**Классы: копирование и присваивание**

Малышев Сергей Михайлович

В этой части мы продолжим начатое в статье "Элементы класса, о которых всегда необходимо помнить" обсуждение конструктора копий (copy constructor) и операции присваивания (assignment operator). Или, вернее, начнем подробное рассмотрение весьма нетривиальной проблемы, каковой на самом деле является копирование и присваивание в классах.

Эти два элемента вполне заслужили отдельного рассмотрения. Создание программ на C++ без понимания внутренней сущности этих функций-членов сродни бегу на марафонскую дистанцию без тренировки (возможно, это не самое удачное сравнение, проще говоря, эти функции очень важны).

Конструктор копий служит для создания новых объектов из существующих. Операция присваивания нужна для того, чтобы сделать один существующий объект эквивалентным другому существующему.

Что означает <создать копию>? Как один из вариантов, это означает присваивание значений элементов одного объекта элементам другого. Этот ответ, однако, далеко не полон. C++ - это язык, который практически не ограничивает выбор пути реализации программы. И способ создания копий объектов - не исключение из этого правила.

Иногда для копирования классов достаточно просто привести один объект в то же состояние, что и другой. Это весьма просто, и мы увидим, как это делается. Однако, если вашему приложению требуются другие методы копирования, C++ не станет создавать их за вас, хотя, если вы не напишете эти функции, компилятор сделает это сам. Правда, результат при этом может существенно отличаться от того, что вам бы хотелось.

В серии этих статей мы рассмотрим все аспекты этого вопроса, по разделам:

Понятие копирования;

Копирование буквальное и развернутое;

Когда выполняется копирование;

Разница между копированием и присваиванием

Положение в классах

Блокирование копирования и присваивания

Реализация копирования через присваивание

Копирование и присваивание в производных классах

Понятие копирования

Здесь мы поговорим об одном из аспектов внутреннего функционирования программ, написанных на C++ - о копировании. Копирование в программах на C++ происходит, прямо или косвенно, буквально на каждом шагу. Причем, не всегда с первого взгляда очевидно, где происходит копирование, а где - нет.

Мы начнем с рассмотрения син-таксиса интересующего нас предмета, а затем попробуем углубиться в его осмысление.

Определение конструктора копий.

Конструктор копий используется для создания новых объектов из уже суще-ствующих. Это означает, что, так же как для других конструкторов, новый объект еще не существует к моменту его вызова. Однако только конструкто-ру копий объект передается как аргумент по ссылке. Итак, синтаксис кон-структора копий прост. Конструктор копий произвольного класса X выгля-дит так:

Х(const X&) ; // конструктор копий класса Х

Так как конструктор копий - это все таки конструктор, то он должен иметь имя, совпадающее с именем класса (не забывайте - с учетом регистра символов). Назначение конструктора копий - дублирование объекта-аргумента для построения нового объекта.

Одно из основных правил: если аргумент не должен изменяться, то его следует передавать как константу. В то же время, если аргумент не описан как константа, то нельзя копировать объекты-константы. Переменный объект всегда можно передать как постоянный аргумент, но не наоборот.

Вторая часть объявления аргумента, X, проста: копируется объект того же самого типа. Аргумент в целом читается как "постоянная ссылка на X". Ссылка существенна по нескольким соображениям. В первую очередь пото-му, что при передаче адреса объекта не создается копия вызывающего объ-екта (в отличие от передачи аргумента по значению).

Если вам чудится здесь какой-то подвох, то будьте внимательны. Работа конструктора копий - создание ранее не существовавшего объекта из уже существующего, а передача по значению (без использования операции получения адреса) требует создания копии аргумента, значит мы получаем бесконечную рекурсию.

Точнее: при передаче объекта по значению создает-ся его копия, если это произойдет в конструкторе копий, то он будет вызы-вать сам себя, пока не исчерпает все ресурсы системы.

Вот несколько правил, которым надо следовать при объявлениях конструк-тора копий:

Имя функции точно совпадает с именем класса.

Аргумент объявляется постоянным, что позволяет принимать как по-стоянные, так и переменные аргументы.

Тип аргумента является типом класса.

Аргумент передается по ссылке, т. е. с помощью операции получения адреса.

Написание конструктора копий является чрезвычайно ответственным поступком. Явное определение конструктора копий вызывает изменения в работе программы. Пока мы не пытались переопределить конструктор копий, исправно работал конструктор, порождаемый компилятором автоматически.

Этот конструктор создавал "фотографические" копии объектов, то есть копировал значения абсолютно всех данных-членов, в том числе и ненулевые значения указателей, представляющие адреса динамических областей памяти.

