

ООП

Содержание курса

Требования.....	4
Обработка исключительных ситуаций.....	5
Создать исключение.....	7
В catch можно использовать данные нескольких типов.....	7
вложенные блоки try-catch.....	8
Задание ограничений на исключения.....	9
Повторная генерация исключений.....	9
Достоинства и недостатки.....	10
Декомпозиция.....	11
Как выделять объекты?.....	11
Операции.....	12
Начало ООП.....	13
Развитие парадигм программирования и языков программирования.....	13
Основные понятия ООП.....	16
Преимущества АТД.....	16
***.....	16
Основные принципы ООП.....	17
Отношения между объектами.....	18
Связи (link).....	18
Отношения между классами.....	18
Преимущества и недостатки ООП.....	18
Классы в C++.....	19
Указатель this.....	21
Инкапсуляция.....	22
Манипуляции с состоянием объекта.....	22
Конструктор и деструктор.....	22
Объявление и определение методов класса. Спецификатор inline.....	26
Инициализация, конструкторы, оператор присваивания.....	27
Прямая инициализация (direct initialization).....	28
Списки инициализации (braced initializer lists).....	29
Конструкторы по умолчанию (Default constructor).....	29
Явные конструкторы (Explicit Constructor).....	31
Конструктор преобразования (Conversion constructor) и конструкторы с двумя и более параметрами.....	33
Конструктор копирования (copy constructor).....	34
Метод swap.....	36
Конструктор перемещений (move constructor).....	37
Конструктор копирования и операция присваивания.....	38
Операторы присваивания.....	39
Оператор присваивания перемещением (move assignment operator).....	39
Оператор присваивания копированием (assignment operator).....	40
Правило трёх (Закон Большой Тройки) → Правило пяти.....	41
Спецификаторы default и delete.....	42

Порядок вызова конструкторов и деструкторов.....	44
Вопросы.....	46
Статические переменные и функции.....	47
Статические члены класса.....	47
Константные члены класса. Модификатор const.....	51
Друзья классов.....	53
Наследование.....	56
Преобразования.....	57
Модификаторы доступа.....	58
Перекрытие имен (overriding).....	59
Множественное наследование.....	61
Наследование vs агрегация.....	61
Полиморфизм.....	63
Статический полиморфизм.....	64
Перегрузка операторов (Operator overloading).....	64
Рекомендации.....	66
Примеры.....	66
Указатели на методы.....	68
Виртуальные методы.....	70
Виртуальный деструктор.....	70
override и final.....	71
Виртуальные базовые классы.....	71
Интерфейсы.....	71
Раннее и позднее связывание.....	72
Динамическая информация о типе (RTTI).....	73
Динамическая информация о типе (RTTI).....	78
ООА и ООР.....	78
UML.....	79
Различия между композицией и агрегацией.....	79
Мощность отношений (Кратность).....	79
Использование стандартных исключений.....	81
Qt Quick и QML.....	81
Рассмотреть.....	81
Лабораторные работы.....	82
Задание. Диаграмма классов.....	82
Задание. Реализовать диаграмму классов на C++.....	82
Задание. Класс «матрица».....	82
Задание.....	83
Работа с Git.....	83
Семестр II.....	83
Задание.....	83
Игра «Жизнь» Конвея, Сапёр, Тетрис, свой вариант.....	83
Задание.....	83
Задание.....	83
Задание 3. Калькулятор.....	84
Темы.....	84
Задание.....	85
Задание.....	85
Бонус.....	85

Задание.....	85
Задание.....	85
Задание.....	86
Задание.....	86
Экзамен 1.....	86
Экзамен 2.....	86
Вопросы.....	87
Вопрос.....	87
Вопрос.....	87
Экзамен.....	88
Литература.....	90
Литература для РП.....	91
Основная литература.....	91
Дополнительная литература.....	91
Источники.....	92
Курсы.....	92
Некоторые статьи.....	92
Рекомендованное ПО и онлайн-сервисы.....	92
IDE.....	92
***.....	92
Добавить.....	94
Установка IDE и т. д.....	94

Требования

К студенту.

- Знание C++, Java? C#?

