# Инкапсуляция и организации доступа к элементам объекта

Класс- это основа, на которой построен язык с++ для реализации поддержки ООП (объектно-ориентированного программирования)

Класс позволяет реализовать объекты по определенному шаблону

Структура класса имеет вид: ключевое слово

class Имя\_Класса

{

набор полей;

набор мета данных;

};

Различия метода и функций

Функция не принадлежит никакому классу, а метод всегда вписан внутри какого-либо класса

У каждого элемента класса может быть свой атрибут доступа (access specifier)

Атрибуты доступа в классе могут быть 3 видов открытый, закрытый, защищённый (public, private, protected)

Класс представляет собой описание пользовательского типа.

При именовании класса рекомендуется придерживаться верблюжьей нотации (Camel Case), которая предполагает написание каждого слова с новой буквы.

Инстанциировать (Instance) объект означает создать объект определённого класса. Всё ООП основано на понятии класса и объекта.

При создании класса всегда создаётся шаблон, чертёж, схема, описание, тип данных, кало.

Класс связывает данные с кодом, который обрабатывает эти данные. Объекты- это экземпляры определённого класса. Класс представляет собой план, который определяет, как можно построить объект. Класс- это логическая абстракция, которая реально не существует до тех пор, пока не будет создан объект этого класса, т.е. пока не будет физически выделена память под этот объект.

К принципам ООП относятся: Инкапсуляция, наследование, полеморфизм.

Инкапсуляция для класса предполагает, что данные и функции работы с этими данными собраны в одном классе.

Переменные объявленные в классе называются элементом данных класса или полем класса, а функция, объявленная в классе, называется методами класса.

Очередь (Queue(кью))

Для открытия элементов класса используется атрибут доступа public. **Обычно** поля класса делаются приватными, методы класса делаются публичными. В описании класса обычно указывается прототип метода. Для создания объектов класса необходимо указать имя класса и через пробел имя создаваемого объекта.

Доступ к приватной переменной любого класса вне класса невозможно.

При реализации метода класса сначала необходимо указать тип возвращаемого значения, затем имя класса, метод которого реализуется, затем ставится оператор :: (оператор расширения области видимости (Scope Resolution Operator), который имеет наивысший приоритет исполнения, после :: пишется имя метода с набором получаемых элементов.

Функция не принадлежит никакому классу.

В классе по умолчанию используется атрибут доступа private, закрывающий компоненты от внешнего использования. Атрибут доступа public используется для описания открытых компонентов класса, обычно закрытыми создаются поля класса, открытыми – методы.

Обычно в классе описываются прототипы методов, а реализация их выносится за тело класса. Оператор this можно использовать внутри класса, для указания того, что вы обращаетесь к полю класса.

Для создания объектов класса необходимо указать имя класса, имя создаваемого элемента.

Метод класса через объект будет вызываться с использованием оператора «.» (оператор доступа к содержимому через объект).

Общий формат объявления класса выглядит следующим образом:

Ключевое слово класс, позволяет указать, что вы создаёте пользовательский тип. После ключевого слово class пишется имя класса с использованием CamelCase нотации, позволяющий создавать имя класса в читаемом виде.

Тело класса обычно состоит из описания набора закрытых (private) полей, описания набора открытых (public) методов.

<windows.h> библиотека для определения кодировки.

В head файлах содержится директива препроцессора. Директивы препроцессора содержащие подключаемые библиотеки, создаваемые константы и описание типа. Тип<->класс.

Имя класса всегда описывает одну сущность(entity). В классах обычно создаются методы (сеттеры и геттеры), позволяющие задавать и получать соответственно значение полей класса.

Манипулятор endl позволяет перенести курсор на новую строчку.

Метод класса всегда называется для конкретного объекта и работает только с его полями. Когда доступ к некоторому элементу класса происходит из вне этого класса, то его всегда необходимо квалифицировать (уточнить или указать) с помощью имени конкретного объекта, но код внутри методов класса может обращаться к другим полям класса и методам этого же класса.

