Теоретическая часть третьего домашнего задания

0. Что такое **lvalue** и **rvalue** ссылки, чем они отличаются друг от друга?

Допустим, что у нас есть некоторое выражение. Зачастую здесь есть оператор присваивания «=». Тогда все, что слева — *lvalue*, справа — *rvalue*, - переменные, некоторые функции, возвращающие переменные и области памяти, адреса. В *lvalue* - переменные, в *rvalue* - рандомные значения, переменные, константы, функции. *Основное отличие* — слева всегда выделяется память под выражение, а справа - не факт.

Операция &(lvalue) вернет адрес левой части. Как выглядят *rvalue* и *lvalue*?

```
int &val = arg; // lvalue
int &&val = 10; // rvalue
```

Как эти ссылки превращать друг в друга? *lvalue* ссылки нужны, на них есть адрес и мы с ними обычно не хотим сильно шаманить. А вот насчет *rvalue* беспокоится куда меньше людей.

Есть пример и среди конструкторов: *move* - конструктор. Он нужен для ускорений некоторых процессов. Он соответсвует *rvalue*.

*(*источник: конспект с семинара по программированию 3*)*

C++ может генерировать временную переменную, если фактический аргумент не соответствует ссылочному аргументу. В настоящее время C++ допускает это только в случае, когда аргументом является ссылка с квалификатором CONST, но это не всегда так. Рассмотрим случаи, когда C++ генерирует временную переменную и выясним, почему ограничение, требующее ссылки const, имеет смысл.

Прежде всего, в каких случаях создается переменная? При условии, что ссылочный параметр является const, компилятор генерирует временную переменную в двух ситуациях:

1. Когда тип фактического аргумента выбрал правильно, но асам параметр не является lvalue;
2. Когда типа фактического параметра выбран неправильно, но может быть преобразован в правильный тип;

Что такое lvalue? Аргумент, являющийся lvalue, представляет собой объект данных, на который можно ссылаться по адресу. Например, переменная, элемент массива, член структуры, ссылка и разыменованный указатель, - все они являются lvalue. К lvalue не относятся литеральные константы (кроме строк в двойных кавычках, которые представлены своими адресами) и выражения, состоящие из нескольких элементов. Понятие lvalue в C первоначально означало сущности, которые могли находиться в левой части оператора присваивания, но это было до появления ключевого слова const. Теперь как обычная, так и переменная const могут рассматриваться как lvalue, поскольку к ним обеим можно обращаться по адресу. Вдобавок обычная переменная может быть дополнительно определена как изменяемое lvalue, а переменная const — как неизменяемая lvalue.

На заметку!

Если передаваемый функции аргумент не является lvalue или не совместим по типу с соответствующим ссылочным параметром const, C++ создает анонимную переменную требуемого типа, присваивает ей значение передаваемого функции аргумента, и делает так, чтобы параметр ссылался на эту переменную.

Используйте const, когда это возможно

Существует три серьезных причины объявлять ссылочные аргументы как ссылки на константные данные.

1. Использование const защищает от внесения в программы ошибок, приводящих к непреднамеренному изменению данных.

2. Использование const позволяет функции обрабатывать фактические аргументы как с const, так и без const. При этом функция, в прототипе которой квалификатор const опущен, может принимать только неконстантные данные.

3. Использование ссылки const позволяет функции генерировать и использовать временные переменные по мере необходимости.

Формальные ссылочные аргументы рекомендуется объявлять с квалификатором const во всех случаях, когда для этого есть возможность.

В C++11 появилась вторая разновидность ссылки — ссылка rvalue, которая может ссылаться на rvalue. Она объявляется применением &&:

```
double && rref = std::sqrt(36.00); // not allowed for double &
double j = 15.0;
double && jref = 2.0* j + 18.5; // not allowed for double &
std::cout << rref << '\n'; // displays 6.0
std::cout << jref << '\n'; // displays 48.5
```

Ссылка rvalue была введена в основном для того, чтобы помочь разработчикам библиотек предоставлять более эффективные реализации определенных операций. Исходный ссылочный тип (с использованием &) теперь называется ссылкой lvalue.

