1. Особенности C++

Синтаксис ЯП C++ является C-подобным. Поэтому, все основные конструкции: ветвление, циклы, функции, синтаксически записываются точно также как в языке C.

C++ имеет те же примитивные типы данных, что и C, например, ***char, int, float, void*** и т.д.. Но в C++ добавлены типы ***double***, которые хранит вещественные числа в 64-битном формате, и ***bool***, предназначенный для хранения булевых значений ***true*** и ***false***. Также, к переменным можно принять модификатор ***long***, для увеличения количества отводимых бит для хранения значения, а также ***unsigned/signed***, которые определяют, может ли переменная иметь отрицательные значения.

Так как размер переменной в битах может варьироваться в зависимости от ОС и компилятора, то C++ введены типы данных фиксированного размера (fixed width), например ***int8_t*** под целочисленное 8-битное значение или ***uint64_t*** для целочисленного беззнакового 64-битного значения. Эти типы данных гарантировано будут занимать указанное количество бит.

Инициализировать переменные можно также как в C, используя оператор присваивания = (например, ***int value = 333***). Также можно инициализировать в функциональном виде, используя круглые скобки (например, ***int value(333)***), или с использованием фигурных скобок. В случае использования фигурных скобок, будет проверка на передаваемое значение, чтобы его тип точно совпадало с типом переменной. Поэтому, запись ***int value{1.11}*** вызовет ошибку компиляции, так как переданное значение имеет тип ***double***, а не ***int***. Ошибка будет даже в случаях, когда один тип приводится к другому.

Для чтения и записи значений из потока ввода/вывода можно использовать функции из языка C: ***scanf*** и ***printf***. Но C++ в модуле ***iostream*** предоставляет объекты ***std::cin***, для чтения из потока ввода, и ***std::cout***, для записи в поток вывода. Для передачи переменных в поток ввода и вывода, используются операторы >> и << соответственно. Основное отличие от функций языка C, что данные объекты позволяют автоматически определять тип переданного значения/переменной, и то как его надо выводить. Также, в модуле ***iomanip*** находятся функции манипулятора потока, которые позволяют форматировать вывод.

Как и в C, C++ позволяет работать с указателями, переменными, которые хранят адрес ячейки памяти размером указанного типа данных.

Для работы с указателями используются операторы & (позволяет получить адрес переменной в виде указателя) и * (позволяет разыменовать указатель, то есть получить значение, которое хранится в указанной ячейки памяти). Нулевому указателю (нулевой адрес) не соответствует никакая ячейка памяти, и попытке записи или чтения через нулевой указатель, возникнет runtime ошибка. Использование целочисленного значения 0 или использование макросса NULL (которое на самом деле прячет за собой значение 0) небезопасно, так как его можно присваивать не только указателям. Поэтому, в C++ введен тип данных специально для обозначения нулевого указателя ***nullptr_t*** и соответствующие значение этого типа данных ***nullptr***, которое можно присвоить только указателям.

Работа с указателями не всегда удобна, так код захламляется операторами & и *, необходимо проводить проверку на нулевой указатель, и быть уверенным, что адрес храний по указателю доступен. В C++ для случаев, когда надо работать с переменной, но использовать преимущества работы с переменной через указатели, например передавать в функции без копирования переменной, или делать разделяемое владение ресурсом, используются ссылочные переменные. Чтобы создать ссылку, нужно при объявлении переменной после типа данных добавить символ &. Ссылки можно рассматривать как обертку над указателем, переменная, которая хранит адрес, но при обращении к переменной, она будет автоматическа разыменована.

Основные различия ссылок и указателей:

1. Ссылка не может быть не инициализированной, то есть всегда ссылается на какую-то переменную. Переменную указателя можно создать, не назначая какой-либо адрес (в таких случаях стандартом не гарантируется, что будет записано в указателе).
2. Следствие из п.1. У ссылок нет нулевого значения в отличие от указателя.
3. Нельзя создать массив ссылок.
4. Нельзя переинициализировать ссылку. Если ссылка начинает ссылаться на какую-то переменную, то до конца жизни она будет хранить этот адрес. Указателю можно задать другой адрес.
5. Нельзя получить ссылку (адрес) на ссылку. При попытке взять адрес ссылки, будет взят адрес переменной к которой привязана ссылка.

