку выделения и освобождения памяти для поиска утечек и других ошибок связанных с динамической памятью.

- Либо можно реализовывать экзотические сценарии, такие как работа с динамической памятью на особой платформе (архитектуре) с разными типами памяти: SDRAM, EEPROM и т. д.

Перегружаем так:

**void \*operator new(size\_t size);**

1. Запрашивается блок памяти, размер которого указывается в параметре (в байтах)
2. Возвращается указатель на выделенный блок, либо китается исключение std::bad\_alloc
3. Если исключение кидает конструктор, то память, выделенная под объект — освобождается.

**void operator delete(void \*p);**

Принимает указатель на ранее выделенную память и возвращает ее в кучу.

Для вариантов с массивами ( new[] и delete[] ) сигнатура соответствующая:

**void \*operator new[](size\_t size);**

**void operator delete[](void \*p);**

Если требуется, чтобы оператор new не бросал исключение в случае нехватки памяти, а возвращал ноль, можно использовать оператор new с параметром std::nothrow. Этот параметр имеет тип std::nothrow\_t.

// Объявление: **void \*operator new(size\_t size, const std::nothrow\_t &nt);**

// Пример использования: **A \*a = new(std::nothrow) A;**