Компьютерный класс

- Visual Studio
- CodeBlocks
- QtCreator
- g++
- Qt (второй семестр?)

Обработка исключительных ситуаций

exception handling

<http://www.c-cpp.ru/books/obrabotka-isklyucheniya>

<http://www.codenet.ru/progr/cpp/Try-Catch-Throw.php>

<https://habrahabr.ru/sandbox/28877/>

Ошибки

- ошибки времени компиляции
- ошибки времени выполнения
 - логические ошибки \неправильный алгоритм\
 - ошибки внешних данных \программа получила некорректные данные, строку вместо числа\

- это механизм языков программирования, предназначенный для описания реакции программы на ошибки времени выполнения и другие возможные проблемы (*исключения*), которые могут возникнуть при выполнении программы и приводят к невозможности (бессмысленности) дальнейшей отработки программой её базового алгоритма.

Исключение можно обрабатывать не там где оно возникло, а в любом месте стека вызовов. Это может быть полезно, потому что на данном уровне стека вызовов (алгоритма) может быть не понятно, что делать если возникло исключение.

Когда возникает исключения создаётся объект его описывающий. «Раскрывая стек» этот объект передаётся от нижнего уровня — к верхнему.

Виды исключений <https://prog-cpp.ru/cpp-exception/>

- аппаратные
- программные

Обработка исключений с помощью кода возврата

```
long DoSomething()
{
    long *a, c;
    FILE *b;
    a = malloc(sizeof(long) * 10);
    if (a == NULL)
        return 1;
    b = fopen("something.bah", "rb");
    if (b == NULL) {
        free(a);
        return 2; }

    fread(a, sizeof(long), 10, b);
```

```

if (a[0] != 0x10) {
    free(a);
    fclose(b);
    return 3; }

fclose(b);
c = a[1];
free(a);
return c;
}

```

Далее требуется в вызывающей функции, в месте вызова DoSomething описать обработчик. В нём по возвращаемому значению будет определяться исключительная ситуация.

Возникает три проблемы:

- требуется обработать исключение функцией находящейся на некотором расстоянии в стеке вызова от той, где возникает исключение. Придётся писать обработчик в каждой функции между ними.

- тип возвращаемого значения трудно приспособить для обозначения исключения. Приётся возвращать данные из функции через параметр или усложнить тип возвращаемого значения.

- для каждой ИС нужен свой код возврата и обработчик в месте возникновения.

Два подхода к обработке исключений

- исключения
- код возврата

Ловушка исключений

```

try {
    защищенный блок кода
    ... тут может возникнуть исключение ...
    ... в любом месте ...
    ... любого вида ...}
catch (тип переменная) {
    обработчик исключения
    код обрабатывающий исключение }
catch (тип переменная) { // обработка остальных исключений }
catch (тип переменная) { // обработка остальных исключений }
catch (...) { // Поймать все исключения }
// остальной код

```

Компиляторы могут (и часто делают) распознавать код, который создаёт исключительные ситуации, например деление на ноль. В результате этот не попадёт в исполняемый файл. Кроме того, компилятор может игнорировать переменные которые после присваивания им значения не используются.

Стоит отметить, что C++ не включает блок `finally` в этой конструкции [[stackoverflow](https://stackoverflow.com)]. Он призван содержать код который выполнится в любом случае, не зависимо от того возникла ли исключительная ситуация или нет. В C++ деструкторы вызываются автоматически, поэтому необходимости в отдельном описании кода деинициализации нет. Этот блок присутствует в C#, PascalABC, Java, Python и др.

Создать исключение

`throw значение` — создать\кинуть\возбудить исключение. Исключение может быть любого типа, в одном из простых случаев `int`. Этот оператор всегда должен быть внутри блока `try`, иначе исключение будет обработано стандартным обработчиком.

Операторы следующие за `throw` никогда не выполняются.

Обработка исключений, возбужденных оператором `throw`, идет по следующей схеме:

1. Создается статическая переменная со значением, заданным в операторе `throw`. Она будет существовать до тех пор, пока исключение не будет обработано. Если переменная-исключение является объектом класса, при ее создании работает конструктор копирования.
2. Завершается выполнение защищенного `try`-блока: раскручивается стек подпрограмм, вызываются деструкторы для тех объектов, время жизни которых истекает и т.д.
3. Выполняется поиск первого из `catch`-блоков, который пригоден для обработки созданного исключения.