Ссылка является неизменяемой как и константный указатель (не путать с указателем не константу), но можно сделать константную ссылку. Через такую ссылку нельзя будет изменять переменную, даже если сама переменная не константная. Константную ссылку можно создать при помощи таких записей: ***int& const*** или ***int const&***. Такие ссылки позволяют избегать лишнего копирования при передаче аргументов в функцию, и гарантируют, что передаваемые переменные не будут изменены изнутри функции.

Хоть ссылки и позволяют уменьшить количество работы с указателями в коде, но для работы с динамической памятью напрямую, всё равно используются указателя. Для выделения динамической памяти в C++ можно использовать функции из языка C, ***calloc/malloc*** для выделения памяти и ***free*** для освобождения памяти. Или можно использовать инструменты языка C++, оператор ***new*** выделения памяти, и оператор ***delete*** для очистки памяти. Данные операторы, в отличии функций языка C, не только выделяют память, но инициализируют значения в памяти.

Выделять память под одно значение можно следующим способом: ***int* value = new int(333)***. Это выделит в памяти ячейку под размер int и сразу запишет туда значение 333. Для очистки памяти нужно выполнить: ***delete value***. Оператор ***delete*** нормально обрабатывает нулевой указатель, но при попытке двойной очистки одной и той же памяти, может возникнуть ошибка ***SegFault***.

Выделение памяти под массив значений может быть произведен таким способом: ***int* arr = new int[n]***. В таком случае число n определяет кол-во элементов массива. Для удаления в таких случаях нужно писать: ***delete [] arr***.

2. ООП - Объектно-ориентированное программирование

Парадигма программирования определяет концепции того, как рассматривается программа в целом, а также определяет стиль написания кода. Существуют различные парадигмы программирования, но самые часто-встречаемые это: структурная, функциональная, объектно-ориентированная.

В структурной парадигме произошел отказ от инструкции goto, программа рассматривается как комбинация трех блоков: последовательное выполнение, ветвление и циклы. Блоки могут быть вложены в друг друга. Такие программы можно описать в виде блок-схемы. И считается, что в программу один выход и выход.

В функциональной парадигме программа рассматривается как композиция более простых функций. Вводятся понятия функции высших порядков (функции, которые принимают в качестве аргументов или возвращают в качестве результата другую функцию), чистые функции (функции, которые зависят только от аргументов, и не зависят от состояния программы), иммутабельность переменных, монады. Также в функциональной парадигме. В настоящее время, различные концепции функционального программирования используются в различных ООП языках.

В объектно-ориентированной парадигме программа рассматривается как взаимодействие различных объектов между собой. Используется моделирование информационных объектов, например, можно описать объект реального мира (например, студента), описать процесс (например, перемещение объекта из точки А в точку Б), описать абстракции (например, базу данных, которая представлена в виде каких-то структур данных, но в реальном мире она физически не существует в таком виде). Объекты обычно выстроены в иерархию, из-за чего выстраивается иерархическая структура программы, где наверху иерархии обычно находится объект представляющий саму программу (с точки зрения бизнес-логики). Для взаимодействия объектов введена концепция отправки сообщений, то есть объекты “общаются” между собой. В самом базовом уровне объекты общаются путем того, что один объект инициирует действие другого объекта. Для описания структуры и действий объектов используется понятие класса. Внутри классов, для описания самих действий используется структурная парадигма.

Основными принципами ООП являются:

- *Абстракция* - рассмотрение только значений информации предметной области, и исключение незначимой.
- *Инкапсуляция* - объединение данных и методов, работающих с этими данными, в классе.
- *Наследование* - свойство системы, позволяющее описывать новый класс на основе существующего.
- *Полиморфизм* - использование объектов с общим интерфейсом без информации о внутренней структуре объекта. Интерфейс в данном случае, это набор действий, которые можно выполнить над объектом, или, которые объект может выполнить сам).