# Конструкторы и деструкторы

Конструктор-это метод класса, который вызывается автоматически явно или неявно при создании объекта. Имя конструктора всегда совпадает с именем класса (буква-буква, регистор-регистору). В объявлении конструктора всегда отсутствует тип возвращаемого значения. В конструкторе пишется логика, предназначенная для корректной первоначальной инициализации объектов.

Деструктор-это метод класса, который вызывается автоматически при выходе объекта из области видимости для разрушения всех выделенных под объект ресурсов. Имя деструктора совпадает с именем класса, но претворяется символом тильда (~). Деструкторы никогда не возвращают значений, следовательно, у них отсутствует тип возвращаемого значения. Создание конструктор и их автоматическое использование позволяет гарантировать корректное первоначальное состояние объекта.

Все конструкторы класса делятся на две категории: конструкторы без аргументов (дефолтный или конструктор по умолчанию или No-Argument) и с параметрами. Конструкторы с параметрами предназначены для того, чтобы на основании полученных аргументов, проинициализировать создаваемые объекты. В отличии от конструкторов деструкторы не могут иметь параметры. Если же возникает редкая ситуация, когда при вызове деструктора объекту необходимо получить доступ к некоторым данным, определяемым только во время выполнения программы, тогда необходимо создать для этой цели специальную переменную и перед разрушением объекта установить в этой переменной необходимое значение. Если конструктор класса принимает только один параметр, то можно использовать альтернативный способ инициализации объекта, в котором после указания имени создаваемого объекта ставится знак равно и передаётся аргумент нужного типа. В общем случае если конструктор принимает один объект, то для инициализации можно использовать либо вариант ob(x), либо ob=x

# Сравнение классов и структур

Структура может включать данные(поля) и код(методы), который манипулируют этими данными точно также как это делает класс, единственное различие между классом и структурой состоит в том, что по умолчанию элементы класса являются закрытыми(private), а элементы структуры публичными(public). В основном структуры и классы имеют одно и тоже предназначение. Структура, как и класс определяет пользовательский тип

# Сравнение классов и объединений

Объединение- это тот же класс, в котором все элементы данных хранятся в одной и той же области памяти. Элементы объединения по умолчанию открыты, как для struct (структуры), так и для объединения(union), можно явно переопределить атрибуты доступа, но struct и union не являются частью ООП.

Повторить из курса ОАИП работу с указателями на функции

# Встраиваемые функции

Встраиваемым называется метод код, которого подставляется в то место программы, из которого он(метод) вызывается. Существуют два способа создания встроенного метода:

1. Использование ключевого слова inline(встроить), например, чтобы создать метод, который возвращает целочисленное значение и не получает ни одного элемента, необходимо написать следующую конструкцию:

inline int f()

{

//…

}

Модификатор inline должен предварять все остальные аспекты объявления метода. Встроенный метод-это небольшой по объёму код, который представляет собой вместо его вызова. Причины существования встроенных методов является эффективность их использования. При каждом вызове обычного метода должна быть выполнена последовательность инструкций, связанных с обработкой самого вызова, включающего помещение его(метода) аргументов в стек, и с возвратом значений из метода. В некоторых случаях значительное количество циклов ЦП используется именно для выполнения таких действий. Но если метод встраивается в программу, то подобные системные затраты отсутствуют, и общая скорость выполнения программы возрастает. Если встраиваемый метод окажется большим, то общий размер программы может существенно увеличится. Поэтому лучше всего в качестве встроенных методов очень маленькие методы (содержащие 1-2 оператора), а методы больше оформлять в виде обычных. Использование модификатора inline является запросом (просьбой), а не командой. По которой компилятор сгенерирует встроенный код.

2. Создание встроенного метода написание его реализации внутри класса. Существуют разные ситуации, которые не позволят компилятору выполнить такой запрос, например:

* Метод содержит цикл
* Switch
* go to
* рекурсия
* Встраивание не проходит для методов, который содержит статические переменные

Ограничения на использование встроенных методов зависит от реализации компилятора. При реализации методов в классе предварять ключевым словом inline не нужно.

# Массивы объектов классов

Массивы объектов класса создаются точно также как массивы значений других типов. В конструкции switch могут использоваться int, char, элемент перечисления. Элементы перечисления автоматически нумеруются, начиная с нуля.