Пример

```
#include <iostream>
using namespace std;
int main()
{
    // Простой пример, замена кода возврата
    // Теперь нет необходимости в написании отдельного обработчика исключительной
    // ситуации с использованием return
    // Тем более, что такой обработчик придется написать в каждой функции всей
    // иерархии вызовов
    try {
        cout << "Exception: "; // будет напечатано в любом случае
        // сгенерируем исключение. Здесь вместо 1 можно использовать заранее
        // оговоренные типы данных
        // чтобы описать исключительную ситуацию.
        throw 1;
        cout << "No exception!"; // не выполнится
    } catch (int a) { // в этом разделе, в зависимости от вида
        // исключительной ситуации реагируем на ней соответствующим образом
        if ( 1 == a ) {;}
        if ( 2 <= a && a <=10 ) {;}
        cout << "There is exception!!!" << endl;
        cout << "Код ошибки: " << a;
    }
    cin.get(); return 0;
}
```

В catch можно использовать данные нескольких типов

В зависимости от типа брошенного значения, будет выполнен код в соответствующем блоке `catch`.

```
try {
    switch (rand() % 3) {
        case 0: throw 123;
```

```

        case 1: throw 'e';
        case 2: throw 3.14; // ошибка времени выполнения. нет
соответствующего case с типом double
    }
}
catch (int e) {cout << "Пойман int" << endl; }
catch (char e) {cout << "Пойман char" << endl; }

```

Преобразования типов во время создания исключения не происходит. Поэтому если кинуть тип `int`, то обработчик исключения `catch (double e)` не будет выполнен, даже если для `int` не найдётся подходящего обработчика.

Если оператор `throw` был вызван вне защищенного блока (что чаще всего случается, когда исключение возбуждается в вызванной функции), или если не был найден ни один подходящий обработчик этого исключения, то вызывается стандартная функция `terminate()`. Она, в свою очередь, вызывает функцию `abort()` для завершения работы с приложением.

Однако есть возможность поймать исключение произвольного типа

```

catch (...){
// какой-то другой тип был пойман
}

```

Такой обработчик всегда должен быть последним. Он должен применяться когда используется оператор `throw` без параметров.

блок `catch (void *)` – после всех блоков с указательными типами.

вложенные блоки try-catch

Блок `try-catch` может содержать вложенные блоки `try-catch` и если не будет определено соответствующего оператора `catch` на текущем уровне вложения, исключение будет поймано на более высоком уровне. Пойманное исключение далее не передаётся.

```

void example444(){
    if (rand() % 2) throw 123;
    else throw "Catch me!";
}
void example44(){
    try{example444();}
    catch (char const *e){
        cout << "поймано в 44" << endl;
        cout << e << endl;}
}
void example4(){
    try{example44();}
    catch (int e){
        cout << "поймано в 4" << endl;
        cout << e << endl;}
}

```

При этом подходящий под тип брошенного значения обработчик будет выбран из всего стека вызовов.

Задание ограничений на исключения

```
возвращаемый_тип имя_функции (список аргументов) throw (список типов)
{
//...
}
```

Если будет сгенерировано исключение какого-либо другого типа, то произойдет аварийное завершение программы. Если необходимо, чтобы функция не могла сгенерировать никакого исключения, то следует оставить список пустым.