С момента появления переопределённой версии конструктора копий, вся работа по реализации алгоритмов копирования возлагается на программиста. Впрочем, заставить конструктор копий копировать объекты совсем несложно. Создадим класс, описывающий точку на плоскости.

class POINT

{

public:

POINT() { X=0; Y=0; } //конструктор по умолчанию

POINT(int a, int b) { X=a; Y=b; } //еще конструктор

POINT(const POINT& Pixel) { X=Pixel.X; Y=Pixel.Y; } //конструктор

//копирования

int X; //координаты точки

int Y;

};

Как видим, этот конструктор копий просто копирует значения координат. В принципе, если бы мы его не определили, то компилятор создал бы его сам, причем в этом случае он делал бы то же самое. Но конструктор, создающий подобные копии объектов, скорее всего, окажется непригодным для работы с объектами, содержащими в качестве членов указатели или ссылки.

Предполо-жим, что класс содержит указатели. Тогда адреса, содержащиеся в указате-лях объекта-оригинала и объекта-копии, будут идентичны. Это означает, что два объекта будут указывать на одну и ту же область памяти, что, как прави-ло, очень опасно. Это мы рассмотрим подробно несколько позднее.

В переопределяемом конструкторе копий (а в классе он может быть только один) можно реализовывать разнообразные алгоритмы распределения памяти. Здесь всё зависит от программиста.

Итак, обычно, конструктор копий, созданный компилятором, удовлетворителен при следующих обстоятельствах:

Среди членов класса нет указателей (\*).

Среди членов класса нет ссылок (&).

В этом случае вполне разумно использовать конструктор, построенный компилятором. В текст определения класса в этом случае полезно поместить соответст-вующий комментарий о том, что конструктор копии не определен вполне сознательно. Вообще говоря, комментарии относи-тельно рассматриваемых четырех членов уместны в каждом классе.

Вот пример:

class X

{

public:

Х(); // конструктор по умолчанию

virtual ~X(); // виртуальный деструктор

// Конструктор копии и операция присваивания не определены

// намеренно. Класс содержит только данные, размещаемые

// в стеке, поэтому предопределенных конструктора копий

//и операции присваивания достаточно.

private:

int data;

char moreData;

float no\_Pointers;

};

Если хотя бы одно из названных условий не выполняется, то следует опре-делить как конструктор копий, так и операцию присваивания.

Определение операции присваивания

По функциональному назначению операция присваивания очень похожа на конструктор копий. Принципиальное отличие состоит в том, что конструктор копий создает новый (возможно, временный) объект, а операция присваивания работает с уже созданными. Вызывающий объект является левым операндом, объект-аргумент - правым.

Операция присваивания также имеет соответствующий синтаксис. Операция присваивания - это функция-член и одновременно двухместная операция. Следовательно, в работу вовлечены два объекта. Первый объект - вызывающий, доступный по указателю this, а второй - это аргумент. Как конструктор копий, так и операция присваивания используют в качестве аргумента постоянную ссылку.

Для произвольного класса X мы имеем следующий синтаксис операции присваивания:

X& operator=(const X&); // синтаксис операции присваивания для

// произвольного класса

Присваивание - это операция, значит мы должны использовать ключевое слово operator и соответствующий символ операции. Так как C++ допускает цепочки присваивания а = b = с = d;

// C++ допускает последовательные присваивания,

// так что это свойство надо сохранить,

то необходимо возвращать ссылку на объект; в противном случае цепочка прервется.

Итак, оператор-функция принимает постоянную ссылку, а возвращает ссылку на объект. Использование ключевого слова const позволяет функции работать как с постоянными объектами, так и с переменными.

Определяя новый класс, если вы решили объявить операцию присваивания, следуйте следующим рекомендациям:

Операция присваивания должна быть членом класса.

Она принимает постоянную ссылку на объект типа того же класса.

Она возвращает ссылку на объект типа того же класса.

Проверка на присваивание самому себе.

В операции присваивания для любого класса надо учитывать один важный момент. Всегда надо проверять: не происходит ли присваивания самому себе. Оно может иметь место в том случае, когда объект прямо или косвенно вызывает операцию присваивания для себя. Прямое присваивание может выглядеть следующим образом:

POINT Pix;

Pix = Pix; // присваивание самому себе

Это самый тривиальный случай, он хорош для приведения примера, не более. В реальных программах такого обычно не бывает и эта ошибка, как правило, принимает далеко не столь очевидные обличия.

Присваивание самому себе порож-дает в программе утечки памяти. Такая ситуация может возникнуть, когда два объекта сообща используют некоторый ресурс и один из них этот ресурс освобождает. При этом состояние ресурса становится неопределенным, но второй объект продолжает на него ссылаться.