*(*источник: Прата «Язык программирования C++. Лекции и упражнения.», стр. 379*)*

1. Что такое списки инициализации конструктора? Зачем они нужны?

Списки инициализации - суть в том, что у нашего класса, наших объектов есть какие-то поля: int, char. Их можно использовать внутри конструктора и не будет проблем. Что нам мешает внутрь нашей строки вставить стандартную? Ничего, но будет существенное отличи, однако во время входа в конструктор, когда мы зашли за первую фигурную скобку, создаются все поля нашего объекта. Если мы будем там присваивать значение, то мы будем именно присваивать значение. Если обычный тип данных - ничего страшного. А если это класс, что делать, если мы вызвали класс?.. Чтобы таких вопросов не было, придумали списки инициализации, то есть список конструкторов.

*(*источник: конспект с семинара по программированию 3*)*

Можно ли в C++11 использовать синтаксис списковой инициализации для классов? Да, можно; для этого потребуется представить в фигурных скобках содержимое, соответствующее списку аргументов конструктора:

```
Stock hot_tip = {«Derivatives Plus Plus», 100, 45.0};
Stock jock {«Sport Age Storage, Inc»};
Stock temp {};
```

Списки в фигурных скобках в первых двух объявлениях соответствуют следующему конструктору:

```
Stock::Stock(const std::string & co, long n = 0, double pr = 0.0};
```

Таким образом, этот конструктор будет использоваться для создания двух объектов. В случае объекта jock для второго и третьего аргументов будут применяться значения по умолчанию — 0 и 0.0. Третье объявление соответствует конструктору по умолчанию, поэтому объект temp будет создан с его помощью.

Вдобавок C++11 предлагает класс по имени std::initializer_list, который может использоваться в качестве типа для параметра функции или метода. Этот класс представляет собой список произвольной длины, все элементы которого имеют один и тот же тип или могут быть преобразованы к одному типу.

*(*источник: Прата «Язык программирования C++. Лекции и упражнения.», стр. 512*)*

2. Какие типы конструкторов вы знаете? В чем особенность каждого из них, зачем он нужен?

Теперь, после ознакомления с рядом примеров конструкторов и деструкторов сделаем паузу и подведем некоторые итоги. Ниже приводится краткий обзор этих методов.

Конструктор — это специализированная функция-член класса, которая вызывается всякий раз при создании объекта данного класса. Конструктор класса имеет то же имя, что и класс, но благодаря возможностям перегрузки функции, существует возможность создавать более одного конструктора с одним и тем же именем и разным набором аргументов. Кроме того, конструктор не имеет объявленного типа. Обычно конструктор используется для инициализации членов объекта класса. Ваша инициализация должна соответствовать списку аргументов конструктора. Например, предположим, что класс Bozo имеет следующий прототип для конструктора:

*Bozo (const char * fname, const char * lname); // constructor prototype*

В этом случае его можно использовать для инициализации объекта следующим образом:

Bozo bozetta = Bozo(«Bozetta», «Biggens»); // main form
Bozo fufu(«Fufu», «O'Dweeb»); // shorted form
*Bozo *pc = new Bozo(«Popo», «Le Peu»); // dynamic object*

В C++11 можно взамен применять спусковую инициализацию:

Bozo bozetta = Bozo{«Bozetta», «Biggens»}; // C++
Bozo fufu{«Fufu», «O'Dweeb»}; // C++
*Bozo *pc = new Bozo{«Popo», «Le Peu»}; // C++*

Когда конструктор имеет только один аргумент, он вызывается в случае инициализации объекта значением, которое имеет тот же тип, что и аргумент конструктора. Например, предположим, что существует следующий прототип конструктора:

Bozo(int age);

Тогда в коде можно использовать любую из следующих форм инициализации объекта:

Bozo dribble = Bozo(44); // primary form
Bozo roon(66); // secondary form
Bozo tubby = 32; // special form of constructor with single argument // new one

Конструктор по умолчанию не имеет аргументов и используется, когда вы создаете объект без явной его инициализации. Если вы не предоставляете ни одного конструктора, то компилятор создаст конструктор по умолчанию самостоятельно. В противном случае вы обязаны определить собственный конструктор по умолчанию. Он может либо не иметь аргументов, либо предусматривать знания по умолчанию для всех аргументов:

Bozo(); // prototype of default constructor
*Bistro(const char *s = «Chez Zero»); // default value for class Bistro*

Программа использует конструкторы по умолчанию для неинициализированных объектов:

Bozo bibi; // using default constructor
*Bozo *pb = new Bozo; // using default constructor*

Подобно тому, как при создании объекта вызывается конструктор, деструтор вызывается при его уничтожении.