Основные понятия ООП:

- *Класс* - единица абстракции. Универсальный тип данных, являющийся информационной моделью.
- *Объект* - экземпляр класса, хранящийся в памяти.
- *Поля данных (также, атрибут)* - параметры (свойства) объекта определяющие его состояние. Переменные, объявленные как принадлежащие классу.
- *Методы* - функции связанные с классом. Определяют действия объекта. Множество методов определяют интерфейс класс.
- *Модификаторы доступа* - механизм определяющий доступность полей и методов объекта.

Одним из преимуществ ООП, это объединение данных. В C++, как и в C, можно использовать структуры, чтобы синтаксически и физически сгруппировать логически связанные данные. Для объявления структуры используется ключевое слово ***struct***. Вот пример (листинг 1) создания структур описывающих точку и отрезок состоящий из двух точек, а также сигнатуры функций, принимающие объекты данных структур в качестве аргументов (функция ***length*** расстояние между двумя точками, ***intersect*** определяет пересечение двух отрезков).

Листинг 1.

```
struct Point{
  double x;
  double y;
};
struct Segment{
  Point p1, p2;
};
double length(Point p1, Point p2);
bool intersect(Segment seg1, Segment seg2, Point *p);
```

В качестве полей структуры можно указать любой тип данных, кроме самой структуры, так как это создаст неполный тип данных (неизвестно, сколько памяти занимает объект структуры), и соответственно, такой объект невозможно создать. Но структура может иметь поля, которые являются ссылкой или указателем, на эту же структуру.

В отличие от C, в C++ внутри структуры можно определить функции. Такие функции называются методами. Методы имеют доступ ко всем

полям структуры, и могут с ними работать, даже если не передаются в качестве аргументов. Это можно рассматривать как глобальные переменные в области видимости внутри структуры. Как было сказано выше, методы определяют интерфейс. В листинге 2 пример, где функция расчета длины отрезка внесена в структуру Segment, и теперь ей не надо принимать аргументы, так как координаты точек, итак уже хранятся внутри структуры. Также, внесена функция поиска пересечения двух отрезков. Так как информация об одном отрезке уже содержится в структуре, то достаточно принять объект другой отрезок, а в качестве результата можно вернуть указатель на ***Point***, и вернуть ***nullptr***, в случае, если отрезки не пересекаются. Таким образом можно описывать необходимо сущности из предметной области, и это называется абстракцией.

Листинг 2.

```
struct Segment{
  Point p1, p2;
  double length(){} //аргументы берутся из структуры
  Point* intersect(const Segment& seg){} //аргументом принимает другой
отрезок
};
```

В C++ основное различие между структурами и классами, это в доступе к полям по умолчанию. В соответствии с формальным синтаксисом C++, объявление структуры создает тип класса, и структуры в C++ по большей части сохранены для совместимости с C. Обычно принято, что структуры используются для объединение переменных (данных) в единое целое, а классы используются, если необходимо добавить методы для работы с данными.

В рассмотренных выше примерах, к полям структуры можно обращаться напрямую, и получается, что нет контроля над ними, и можно изменять их как угодно. Например, можно поставить такие значения, которые нарушают логику структуры. Для защиты данных структуры и классов от внешнего воздействия, можно применять модификаторы доступа. Всего существует 3 модификатора:

- **public** - доступ имеют все, кто видит определение структуры
- **protected** - доступ закрыт для всех, кроме самого класса и его наследников
- **private** - доступ закрыт для всех, кроме самого класса

По умолчанию, у класса все поля ***private***, а у структур наоборот ***public***. Модификаторы доступа применимы как для класса, так и для структуры. И для доступа к закрытым полям обычно используются методы называемые геттер и сеттер, соответственно получить доступ, и записать значение, такие методы позволяют контролировать доступ, например проводить проверку на валидность передаваемого значения. Порой логика класса может быть такова, что доступ к закрытым полям может быть только у самого класса, и никто не должен иметь возможность узнать значение поля или изменить поле, в таких случаях геттер и/или сеттер может отсутствовать. Это и есть инкапсуляция, когда данные спрятаны, и методы, которые должны работать с этими данными, помещены в тот же класс.