если в функции содержится свой блок try, то она может генерировать любые исключения до тех пор, пока эти исключения перехватываются инструкциями catch внутри самой функции.

```
void Xhandler(int test) throw (int, char, double)
{
if(test==0) throw test; // генерация int
if(test==1) throw 'a'; // генерация char
if(test==2) throw 123.23; // генерация double
}
```

Повторная генерация исключений

Исключения можно создавать и внутри блока catch() {}

В результате текущее исключение будет передано во внешнюю последовательность try/catch обработки исключений. Причиной для этого может послужить желание обрабатывать исключения несколькими обработчиками. Например, один обработчик может заниматься одним аспектом исключения, а второй обработчик — другим. Исключение может быть снова сгенерировано или изнутри блока catch, или из функции, вызванной в этом блоке. Когда повторно генерируется исключение, оно не будет перехвачено той же самой инструкцией catch. Оно будет распространяться до следующей внешней инструкции catch.

```
void Xhandler()
{
try {
throw "hello"; // генерация char *
}
catch (char *) { // перехват char *
cout << "Caught char * inside Xhandler\n";
throw; // повторная генерация char * извне функции
}
}
int main()
{
cout << "Start\n";
try{ Xhandler(); }
catch(char *) {
cout << "Caught char * inside main\n"; }
cout << "End";
return 0;
}
```

Достоинства и недостатки

Достоинства

Достоинства использования исключений особенно заметно проявляются при разработке библиотек процедур и программных [компонентов](#), ориентированных на массовое использование. В таких случаях разработчик часто не знает, как именно должна обрабатываться исключительная ситуация (при написании универсальной процедуры чтения из файла невозможно заранее предусмотреть реакцию на ошибку, так как эта реакция зависит от использующей процедуру программы), но ему это и не нужно— достаточно сгенерировать исключение, обработчик которого предоставляется реализовать пользователю компонента или процедуры.

- пользователь компонента сам определяет реакцию на исключение.

Недостатки

- реализация механизма обработки исключений существенно зависит от языка, и даже компиляторы одного и того же языка на одной и той же платформе могут иметь значительные различия. Это не позволяет прозрачно передавать исключения между частями программы, написанными на разных языках;

- обработка исключений — медленная. В местах программы, критичных по скорости, не рекомендуют возбуждать и обрабатывать исключения.

Продолжение — Использование стандартных исключений.

ООП

Декомпозиция

Программы — сложные. Декомпозиция — путь к упрощению. Разбиение программы на независимые части. В структурном программировании — это алгоритмическая декомпозиция.

Алгоритмическая декомпозиция — разбиение функционала от общего к частному. Алгоритмическая декомпозиция часто представляет собой некоторую иерархию. Где в основной алгоритм (задачу) включается несколько подзадач, а в них в свою очередь, включаются более элементарные задачи и так далее.

Другой путь уменьшения сложности - разделить некую систему по признаку принадлежности ее элементов различным абстракциям данной проблемной области.

Объектно-ориентированная декомпозиция - это разбиение системы на сущности, являющиеся какими-либо объектами действующими в той ситуации, которую как раз и моделирует система.

Готовясь к экзамену не всегда проще изучать материал последовательно, как она записан в конспекте. Особенно, если ежу есть представление о материале. Иногда удобнее представить материал в виде некоторой иерархии, дерева: Разделы → подразделы → подподразделы и т.д. Но понятия, принципы, модели и т.п. вещи не всегда можно представить иерархически. Чаще всего они связаны между собой непоследовательно и их можно связать в сеть. Похожий подход исповедует ОО декомпозиция.

Такая декомпозиция считается "более продвинутой" (при разработке сложных систем) в отличии от **Алгоритмической**. Мир представляет собой совокупность взаимодействующих объектов. Каждый объект в такой системе моделирует поведение объекта реального мира.

Что лучше: алгоритмическая или объектная декомпозиция? Ответ – обе хороши (или не достаточно хороши). Разделяя алгоритмы, мы концентрируем внимание на порядке действий, разделяя по объектам, мы концентрируемся на агентах, являющихся или объектами, или субъектами действий

Как выделять объекты?

Объект — это нечто, имеющее четко определенные границы; Объект должен характеризоваться состоянием, поведением и идентичностью.

Состояние объекта характеризуется перечнем всех (как правило, статических) свойств данного объекта и текущими (как правило, динамическими) значениями каждого из этих свойств.

Поведение — это действия и реакции объекта, выраженные через изменения состояния объекта и передачу сообщений.

Состояние объекта представляет собой суммарный результат его поведения.

см. шаблоны проектирования.

Операции

Операция — это услуга, которую класс оказывает своим клиентам.

На практике клиент обычно совершает над объектами операции пяти видов.