*(*источник: Прата «Язык программирования C++. Лекции и упражнения.», стр. 513*)*

3. Как и для чего нужно использовать **const** в методах класса.

Используйте const, когда это возможно

Существует три серьезных причины объявлять ссылочные аргументы как ссылки на константные данные.

1. Использование const защищает от внесения в программы ошибок, приводящих к непреднамеренному изменению данных.
2. Использование const позволяет функции обрабатывать фактические аргументы как с const, так и без const. При этом функция, в прототипе которой квалификатор const опущен, может принимать только неконстантные данные.
3. Использование ссылки const позволяет функции генерировать и использовать временные переменные по мере необходимости.

Формальные ссылочные аргументы рекомендуется объявлять с квалификатором const во всех случаях, когда для этого есть возможность.

Рассмотрим следующий фрагмент кода:

```
const Stock land = Stock(«Kludgehorn Properties»);
land.show();
```

Компилятор совершенного языка C++ не должен принять вторую строку. Почему? Причина в том, что код show() не гарантирует того, что он не изменит объект, который из-за объявления как const меняться не должен. Вы должны предварительно позаботиться о решении этой проблемы, объявив аргумент функции как ссылку const. Вместо них используемый объект неявно задан вызовом этого метода. Необходим новый синтаксис, который укажет на то, что функция-член не будет модифицировать объект. Решение, предлагаемое C++, заключается в помещении ключевого слова const после скобок функции. То есть объявление метода show() должно выглядеть следующим образом:

```
void show() const; // promises not to change object
```

Аналогично начало определения функции должно выглядеть так, как показано ниже:

```
void Stock::show() const // promises not to change called object
```

Функции класса, объявленные и определенные подобным образом, называются константными функциями-членами. Точно так же, как константные ссылки и указатели, где это необходимо, используется в качестве формальных аргументов функций, вы должны делать методы класса константными всегда, когда они не модифицируют объект, с которым работают. Отныне мы будем следовать этому правилу.

*(*источник: Прата «Язык программирования C++. Лекции и упражнения.», стр. 512*)*

4. Как можно переопределять операторы в C++? Какие есть ограничения?

Оператор в C++ - это некоторое действие или функция обозначенная специальным символом. Для того что бы распространять эти действия на новые типы данных, при этом сохраняя естественный синтаксис, в C++ была введена возможность перегрузки операторов.

Не все операторы можно переопределять. Операторы "." и «a?b:c" (тернарный оператор) переопределить нельзя. Так же нужно отметить, что переопределяя операторы "," "&&" "||" теряются их "ленивые" свойства. Операторы "a->", "[]", "()", "=" и "(type)" можно переопределить только как методы класса.

В C++ можно выделить четыре типа перегрузок операторов:

1. Перегрузка обычных операторов + - * / % ^ & | ~ ! = < > += -= *= /=%= ^= &= |= << >> >>= <<= == != <= >= && || ++ -- , ->* -> ( ) <=> [ ]
2. Перегрузка операторов преобразования типа
3. Перегрузка операторов аллокации и деаллокации new и delete
4. Перегрузка литералов operator»"

Обычные операторы

Важно помнить, что перегрузка расширяет возможности языка, а не изменяет язык, поэтому перегружать операторы для встроенных типов нельзя. Нельзя менять приоритет и ассоциативность (слева направо или справа налево) операторов. Нельзя создавать собственные операторы и перегружать некоторые встроенные: :: . .* ?: sizeof typeid. Также операторы && || , теряют свои уникальные свойства при перегрузке: ленивость для первых двух и очерёдность для запятой (порядок выражений между запятыми строго определён, как лево-ассоциативный, то есть слева-направо). Оператор -> должен возвращать либо указатель, либо объект (по копии или ссылке).