Многомерные массивы индексируются точно также, как и массивы значений других типов. Для создания констант на языке C++ необходимо написать ключевое слово const перед типом создаваемой переменной. Одной из форм инициализации полей классов в конструкторе является написание : после сигнатуры конструктора и перечисления полей, через запятую, которые инициализируются значениями в скобках.

MyClass(int a,int b): a(a),b(b) {}

При создании массива объектов и одновременно его инициализации в фигурных скобках необходимо указывать значения, типы которых соответствуют имеющемуся в классе конструктору.

# Указатели на объекты

Доступ к содержимому структуры можно получить напрямую через объект или через указатель на эту структуру. Аналогично можно обращаться и к объекту класса: непосредственно через объект или через указатель на объект. Чтобы получить доступ к отдельному элементу объекта, через объект, используется оператор точка или оператор доступа к содержимому через объект. Если имеется в распоряжении указатель на объект, то необходимо использовать оператор «стрелка» (->), для объявления указателя на объект используется тот же синтаксис, как и объявление указателей на переменные других типов. При инкрементации и декрементации указателя он инкрементируется и декрементируется так чтобы всегда указывал на следующий или предыдущий элемент **базового** типа. То же самое происходит и при инкрементации и декрементации указателя на объект: он будет указывать на следующий или предыдущий объект класса. Указатели на объекты играют главную роль в реализации полиморфизма.

# Friend функции

В С++ существует возможность совершить доступ к закрытым элементам класса методом, который не является элементом этого класса. Для этого достаточно объявить эти методы дружественными (friend) по отношению к рассматриваемому классу.

Чтобы сделать функцию «другом» необходимо включить её прототип в public раздел объявления этого класса и предварить его ключевым словом friend.

class My\_Class

{

public:

friend void func();

};

Ключевое слово friend предоставляет методам, которые не являются элементами класса доступ **ко всем** его элементам. Один метод может быть friend для нескольких классов. Одна из причин почему с++ допускает существование friend функций связано с ситуацией, когда два класса должны использовать одну и ту же функцию. Указатель на базовый класс может указывать на любой объект любого дочернего класса (начало полиморфизма). Friend функции вызываются обычным образом, т.е. без привязки к объекту, без использования оператора «.». Так как friend функция не является методом класса, то при её вызове имя объекта указывать не нужно. **Обычно** friend функции в качестве параметра передаётся один или несколько объектов класса для которых она является friendом, существуют ситуации, при которых статус friend имеет важное значение:

1. friend функции могут быть использованы при перегрузке оператора

2. friend функции упрощают некоторые функции ввода/вывода.

3. В некоторых случаях, два и больше классов находятся во взаимосвязи между собой и с другими частями программы.

Часто при создании friend функции используются опережающее объявление класса т.е. указание ключевого слова class, имени класса и «;»

Friend одного класса может быть методом другого класса.

# Перегрузка функции (function overloading)

Перегрузка функций -это механизм, который позволяет двум родственным по своим задачам функциям позволяет иметь одинаковые имена. Несколько функций в C++ могут иметь одинаковые имена только при условии, что их параметры различны. При этом функции являются перегруженными. Перегрузка функции относится к полиморфизму этапа компиляции. Перегрузка функций один из способов реализации полиморфизма. Поскольку списки параметров для всех версий перегруженной функции различны компилятор обладает достаточной информацией чтобы вызвать правильную версию каждой функции. Для определения того какую версию перегруженной функции вызвать компилятор использует типы, количество и порядок полученных аргументов.