- **Модификатор:** операция, изменяющая состояние объекта.
- **Селектор:** операция, имеющая доступ к состоянию объекта, но не изменяющая его.
- **Итератор:** операция, обеспечивающая доступ ко всем частям объекта в строго определенном порядке.
- **Конструктор:** операция, создающая объект и/или инициализирующая его состояние.
- **Деструктор:** операция, стирающая состояние объекта и/или уничтожающая сам объект.

Начало ООП

Процедурное или модульное программирование и часто приводит к созданию так называемых монолитных приложений, все функции которых сконцентрированы в нескольких модулях кода (а то и вовсе в одном). В ООП обычно используется гораздо больше модулей, каждый из которых обеспечивает конкретные функции и может быть изолирован или даже полностью отделен от всех остальных.

ООА (OOA) - это методология, при которой требования к системе воспринимаются с точки зрения классов и объектов, выявленных в предметной области.

ООП (OOD) - Объектно-ориентированное проектирование - это методология проектирования, соединяющая в себе процесс объектной декомпозиции и приемы представления логической и физической, а также статической и динамической моделей проектируемой системы.

ООА → OOD → OOP

На результатах ООА формируются модели, на которых основывается OOD; OOD в свою очередь создает фундамент для окончательной реализации системы с использованием методологии OOP

Пример 1: Преподаватель читает лекцию, делает рисунки на доске, студент конспектирует.

Пример 2: Любой элемент графического интерфейса — объект. Кнопка, меню, область ввода, вкладка и т.д.

Развитие парадигм программирования и языков программирования

Объекты, как формализованный концепт появились в программировании в 1960-х в [Simula](#) 67. В языке использовался автоматический сборщик мусора.

Идеи Simula оказали серьезное влияние на более поздние языки, такие как Smalltalk, варианты Lisp (CLOS), Object Pascal, и C++.

Язык [Smalltalk](#), который был изобретен в компании [Xerox PARC](#) Аланом Кэем ([Alan Kay](#)) и некоторыми другими учеными, фактически навязывал использование «объектов» и «сообщений» как базиса для вычислений.

Объектно-ориентированное программирование развилось в доминирующую методологию программирования в начале и середине 1990 годов, когда стали широко доступны поддерживающие ее языки программирования, такие как Visual FoxPro 3.0, [C++](#), и Delphi.

Доминирование этой системы поддерживалось ростом популярности графических интерфейсов пользователя, которые основывались на техниках ООП.

см. заметки.

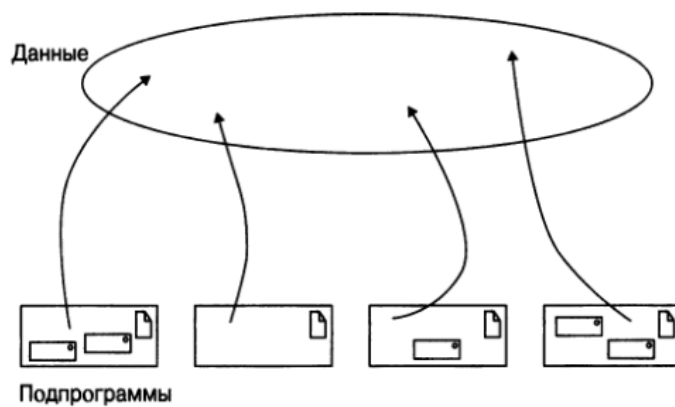


Рисунок 1. Программа на языке программирования 1-2 поколения.