Операторы могут быть перегружены и как отдельные функции, и как функции-члены класса. Во втором случае левым аргументом оператора всегда выступает объект *this. Операторы = -> [] () могут быть перегружены только как методы (функции-члены), но не как функции.

Можно сильно облегчить написание кода, если производить перегрузку операторов в определённом порядке. Это не только ускорит написание, но и избавит от повторений одного и того же кода. Рассмотрим перегрузку на примере класса, представляющего собой геометрическую точку в двумерном векторном пространстве:

```
class Point
{
    int x, y;
  public:
    Point(int x, int xx): x(x), y(xx) {}  // Конструктор по-умолчанию исчез.
    // Имена аргументов конструктора могут совпадать с именами полей класса.
}
```

- Операторы присваивания копированием и перемещением operator=
- Стоит учитывать, что по-умолчанию C++ помимо конструктора создаёт пять базовых функций. Поэтому перегрузку операторов присваивания копированием и перемещением лучше отдать на реализацию компилятору или реализовать с помощью идиомы Copy-and-swap.
- Комбинированные арифметические операторы += *= -= /=%= и т. д.
- Если мы хотим реализовать обычные бинарные арифметические операторы, удобнее будет реализовать вначале данную группу операторов.

```
Point& Point::operator+=(const Point& rhs) {
    x += rhs.x;
    y += rhs.y;
    return *this;
}
```

Оператор возвращает значение по ссылке, это позволяет писать такие конструкции: (a += b) += c;

- Арифметические операторы + * - / %
- Чтобы избавиться от повторения кода, воспользуемся нашим комбинированным оператором. Оператор не модифицирует объект, поэтому возвращает новый объект.

```
const Point Point::operator+(const Point& rhs) const {
return Point(*this) += rhs;
}
```

Оператор возвращает const значение. Это защитит нас от написания конструкций подобного вида (a + b) = c;. С другой стороны, для классов, копирование которых дорого обходится, гораздо выгоднее возвращать значение по неконстантной копии, то есть : MyClass MyClass::operator+(const MyClass& rhs) const;. Тогда при такой записи x = y + z; будет вызван конструктор перемещения, а не копирования.

- Унарные арифметические операторы + -
- Унарные плюс и минус не принимают аргументов при перегрузке. Они не изменяют сам объект (в нашем случае), а возвращают новый изменённый объект. Следует перегрузить и их, если перегружены их бинарные аналоги.

```
Point Point::operator+() {
	return Point(*this);
}
Point Point::operator-() {
	Point tmp(*this);
	tmp.x*=-1;
	tmp.y*=-1;
	return tmp;
}
```

- Операторы сравнения == != < <= > >=
- Первыми следует перегрузить операторы равенства и неравенства. Оператор неравенства будет использовать оператор равенства.
-

```
	bool Point::operator==(const Point& rhs) const {
		return (this->x == rhs.x && this->y == rhs.y);
	}
	bool Point::operator!=(const Point& rhs) const {
		return !(*this == rhs);
	}
```

Следом перегружаются операторы < и >, а затем их нестрогие аналоги, с помощью ранее перегруженных операторов. Для точек в геометрии такая операция не определена, поэтому в данном примере нет смысла их перегружать.

- Побитовые операторы <<= >>= &= |= ^= и << >> & | ^ ~
- На них распространяется те же принципы, что и на арифметические. В некоторых классах пригодится использование битовой маски std::bitset. Внимание: оператор & имеет унарный аналог и используется для взятия адреса; обычно не перегружается.
- Логические операторы && ||
- Эти операторы потеряют свои уникальные свойства ленивости при перегрузке.
- Инкремент и декремент ++ --
- C++ позволяет перегрузить как постфиксные, так и префиксные инкремент и декремент. Рассмотрим инкремент:
-

```
Point& Point::operator++() { //префиксный
	x++;
	y++;
	return *this;
}
Point Point::operator++(int) { //постфиксный
	Point tmp(x,y,i);
	++(*this);
	return tmp;
}
```

Заметим, что функция-член operator++(int) принимает значение типа int, но у этого аргумента нет имени. C++ позволяет создавать такие функции. Мы можем присвоить ему

(аргументу) имя и увеличивать значения точек на этот коэффициент, однако в операторной форме этот аргумент по-умолчанию будет равен нулю и вызывать его можно будет только в функциональном стиле: A.operator++(5);

- Оператор () не имеет ограничений на тип возвращаемого значения и типы/количество аргументов и позволяет создавать функторы.
- Оператор передачи класса в поток вывода. Реализуется в виде отдельной функции, а не функции-члена. В классе эта функция помечается как дружественная.friend std::ostream& operator<<(const ostream& s, const Point& p);

На остальные операторы не распространяются какие-либо общие рекомендации к перегрузке.