Синтаксически, класс описывается также как и структура, только вместо ключевого слова ***struct*** используется ключевое слово ***class***. В листинге 3 приведен класса, описывающий линейную функцию. Он содержит два поля ***private*** (коэффициенты линейной функции ***a*** и ***b***), одно поле ***protected*** (строка, которая хранит название), и два метода по установке коэффициентов лин. ф-ции и расчет значения функции в точке, которые объявлены с модификатором ***public***. Для обозначения модификатора доступа пишется ключевое слово нужного модификатора и ставится двоеточие, далее, все поля и методы будут иметь этот модификатор доступа, пока не будет прописан другой модификатор в рамках этого класса.

Листинг 3.

```
class LinearFunction{
private:
  double a = 0.0;  double b = 0.0;
protected:
  std::string name = "Linear function";
public:
  void setCoefficients(double new_a, double new_b){a = new_a; b = new_b;}
  double calculate(double x){return a*x + b;}
};
```

Методы класса можно, и даже рекомендуется, определять вне класса, для этого в самом классе остается только сигнатура метода, а потом после класса пишется реализация метода. Но так как, чтобы различать, какой метод принадлежит к какому классу (ведь в двух разных классах, могут

быть методы с одинаковой сигнатурой), то в начало названии метода вне класса, прописывается название класса, и между названием класса и метода ставится оператор ::. Таким образом, для метода ***setCoefficients*** из листинга 3 полная сигнатура будет выглядеть так
void LinearFunction::setCoefficients(double new_a, double new_b) . И по аналогии с обычными функциями, как в языке С, можно разделить код на заголовочный файл (расширение .h или .hpp) и файл с реализацией (расширение .cpp). Таким образом в заголовочном файле остается только объявление класса и его методов, а в файле с реализацией находится реализация его методов. Это является хорошей практикой, так как улучшает читаемость кода, и влияет на процесс компиляции.

Так как C++ является компилируемым языком, который преобразуется до языка ассемблера, а в ассемблере нет классов, то все методы преобразуются до обычных функций, и добавляет первый аргумент, который содержит указатель на объект класса. В самом C++, этот указатель не добавляется явном виде, но доступ к нему есть. Называется неявный указатель ***this***, этот указатель есть у каждого объекта, и хранит адрес объекта, и имеет тип указателя на класс объекта. К нему можно обращаться в любом методе, и этот указатель может использоваться для разрешения проблемы перекрытия имен, или чтобы объект мог сам себя передать в какую-то функцию/метод.

Можно создавать константные методы, для этого в сигнатуре метода после списка аргументов и до фигурных скобок (тела метода) нужно поставить ключевое слово ***const***. Такие методы не могут никак изменять состояние объекта (если попытаться изменить, то будет ошибка компиляции). Неявный указатель ***this*** в таких методах преобразуется до указателя на константу. У константных объектов, можно вызывать только константные методы. А также, ключевое слово ***const***, является частью сигнатуры метода, что позволяет проводить константную перегрузку методов. Это когда есть два идентичных по названию и аргументам метода, но один из них константный. Эти методы могут иметь разное тело, что позволяет определить поведение когда метод вызывается у константного и не константного объекта. Например, при реализации класса контейнера, и можно определить две версии получения элемента, в обычной версии метод возвращает ссылку на элемент, а константная версия вернет копию элемента.

3. Создание и уничтожение объектов

В рассмотренных ранее примерах при создании объекта, его поля инициализировались значениями по умолчанию, и затем через методы задавались необходимые значения для полей. Такой подход опасен, так как между созданием объекта и вызовом метода, объект может находится в некорректном состоянии, и потенциально его кто-то попытаться использовать.

Поэтому, для определения способа инициализации объекта класса, используется специальный метод, называемый конструктором. Имя конструктора совпадает с именем класса, и конструктор не имеет возвращаемого значения. Аргументы могут быть любые, и принимают информацию, на основе которой происходит инициализации полей. Например, можно передать число, которое определит, какой объем памяти нужно выделить под динамический массив, хранящийся в классе, и в самом конструкторе уже выделить память нужного размера. Также конструкторы можно перегружать, и таким образом по-разному инициализировать объект, в зависимости от типа/количества переданных аргументов.