Несмотря на то, что перегруженные методы могут отличаться и типами возвращаемых значений, этого вида информации недостаточно для того чтобы компилятор мог решить, какую именно функцию надо вызвать. Язык C не поддерживает технологию перегрузки функции. Для принятия решения, какую функцию надо вызвать всегда анализируются получаемые аргументы. Перегрузка функций позволяет обращаться к связанным по решаемым задачам функциям посредством одного общего имени. Каждая версия перегруженной функции может выполнять любые действия. Не существует правил, которые бы обязывали программиста связывать перегруженные методы общими выполняемыми действиями. Перегружать имеет смысл только тесно связанные функции. Тип clock\_t представляет собой псевдоним длинного целого числа способного представлять тактовые счётчики. Тип clock\_t представляет число системных тиков. Тактовые тики- это единицы времени постоянной, но системно зависимой длины возвращаемое системными часами. Константа CLOCK\_PER\_SECONDS позволяет преобразовывать количество системных тиков в секунды. При разработке полноценной библиотеки классов необходимо предусмотреть набор различных конструкторов, охватывающий весь необходимый спектр инициализации объекта. Объекты класса могут быть проинициализированы как динамически, так и статически. Динамическая инициализация объектов предполагает определение значения переменной во время выполнения программы, которая может быть получена либо от пользователя, либо из любого выражения инициализации

Наличие множества форматов инициализации избавляет программиста от выполнения дополнительных преобразований при инициализации объекта.

Механизм перегрузки конструктор способствует понижению уровня сложности программ, позволяя создавать объекты наиболее естественным для их образа. Загромождение кодом инструктора для обработки редко возникающих ситуаций оказывают дестабилизирующее влияние на программу.

# Присваивание объектов

Если два объекта имеют одинаковый тип, т.е. являются объектами одного класса, то один объект можно присвоить другому. Для присваивания недостаточно чтобы два класса были физически подобны, имена классов должны совпадать. Если один объект присваивается другому, то по умолчанию данные первого объекта **поразрядно** копируются во второй. Присваивание одного объекта другому делает их данные идентичными, но эти два объекта остаются **совершенно независимыми**. Объекты передаются функциям путём соглашения о передаче параметров **по значению (pass by value).** Функции при передаче по значению передаётся копия объекта. Массивы всегда передаются по ссылке, так как они предназначены для хранения большого набора данных.

# Особенности работы конструкторов и деструктор при передаче объектов в функцию и возврате из функции

Конструктор всегда вызывается при создании объекта, независимо от формы создания объекта (с аргументами, без аргументов и тд). Деструктор всегда вызывается при выходе объекта из области видимости. Деструктор для любого объекта может быть вызван явно. Когда при вызове метода создаётся копия аргумента, обычный конструктор (с параметрами или без) не вызывается. Вместо этого вызывается конструктор копии (конструктор копирования). Если в классе не создан конструктор копирования, то компилятор c++ предоставляет его по умолчанию. Когда метод завершается и разрушается копия объекта, используемая в качестве аргумента метода, вызывается деструктор этого объекта. Конструктор копии созданный по умолчанию создаёт побитовую копию (идентичную копию) этого объекта.

# Потенциальные проблемы при передаче объектов в качестве параметров и при возврате их из функции

Если объект используемый как аргумент требует динамического выделения памяти и освобождает эту память при разрушении, то его локальная копия при вызове деструктора освободит ту же самую область памяти, которая была выделена оригинальному объекту. Повторное освобождение уже освобождённой памяти в стандарте C++ считается неопределённой ошибкой. Работа конструктора копирования заключается в том, чтобы создать реальную копию объекта, со своей личной выделенной динамической памятью. Подобная схема выделения и освобождение динамической памяти происходит при возврате объектов из функции. Если метод возвращает объект класса, то он автоматически создаёт временный объект, который хранит возвращаемое значение. Для разрешения описанных потенциальных проблем, можно вместо передачи и возврата по значению, передавать и возвращать данные по ссылке.

# Конструктор копирования

Конструктор копии позволяет управлять действиями, составляющими процесс создания копии объекта конструктор копии вызывается в трёх случаях:

1) Когда объект передаётся в функцию по значению.

2) Когда объект возвращается из функции по значению.

3) Когда объект создаётся и одновременно инициализируется содержим другого объекта.

Конструктор копии всегда имеет один аргумент, который является константной ссылкой на объект текущего класса.

MyClass(const MyClass& ob);

Каждый профессионально написанный класс, содержащий динамическое выделение памяти содержит конструктор копии, позволяющий избежать побочных эффектов, возникающих в результате создания побитовых копий объекта по умолчанию.