*(*источник: wikipedia*)*

5. Для чего нужно ключевое слово **friend**?

Функция, не являющаяся членом класса, может иметь доступ к его частным членам в случае, если она объявлена другом (friend) класса. Например, в следующем примере функция frd() объявлена другом класса cl:

```
    class cl {
public:
    friend void frd();
};
```

Как можно видеть, ключевое слово *friend* предшествует объявлению функции. Одна из причин, почему язык C++ допускает существование функций-друзей, связана с той ситуацией, когда два класса должны использовать одну и ту же функцию.

Имеется два важных ограничения применительно к дружественным функциям. Первое заключается в том, что производные классы не наследуют дружественных функций. Второе заключается в том, что дружественные функции не могут объявляться с ключевыми словами *static* или *extern*.

6. В чем особенность **new** и **delete** по сравнению с **malloc( )** и **free( )**?

1. *new* бросает исключение в случае неудачи, *malloc* возвращает 0.
2. *new* вызывает конструктор класса и выделяет память либо просто вызывает конструктор (в случае placement *new*), *malloc* только выделяет память.
3. *new* нужен для работы с объектами, *malloc* для работы непосредственно с памятью.

Эти операторы связаны с динамическим выделением памяти. Динамическая память может выделяться в одной функции и освобождаться в другой. В отличие от автоматической памяти, динамическая память не подчиняется схеме LIFO. Порядок выделения и освобождения памяти определяется по тому, когда и как применяются операции *new* и *delete*.

Несмотря на то, что концепции схем хранения неприменимы к динамической памяти, они применимы к автоматическим и статистическим переменным-указателям, используемым для отслеживания динамической памяти. Например:

```
float * p_fees = new float [20] ;
```

Некоторое количество памяти (лучше использовать то, какой размер знаешь) остаются занятыми до тех пор, пока операция *delete* не разлучит их. Однако этот указатель перестает существовать по выходе из блока. Если требуется возможность работы с другими функциями, то задавать нужно глобально, а в самой функции использовать *external*. Рассмотрим инициализацию с помощью new.

Если требуется создать и инициализировать хранилище для одного из встроенных скалярных типов, необходимо указать имя типа и инициализирующее значение, заключенное в круглых скобках:

```
int *pi = new int (6); // *pi устанавливается в 6.
double * pd = new double (99.99); // *pd устанавливается в 99.99
```

Синтаксис с круглыми собаками также может использоваться с классами, которые имеют подходящие конструкторы, но об этом позже (или нет).

Для инициализации обычной структуры или массива необходим стандарт C++11 и фигурные скобки списковой инициализации. Новый стандарт позволяет следующее:

```
struct where {double x; double y; double z;};
where * one = new where {2.5, 5.3, 7.2}; // C++11
int * ar = new int [4] {1, 2, 3, 4}; // C++11
```

Также можно применять инициализацию с помощью фигурных скобок для переменных с одиночным значением:

```
int *pin = new int { }; // !TODO
double * pdo = new double {99.99};
```

*(*источник: Пата «Язык программирования C++. Лекции и упражнения.», стр. 457*)*

new — оператор языка программирования C++, обеспечивающий выделение динамической памяти в куче. За исключением формы, называемой «размещающей формой new», new пытается выделить достаточно памяти в куче для размещения новых данных и, в случае успеха, возвращает адрес выделенного участка памяти. Однако, если *new* не может выделить память в куче, то он передаст (*throw*) исключение типа «*std::bad_alloc*». Это устраняет необходимость явной проверки результата выделения. После встречи компилятором ключевого слова new им генерируется вызов конструктора класса.

*(*источник: wikipedia*)*