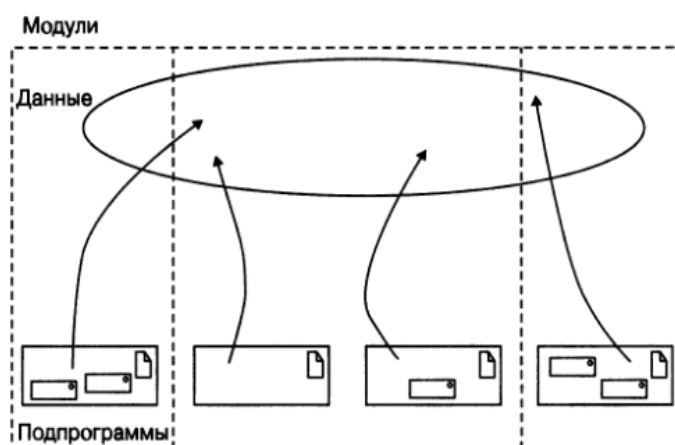


Рисунок 2. Программа на языке программирования третьего поколения

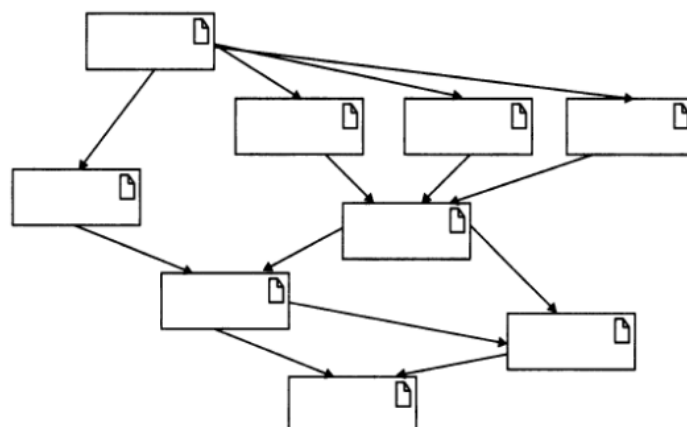


Рисунок 3. Программа на ОО языке программирования

Основные понятия ООП

Абстрактный тип данных (АТД) – это такой тип данных, который скрывает свою внутреннюю реализацию от клиентов. Это набор возможных значений и возможных операций. Различие между абстрактными типами данных и структурами данных, которые реализуют абстрактные типы, можно пояснить на следующем примере.

Абстрактный тип данных список может быть реализован при помощи массива или линейного списка, с использованием различных методов динамического выделения памяти. Однако каждая реализация определяет один и тот же набор функций, который должен работать одинаково (по результату, а не по скорости) для всех реализаций.

Преимущества АТД

- **Инкапсуляция деталей реализации.**

Это означает, что единожды инкапсулировав детали реализации работы АТД мы предоставляем клиенту интерфейс, при помощи которого он может взаимодействовать с АТД. Изменив детали реализации, представление клиентов о работе АТД не изменится.

- **Снижение сложности.**

Путем абстрагирования от деталей реализации, мы сосредотачиваемся на интерфейсе, т.е. на том, что может делать АТД, а не на том как это делается. Более того, АТД позволяет нам работать с сущностью реального мира.

- **Ограничение области использования данных.**

Используя АТД мы можем быть уверены, что данные, представляющие внутреннюю структуру АТД не будут зависеть от других участков кода. При этом реализуется “независимость” АТД.

- **Высокая информативность интерфейса.**

АТД позволяет представить весь интерфейс в терминах сущностей предметной области, что, согласитесь, повышает удобочитаемость и информативность интерфейса.

Класс - это элемент ПО, описывающий АТД и его частичную или полную реализацию.

Класс - универсальный, комплексный тип данных, состоящий из тематически единого набора «полей» (переменных более элементарных типов) и «методов» (функций для работы с этими полями)

Методы класса — это его функции.

Свойства\атрибуты\поля\информационные члены класса — его переменные.

Члены класса — методы и поля класса.

Объекты являются экземплярами некоторого заранее описанного класса.

Интерфейс совокупность средств, методов и правил взаимодействия (управления, контроля и т.д.) между элементами системы.

Интерфейс (ООП) - то, что доступно при использовании класса извне. Как правило это набор методов.

Главное отличие класса от интерфейса — в том, что **класс состоит из интерфейса и реализации**.

Каждый **объект** характеризуется:

- **Состояние** - набор атрибутов, определяющих поведение объекта.
- **Поведение** - это то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений.
- **Идентичность (уникальность)** - это такое свойство объекта, которое отличает его от всех других объектов.

В большинстве языков программирования для различения объектов используют имя, тем самым путая адресуемость и идентичность. В базах данных различают объекты по набору ключевых полей, тем самым смешивая идентичность и значение данных.