Как Java, так и C# динамически выделяет память для всех объектов, а программист оперирует этими объектами через ссылки. Необходимость создания конструктора копии отпадает, так как они создаются без участия разработчика.

# Механизмы наследования и определение собственных типов данных

Наследование (inheritance) это процесс создания новых классов, называемых наследниками или производными классами из уже существующих или базовых классов и **добавление** в создаваемый класс новых полей. Наследование позволяет реализовать принцип повторного использования кода (code reuse). Производный класс получает все возможности базового класса, но может быть усовершенствован за счёт собственных полей и методов. Базовый класс при этом остаётся **неизменным**, а дочерний класс модифицируется с учётом его специфики. Наследование, генерализация в нотации UML всегда отображается как стрелка вверх, которая указывает на базовый класс. Преимущество от использования наследования состоит в том, что наследование позволяет использовать существующий код несколько раз, т.е. имея написанный и отлаженный базовый класс, нет необходимости его модифицировать, а наследуя, приспособить для работы в других специфических ситуациях. Наследование также можно использовать при начальном описании системы, при разработке её общей структуры. Наследование позволяет эффективно распространять библиотеки классов.

Для того чтобы не тратить время на повторные тестирования модифицируемого класса или при невозможности модификации класса можно использовать наследование. Если не создан конструктор в производном классе, то будет использован подходящий конструктор базового класса.

# Спецификатор доступа protected (защищённый)

Методы производного класса имеют доступ к элементам базового класса, которые объявлены со спецификаторами public или protected. К элементам, объявленным в базовом классе как private, методы производного класса доступа не имеют. Элемент, объявленный как protected, будет доступен методам своего класса и методам любого производного класса. Недостатками использования спецификатора доступа protected являются, то что любой может получить доступ к элементам protected просто создавая производные класса, т.е. элементы, объявленные со спецификатором protected, являются менее защищёнными, чем элементы с атрибутом private. При наследовании базовых классов остаётся **неизменным,** что базовый класс ничего не знает о его производных классов. При создании объекта дочернего класса всегда сначала происходит вызов дочернего соответствующего конструктора, затем первого базового класса, затем второго базового класса и так далее, а завершает выполнение сначала конструктор первого базового класса, затем все остальные конструкторы базовых классов и только затем конструктор производного класса. При разрушении объекта дочернего класса деструктор вызывается в следующем порядке: сначала деструктор производного класса, затем деструкторы базовых классов в порядке обратном порядку наследования.

# Переопределение методов (overriding)

Можно создавать в производном классе методы имеющие такие же сигнатуры, как и в базовом классе. Такая ситуация называется переопределением метода.

При переопределении методов имена, аргументы и возвращаемые значения в базовом и производном классе должны быть одинаковыми. Существует правило, по которому определяется, какая реализация должна быть вызвана:

Для объекта производного класса будет выполнен метод производного класса, для объекта базового класса, так как ему ничего не известно об производном классе, будет выполнен метод базового класса. При наследовании атрибутом наследования по умолчанию является атрибут private. При создании дочернего класса с атрибутом доступа public, все поля и методы имеют такие же атрибуты, как и в базовом классе, приватные поля в дочернем классе недоступны. При наследовании с атрибутом доступа protected public и protected поля в дочернем классе становятся protected. При наследовании с атрибутом доступа private, в дочернем классе public и protected поля становятся private.

Если существует класс, который не содержит достаточное количество полей или методов для инстанциирования (создания объекта), такого рода класс называется абстрактным.

# Множественное наследование классов

Множественное наследование классов в C++ позволяет одному классу наследоваться **одновременно** от двух и более классов. C# и Java не поддерживают множественное наследование классов, но поддерживают множественное наследование интерфейсов.