Так как, сначала выделяется память под объект, и только затем выполняются действия прописанные в теле конструктора, и в этот момент, поля уже обладают каким-то значениями по умолчанию. Иногда это может приводить к проблемам, например, если для поля класса нет значения по умолчанию, или если поле константное, и его уже нельзя изменять. Для того, чтобы задавать значения полям в момент выделения памяти, используются списки инициализации. Для их записи нужно после списка аргументов конструктора и до его тела поставить двоеточие, и инициализировать поля класса (листинг 4). В списке инициализации можно записать инициализацию в любом порядке, но они всё равно будут инициализировать в том порядке, в котором они объявлены в классе.

Листинг 4.

```
class LinearFunction{
private:
  const double a = 0.0;
  const double b = 0.0;
public:
  LinearFunction(double a, double b):a(a),b(b){ //Перекрытия имен не будет
  }
};
```

Если в классе не реализовать никакого конструктора, то компилятор автоматически создаст конструктор, у которого нет аргументов и нет тела, и объект всё равно можно будет создать. Такой конструктор без аргументов называется конструктором по умолчанию, и его можно реализовать по своему. В таком случае, это определить процесс создания объекта по умолчанию, то есть когда объявляется переменная, которой не задается никакое значение.

Иногда есть необходимость в пустом конструкторе по умолчанию, который бы создал компилятор, но в классе уже есть другие конструкторы. В таком случае можно создать самому такой конструктор с пустым телом, либо использовать ключевое слово default, которое явно говорит компилятору, создать конструктор по умолчанию. Например для класса ***Segment***, создание конструктора по умолчанию выглядело бы следующим образом ***Segment() = default;***.

Также, может быть ситуация, что перегрузке конструкторов, их тела идентичным, или имеют большой количество дублирующий логики. Чтобы уменьшить количество дублирующего кода, можно какие-то конструкторы сделать делегирующими. Это такое конструкторы, которые в списке инициализации, вместо инициализации полей, вызывает другой конструктор этого же класса. Таким образом, можно создать один общий конструктор, в котором находится вся логика общая для всех конструкторов, и более специфичные конструкторы вызывают общий, а затем выполняют свое тело.

Помимо контроля создания объекта, можно и контролировать процесс его уничтожения (когда он выходит из области видимости, или в следствие вызова оператора ***delete***). За уничтожение объекта отвечает специальный метод называемый деструктором. Этот метод не имеет возвращаемого значения, его имя совпадает с именем класса с добавлением символа ~ в начало, и у него нет аргументов. Также как и конструктор по умолчанию, компилятор создаст деструктор, если его нет. И в отличии от конструкторов, деструктор только один, и его нельзя перегрузить. Основная задача деструктора, это освобождение ресурсов, когда объект уничтожается, например, закрытие файла или освобождение динамически выделенной памяти.

Если в классе хранятся какие-то ресурсы, например, динамически выделенная память, то недостаточно корректно и выделять и удалять потом ее в деструкторе. Проблемы могут произойти при копировании или

перемещения объекта, так как скопируется указатель, и два объекта будут ссылаться на одну область памяти, и после того, как один из объектов будет уничтожаться, то он освободит память, и для второго она станет недоступна, либо один объект меняет значение в памяти, и эти изменения будут видны у второго, что может противоречить логике программы. Это происходит, потому что по умолчанию в C++ происходит поверхностное копирование и перемещение.

Копирование, это процесс, при котором создается объект А идентичный объекту Б, причем объект Б остается неизменным, и объекты А и Б независимы друг-друга, то есть изменение одного объекта, не должно нарушать корректность другого объекта.

Перемещение, это процесс, при которым создается объект А путем “получения” всех характеристик объекта Б. При чем, объект Б может оказаться в неопределенном состоянии. Обычно это происходит, когда объект Б должен быть уничтожен, после перемещения его в объект А.