Есть много путей, ведущих к возникновению этой проблемы. Для ее предотвращения и следует предусмотреть в операции присваивания проверку на присваивание самому себе. Она очень проста и выглядит всегда совершенно одинаково:

POINT& POINT::operator=(const POINT& rhs)

{

if(this == &rhs) return \*this; // проверка на присваивание себе

else { X=rhs.X; Y=rhs.Y; } //то, что делает оператор полезного

return \*this; // возврат ссылки на объект

}

Сейчас мы попробуем в ней разобраться, благо для всех операций присваивания проверка на присваивание себе со-вершенно одинакова. Оператор if(this == &rhs) проверяет, не совпадает ли аргумент с самим объектом. Указатель this со-держит адрес вызывающего объекта, &rhs читается как "адрес rhs".

Таким образом, сравниваются два адреса. Если они эквивалентны (==), то это один и тот же объект. В этом случае, в полном соответствии с требованием воз-врата ссылки на объект, просто возвращаем \*this (заметьте, что в конце функции делается то же самое) и выходим из функции.

Вспомните, что this - это указатель. Значение указателя - это адрес. Чтобы получить значение указателя, его следует разыменовывать. Разыменование указателя выглядит так: \*ptr. Указатель this разыменовывается точно так же: \*this.

Помещая эти две строки в начале и в конце тела операции присваивания, мы уменьшаем вероятность возникновения утечек памяти из-за этой опера-ции. Если вы запомнили приведенный здесь синтаксис копирования и при-сваивания и, определяя новый класс, сразу будете определять и их тоже, то это уже полдела.

Зачем C++ требует определения этих функций-членов?

Язык C++ не слишком сильно ограничивает свободу программистов в методах разработки программного обеспечения. В частности, он не навязывает вам способы копирования и присваивания. Количество и разнообразие ситуаций, в которых происходит копирование объектов, удивительно велико. Для очень простых объектов, состоящих из одного-двух элементов, затраты на копирование незначительны, но для более сложных, таких как графический интерфейс пользователя или комплексные типы данных, оперирующие с динамической памятью, издержки на копирование существенно возрастают.

Во-первых, имейте в виду, что если вы не определите для нового класса конструктор копий, то C++ создаст его сам. Причина заключается в том, что компилятору самому может потребоваться возможность создания копий, значит, эти две функции обязаны быть определены.

Во-вторых, вам может потребоваться заблокировать копирование, либо вести подсчет ссылок, или еще что-нибудь. Если вы не создадите эти функции, то C++ создаст для них версии по умолчанию.

Создаваемые компилятором версии обеих этих функций не всегда будут вас удовлетворять. Версии компилятора выполняют буквальное, или поразряд-ное, копирование. В некоторых случаях это неразумно. Не пожалейте времени на изучение ситуаций, которые могут вам встретиться при разработке программ.

Вот лишь некоторые из бесчисленного множества возможных ситуаций, в которых происходит копирование:

POINT х;

POINT у(х); // Прямой вызов конструктора копий.

POINT х = у; // Выглядит как присваивание, но на самом деле

// вызывает конструктор копий. Почему? См. ниже.

POINT a, b;

a = b; // Вызов операции присваивания

POINT Foo(); // Возврат по значению, вызывает копирование

void Foo(POINT); // Передача по значению, создает копию

Во всех этих случаях выполняется копирование. В ходе выполняемой компилятором оптимизации могут появиться и другие варианты. Это та область, где знание действительно сила, способная помочь вам избежать утечек памяти.

В операторе типа POINT х = у; не вызывается операция присваивания класса POINT, хотя на первый взгляд выглядит это именно так. Причина состоит в том, что операция присваивания - это функция-член, а значит может быть вызвана только для уже существующих объектов, в то время как в этом фрагменте происходит создание нового объекта х.

Если объект создается в той же строке, в которой он выступает в качестве левостороннего аргумента, то вызывается конструктор. Строка

Х х = у; // вызов конструктора копий

эквивалентна строке

Х х(у); // вызов конструктора копий

БИКЮ , что совсем не то же самое, что

Х х, у;

х = у; // вызов операции присваивания

Вам следует понимать, что же на самом деле вызывается, когда и почему. Это одна из тех особенностей, благодаря которым C++ труднее и интерес-нее, чем С. В предыдущем разделе мы пришли к заключению, что не стоит определять операцию присваивания без конструктора копий и наоборот.

Следовательно, напрашивается вывод, что основные рекомендации для операции присваивания справедливы также и для конструктора копий.

На этом, пожалуй пока и остановимся. Небольшое резюме напоследок.

Если класс содержит указатели или ссылки, то скорее всего вам придется определять операцию присваивания и конструктор копий для этого класса самостоятельно, не полагаясь на компилятор. В противном случае можно спокойно использо-вать созданные компилятором присваивание и копирование, но при этом по-лезно упомянуть об этом в комментариях к классу.

**Список литературы**

P.Kimmel Using Borland C++ 5 Special Edition перевод BHV - С.Петербург 1997

C++. Бархатный путь Марченко А.Л. Центр Информационных Технологий

www.citmgu.ru

Thinking in C++, 2nd ed. Volume 1 c2000 by Bruce Eckel