# Неопределённость при множественном наследовании

При одновременном наследовании двух и более классов может возникнуть следующая ситуация: Если в базовых классах присутствуют методы с одинаковыми именами, то при вызове этого метода через объект дочернего класса возникает неоднозначность, какой метод должен быть вызван. Разрешение этой неоднозначности можно достичь за счёт указания имени класса, для которого вызывается метод. Разрешение этой неоднозначности можно достичь также за счёт использования ключевого слова virtual. Наследование не является единственной связью между классами, кроме связи наследования, которая описывает связь «является», есть другие типы связей, например, включение, которое описывает взаимоотношение «имеет или содержит». Включением называют взаимоотношением “часть целого» т.е. книга является частью библиотеки. Агрегация тоже описывает часть целого, но часть может существовать даже если целого нету, двигатель и автомобиль в СТО. Программная реализация композиции (одного целого) требует выделение памяти под «часть» в конструкторе «целого», программная реализация агрегации может предполагать наличие метода, который инициализирует поле «часть».

# Полиморфизм и его основные проявления, механизмы использования

Одним из принципов ООП является полиморфизм – свойство, которое позволяет одно и тоже имя использовать для решения двух и более схожих, но технически разных задач. В C++ полиморфизм представляет собой термин, используемый для описания процесса, в котором различные реализации метода могут быть доступны посредством одно и того же имени. Это означает, что ко всем методам класса, можно получить доступ одним и тем же способом несмотря на возможное различие в конкретных действиях, связанных с каждой отдельной реализацией дочерних классов. Полиморфизм делится на два вида:

1) Полиморфизм этапа выполнения (run time), к которому относится использование виртуальных функций.

2) Полиморфизм этапа компиляции, к которому относится перегрузка функций и операторов. (Перегрузка функций и операторов)

Полиморфизм этапа компиляции называется статическим. Полиморфизм этапа выполнения называется динамическим.

Указатель на базовый класс может ссылаться **на любой** объект любого дочернего класса. Фундаментом для динамического полиморфизма служит указатель на базовый класс. Указатели на базовые классы и на производные классов связаны такими отношениями, которые несвойственны указателям других типов. Обычно указатели одного типа не могут указывать на объекты другого типа, однако указатели на базовый класс и объекты производных класса исключение из этого правила. В C++ указатель на базовый класс может ссылаться на любой объект любого дочернего выведенного из базового.

Если необходимо с помощью указателя на базовый класс получить доступ к элементам, определённым в дочернем классе, то можно приведя тип указателя базового класса к типу указателя дочернего класса, вызвать методы дочернего класса. Хотя базовый указатель можно использовать для доступа к содержимому производного типа обратное утверждение неверно. Указатель на базовый класс «++» и «--» относительно своего базового типа. Следовательно, если указатель на базовый класс используется для доступа к объекту производного типа, инкрементирование или декрементирование не заставит его ссылаться на следующий объект производного типа. Вместо этого он будет указывать на следующий объект базового типа. Таким образом ++ или -- на базовый класс, следует расценивать как некорректную операцию, если этот указатель используется для ссылки на объект производного класса. Параметр, который имеет тип ссылки на базовый класс может принимать объект базового класса, также объекты любого другого типа, выведенного из него.

# Использование виртуальных методов

Динамический полиморфизм возможен только благодаря сочетанию двух средств:

1) Наследование

2) Виртуальные методы

Виртуальный метод объявляется в базовом классе с использованием ключевого слова virtual. Виртуальный метод может быть переопределён (переделан) в одном или нескольких производных классов. Решение о том, какая реализация метода должна быть выполнена решается во время выполнения программы и определяется типом адресуемого объекта, а не типом указателя.

Только при обращении к виртуальной функции через указатель на базовый класс достигается динамический полиморфизм. Виртуальная функция, переопределенная в дочернем классе, часто называется просто переопределённая. Виртуальная функция должна быть элементом класса, а не его другом. Прототипы виртуальной функции и её переопределений должны быть абсолютно одинаковы. Атрибут virtual передаётся по наследству т.е. если метод объявлен как виртуальный, то он остаётся таким независимо от того через сколько уровней иерархии он прошёл. Если производный класс не переопределяет виртуальную функцию, то используется её версия из базового класса. Необходимость использования виртуальных функций заключается в том, что программист может легко создать новый тип, который будет разделять общий интерфейс с другими родственными типами.