В языке C++, все выражения (просто переменная также выражение) делятся на категории. И одна из характеристик категорий, можно ли копировать или перемещать значения/переменные.

Категория **lvalue** обозначает, что выражение обладают идентичностью, то есть для двух таких выражений можно определить, ссылаются ли они на одну и ту же область памяти. И такие выражения можно копировать. Например, это просто переменная, или результат функции в виде ссылки. Но такие выражения нельзя перемещать

Категория **rvalue** означает, что выражение обладает идентичностью, но должно быть сейчас уничтожено, например, результат функции, которые должен быть передан сразу как аргумент другой функции. Либо у такого выражения нет идентичности, например просто литерал **true**, который не был записан ни в какую переменную. В обоих случаях, их можно перемещать.

Также есть простое правило, что **lvalue** может слева от знака =, а **rvalue** нет. На уровне синтаксиса языка C++, их можно различать при помощи соответствующих ссылок. Ссылки, которые были рассмотрены ранее, это **lvalue** ссылки, но существуют и **rvalue** ссылки, которые записываются с двумя &&, и выглядят как ссылка на ссылку. **lvalue** ссылка может ссылаться только на **lvalue**, **rvalue** ссылка только **rvalue**, а константная **lvalue** ссылка может ссылаться на **lvalue** и **rvalue**. В случае второго, время жизни **rvalue** будет продлено, пока существует ссылка.

Это позволяет перегружать функции таким образом, что для разных категорий они будут вести себя по разному. А соответственно, можно создать такие конструкторы, которые принимают другой объект того же класса по **lvalue** ссылки и по **rvalue** ссылке. В первом случае, конструктор будет конструктором копирования, и должен иметь сигнатуру ***Classname(const Classname&)***, и соответствующий оператор присваивания с копированием будет иметь сигнатуру ***Classname& operator = (const Classname&)***. Во втором случае это будет конструктор перемещения с сигнатурой ***Classname(Classname&&)***, и соответствующий оператор присваивания с перемещением с сигнатурой ***Classname& operator = (Classname&&)***. Перемещение в C++ основывается на move-семантике. Функция **std::swap** позволяет обменять значения двух переменных, и при перемещении объекта Б в объект А, их поля можно просто обменять, так как объект Б скорее всего будет уничтожен, а значит он очистит все ресурсы у себя в деструкторе. Функция std::move позволяет явно обозначить перемещение, и по сути, она преобразует **lvalue** в **rvalue**.

Существует правило 5, если реаливано что-то из списка: конструктор копирования, оператор присваивания с копированием, конструктор перемещения, оператор присваивания с перемещением, деструктор; то нужно реализовать всё остальное из списка, так как скорее всего внутри класса используются ресурсы, с которыми нужно правильно обращаться.

Также, можно запретить объект класса копировать и/или перемещать, для этого используется ключевое слово ***delete*** по аналогии с ключевым словом ***default***.

4. Наследование

Наследование один из основных механизмов ООП, так как на нем основывается полиморфизм (динамический). Наследование позволяет создать новый класс на основе существующего, причем новый класс перенимает всего поля и методы существующего, и может расширить функционал, либо видоизменить. В терминологии такой новый класс называют производный/дочерний/подкласс, а существующий базовый/родительский/суперкласс.

В C++ для того, что класс наследовался от другого, у наследника после имени класса через двоеточие нужно перечислить названия классов от которых происходит наследование с указанием модификаторов доступа (листинг 5).

Листинг 5.

```
class Base1{};

class Base2{};

class Deriv: public Base1, public Base2{};

class DerivDeriv: public Deriv{};
```

Объект производного класса можно приводить к базовому типу, так как в производном классе содержатся все поля, которые есть у базового, в том же порядке. Но, если приводим не к ссылке или указателю, то данные, которые есть только у производного будут утеряны, поэтому так делать не рекомендуется, но приводить к ссылке или указателю базового не вызывает таких проблем, и такое преобразование является часть полиморфизма.