# Чисто виртуальные функции и абстрактные классы

Чисто виртуальная функция (pure virtual function) это виртуальная функция, которая не имеет определения в базовом классе. Если класс содержит чисто виртуальную функцию, то он становится абстрактным. Для абстрактного класса нельзя создать объект, но можно наследовать и создавать указатели. Чисто виртуальная функция **гарантирует,** что производный класс будет содержать все необходимые методы. Чисто виртуальная функция объявляется в следующем виде:

В базовом классе:

virtual void showArea()=0;

В производных класса, пишем реализацию.

Объявив функцию, чисто виртуальной, создаются условия, при которых производный класс **вынужден** иметь её собственную реализацию. Класс, который содержит хотя бы одну чисто виртуальную функцию будет являться абстрактным.

Раннее связывание (early binding) подготавливается во время компиляции, к нему относится перегрузка операторов и функций. Ранее связывание требует меньше аппаратных ресурсов. Позднее связывание (late binding) происходит во время выполнения программы, требует больше аппаратных ресурсов, также называется динамические полиморфизм к нему относится использование виртуальных функций.

# Использование шаблонов (templates)

Шаблон – средство языка c++, которое позволяет создать многократно использующийся код. Шаблонами могут быть как классы, так и функции. Шаблонная функция перегружает саму себя. Шаблонную функцию создают, когда имеется один общий алгоритм, отличающийся только типами обрабатываемых данных. Обобщённая функция имеет следующий вид:

template <class T> //T – имя параметра типа,

void swap (T&a)

{

T age;

}

swap (int a); //везде, где стоит T вместо типа, станет типом int

swap (double a); //везде, где стоит T вместо типа, станет типом double

Ключевое слово template обозначает, что начинается шаблонная функция или класс. Ключевое слово class или typename обозначает имя параметра шаблона. Имя шаблона представляет собой **заполнитель,** вместо которого компилятор автоматически подставит реальный тип данных при создании конкретной версии функции. Обобщённой функцию стоит делать тогда, когда имеется один общий алгоритм, отличающихся только разными типами данных. Поскольку c++ не распознаёт символ конца строки в качестве признака конца инструкции, то template часть определения обобщённой функции может не находится в одной строке с именем этой функции, но между ними не может быть никаких других инструкций. При создании обобщённой функции можно использовать столько параметров типа, сколько необходимо. Можно создать явную специализацию обобщённой функции, задав вариант её перегрузки для конкретного типа вручную. (функция сложения, для чисел всё ок, для строк нужна отдельная «специализация»). template <> часто пишутся перед специализацией шаблона функции, для того чтобы указать, что где-то есть шаблонный вариант этой специализации. Перегрузка шаблона функции возможна, когда имеется несколько вариантов реализации функции с разными шаблонами. При перегрузке шаблона функции будет играть значение только количество параметров. В шаблонных функциях, кроме параметров типа, могут использоваться стандартные параметры.

template <class T>

void func( X a, int b)

{

}

При перегрузке функции в теле каждой из них обычно задаются различные действия. Но обобщённая функция должна выполнять **одно и тоже действие** для всех её версий, отличие будет состоять только в типе данных.

# Обобщённые классы

При создании обобщённого класса, предполагается что все алгоритмы в классе будут одинаковыми, при этом реальный тип обрабатываемых в нём данных будет задан при создании объектов этого класса.

template <class T> class имя\_класса //T- заполнитель/параметр типа

{

};

При создании шаблонного класса можно указывать сколько хочешь параметров типа. В template спецификации для обобщённого класса можно задавать параметры, не являющиеся типами, у этих параметров есть важное свойство: не типовые параметры шаблона могут быть либо целочисленными типами, либо указателями, либо ссылками и всегда являются константами. Шаблонный класс может определить значение параметра типа по умолчанию.

template <class X=int>

class myClass

{

//

};

int main()

{

myClass<float> f;

myClass <> f;

}

Для класса тоже можно создать специализацию указав перед именем класса пустую спецификацию template, а после имени класса задав тип спецификации

template <> class myClass<int>

{

};

# Параметризация и наследование

ОПП языки позволяют 3 способа упорядочивания типа: наследование, агрегация, параметризация.