При наследовании необходимо указывать модификатор доступа, обычно указывается **public**. Данные модификаторы могут изменить доступ полей и методов, всё приводится к самому ограничивающему модификатору. Например **public** метод базового класса при protected наследовании станет **protected**, а **protected** и **private** методы не поменяют доступ. Также модификатор доступа при наследовании вводит ограничение на приведение типов.

При создание объекта производного класса, объект создает по цепочке “снизу-вверх”, то есть сначала будут вызваны конструкторы базовых классов, начиная с самого базового, потому конструкторы полей производного класса, и только потом сработает тело конструктора производного класса. Деструкторы будут вызываться в обратном порядке.

Если конструктор базового класса должен принимать аргументы, то его обязательно необходимо вызывать в списке инициализации производного класса, так как конструкторы и деструкторы не наследуются.

При наследовании, производный класс может вызывать **public** и **protected** методы базового класса. Извне можно вызывать только **public** методы базового и производного класса, причем, используя оператор :: можно явно указывать, класс в цепочке наследования, чей метод нужно вызвать.

Переопределение метода, это создание в производном классе метода, у которого сигнатура полностью совпадает с методом базового класса. За

счет этого можно менять поведение базового класса. То, какой метод вызывать у объекта, компилятор смотрит сначала наличие подходящего метода в производном классе, если не находит, то поиск производится в базовом классе. Это является статическим полиморфизмом, так как выбор метода определяется на этапе компиляции.

Основная проблема с переопределением методов заключается в том, что если в производном классе создать метод совпадающий по имени с методом базового класса, но с другими аргументами, то компилятор увидит, что нужного метода нет в производном классе, и не будет искать нужный метод в базовом, даже если таковой имеется. Для решения этой проблемы можно использовать ключевое слово **using**, и в производном классе после ключевого слово **using** указать метод, который нужно “подтянуть” в производный класс. Нужно учитывать, что “подтянутые” методы будут иметь тот модификатор доступа, в котором было прописано **using**.

При уничтожении объектов при наследовании, надо учитывать, через объект какого типа данных производилось уничтожение объекта. Так как базовый класс не знает ничего про производные, в том числе и про деструктор, то есть было приведение объекта производного класса к базовому, то деструктор производного класса не сработает. Что чревато, например, если в производном классе выделялась динамическая память, то в таком случае будет происходить утечка памяти.

Для того, через тип базового класса можно было использовать методы объекта производного класса нужно использовать виртуальные методы. Виртуальные методы, это такие методы класса, которые предполагается, что они будут переопределены. Для того, чтобы метод сделать виртуальным, нужно в базовом классе, для нужного метода перед возвращаемым типом добавить ключевое слово **virtual** (в производных класса это ключевое слово можно не указывать, виртуальность распространится дальше по цепочке наследования). Также можно помечать деструктор виртуальным, и тогда не будет проблемы, описанной ранее.

Виртуальные методы - основной механизм динамического полиморфизма. А класс, в котором есть хотя бы один виртуальный метод называется полиморфным. Объекты полиморфных классов хранят у себя указатель на таблицу виртуальных методов (**vtable**). За счет это таблицы, происходит выбор метода для вызова во время работы программы, а не компиляции. Даже если объект производного полиморфного класса был

приведен к базовому, указатель **vtable** сохраняет свое значение, и будет вызываться метод, который находится в таблице, то есть метод производного объекта.

Если в базовом классе у виртуального метода убрать определение, вместо тела после списка аргументов указать ***=0***, то такой метод называется чистый виртуальный метод. Это предполагает, что данный момент будет определен позже, и каждый производный класс может иметь свою версию.

Если в классе есть хотя бы один чистый виртуальный метод, то такой класс называется абстрактным. Такие классы предназначаются, для хранений общей реализации и поведения некоторого множества дочерних классов. За счет этого, уменьшается дублирование кода. Объекты абстрактного класса нельзя создать, но можно создать ссылку или указатель на такой класс, чтобы туда подставлять производные классы. И даже, учитывая, что объект абстрактного класса нельзя создать, всё равно рекомендуется создавать виртуальный деструктор для такого класса.

Так как в C++ нет специальной синтаксической конструкции для создания классов-интерфейсов (не путать с интерфейсом из начала лекции), то для создания таких классов приняты правила, что интерфейс обладает следующими свойствами:

- все члены класса имеют модификатор доступа **public**
- содержит только чистые виртуальные методы
- не содержит никаких полей-данных

Каждый интерфейс является абстрактным классом, но не каждый абстрактный класс интерфейс. Интерфейсы нужны для того, чтобы описать поведение (то какие методы можно вызывать), но конкретную реализацию определять производные классы, которые реализуются данный интерфейс (когда наследуются от интерфейса, то говорят, что его реализуют). Так как C++ поддерживает множественное наследование, в отличии от многих других ООП языков, то рекомендуется наследоваться только от класса, но реализовывать можно сразу несколько классов.

В листинге 6 приведен пример кода, где создан интерфейс ***IPrinter***, с методом ***print***, который предполагается, что будет выводить целое число. Далее этот интерфейс реализован в классах ***TerminalPrinter*** и ***FancyPrinter***, первый просто выводит в терминал, а второй выводит с добавлением префикса заданного через конструктор. Далее, функция ***client*** пользуется только интерфейсом. Данная функция считывает два целых

числа и через интерфейс из выводит куда-то, куда функция не знает. Это будет зависеть то того, какой производный класс будет передан в функцию. Листинг 6.

```
#include<iostream>
#include<string>

class IPrinter{
public:
  virtual void print(int value) = 0;
};
class TerminalPrinter: public IPrinter{
public:
  void print(int value){
    std::cout << value << '\n';
  }
};
class FancyPrinter: public IPrinter{
  std::string prefix;
public:
  FancyPrinter(std::string str):prefix(str){}
  void print(int value){
    std::cout << prefix << ' ' << value << '\n';
  }
};

void client(IPrinter* printer){
  int x,y;
  std::cin >> x >> y;
  printer->print(x+y);
}

int main(){
  TerminalPrinter simple_printer;
  FancyPrinter fancy_printer("Sum is");
  client(&simple_printer);
  client(&fancy_printer);
}
```

Таким образом, можно динамически задавать поведение программы, выбор того, какой объект создать может происходить в условном операторе, или нужный объект можно достать из контейнера по нужному ключу. Если понадобится реализовать новый способ вывода, то достаточно создать новый класс, который реализует интерфейс, и код существующих

классов, а также функции клиента никак не изменяется. Единственное, всегда есть точка в коде, где создается объект конкретной реализации интерфейса, в данном случае это функция ***main***. По аналогии можно было бы для клиента создать интерфейс, откуда он получает два целых числа, и создать разные реализации, например, из терминала и из файла.

Интерфейсы также применяются в технике называемой “Внедрение зависимости”. Это такая техника, при которой объект настраивается внешними сущностями, а не “самонастройкой”. То есть, когда часть логики класса выносится из него, создаются интерфейсы для работы с вынесенной логикой, и класс у себя хранит и работает через интерфейсы. А конкретные реализации логики передаются в конструктор при создании объекта. Например, если есть класс, который должен достать из базы данных изображение, сжать его, зашифровать, и отправить его куда-то на сервер. Если вся логика была бы прописана в самом классе, то код был бы плохо масштабируемым, и его сложнее тестировать. Поэтому, для всех этих шагов можно создать интерфейсы, чтобы этот класс работал только с ними, и при создании объекта, в конструкторе передать конкретные реализации работы с БД, алгоритма сжатия, алгоритма шифрования, и отправки на сервер. И в таком случае, если меняется алгоритм шифрования, то создается новая реализация, и при создании объекта в конструктор передается новый алгоритм. При этом, это изменение будет ровно в одной точке, сам класс даже не придется менять, и замена одной части общего алгоритма, никак не затрагивает другие части. Также и облегчается тестирование, так как порой нельзя протестировать части в отдельности, и в данном случае, можно проверить, что сжатие и шифрование работает корректно, заменив на время тестирования загрузку из БД загрузкой локального файла, а отправку на сервер сохранением на локальный диск.