Структура же и поведение схожих объектов определяется в общем для них классе.

Основные принципы ООП

Абстрагирование (Abstraction) означает выделение значимой информации и исключение из рассмотрения незначимой. В ООП рассматривают лишь абстракцию данных.

Наследование (Inheritance) касается способности языка позволять строить новые определения классов на основе определений существующих классов.

Инкапсуляция (Encapsulation) - это механизм программирования, объединяющий вместе код и данные, которыми он манипулирует, исключая как вмешательство извне, так и неправильное использование данных. Доступ к коду и данным жестко контролируется интерфейсом.

Полиморфизм (Polymorphism) - свойство системы, позволяющее использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Например для сортировки целочисленного массива и сортировки массива строк можно использовать функцию(метод) с одним и тем же именем и набором параметров. Однако это фактически будут две функции, внутренний механизм работы которых скрыт от программиста. Это упрощает работу.

Иерархия - это ранжирование или упорядочение абстракций.

Типизация - это система правил, предотвращающих или сильно ограничивающих взаимную замену объектов разных классов.

Параллелизм - это свойство, отличающее активные объекты от пассивных.

Сохраняемость (персистентность) - это способность объекта преодолевать временные или пространственные пределы.

Отношения между объектами

- **Связь (association).** Объекты передают друг другу сообщения. (Писатель передаёт сообщение ручке, ручка — листу бумаги.). Отношения вида клиент — сервер.
- **Агрегация. (aggregation)** Один объект может быть частью другого. Иерархия часть — целое.

Связи (link)

Участвуя в связи, объект может выполнять одну из следующих трех ролей:

- **Контроллер (controller).** Этот объект может выполнять операции с другими объектами, но сам никогда не подвергается воздействию других объектов. В некоторых предметных областях термины активный объект (active object) и контроллер являются синонимами.

- **Сервер (server).** Этот объект не выполняет операций с другими объектами, а лишь подвергается воздействию со стороны других объектов.

- **Агент (проху).** Такой объект может выполнять операции над другими объектами и подвергаться воздействию с их стороны. Как правило, агенты создаются для представления реальных объектов, существующих в предметной области конкретного приложения.

Отношения между классами

- **Связь(ассоциация, association).** Семантическая зависимость между классами. (Например писатель — ручка)
- **Агрегация.** Аналогичны отношениям классов.
- **Наследование.** обобщение/специализация

Вышеприведённые отношения можно ещё разбить на подтипы.

Преимущества и недостатки ООП

Достоинства ООП:

- Основным достоинством объектно-ориентированного программирования по сравнению с модульным программированием является «более естественная» декомпозиция программного обеспечения, которая существенно облегчает его разработку.

- Кроме этого, объектный подход предлагает новые способы организации программ, основанные на механизмах наследования, полиморфизма, композиции, наполнения.

- Эти механизмы позволяют конструировать сложные объекты из сравнительно простых. В результате существенно увеличивается показатель повторного использования кода и появляется возможность создания библиотек классов для различных применений. Чтобы поменять бизнес-логику нужно заменить одни классы другим или добавить новые. В случае алгоритмической декомпозиции часто приходится менять алгоритм и данные с которыми он работает.

Недостатки ООП обуславливаются следующим:

- Освоение базовых концепций ООП не требует значительных усилий. Однако разработка библиотек классов и их использование требуют существенных трудозатрат.

- Документирование классов – задача более трудная, чем это было в случае процедур и модулей.

- В сложных иерархиях классов поля и методы обычно наследуются с разных уровней. И не всегда легко определить, какие поля и методы фактически относятся к данному классу.

- Код для обработки сообщения иногда «размазан» по многим методам (иначе говоря, обработка сообщения требует не одного, а многих методов, которые могут быть описаны в разных классах).

Основной недостаток ООП - некоторое снижение быстродействия за счет более сложной организации программной системы.

Классы в C++

```
class Имя_класса {
private:
    определение_закрытых_членов_класса
    доступный только собственным методам
public:
    определение_открытых_членов_класса
    доступны для использования со стороны
внешнего кода
    здесь описываются методы класса, реализующие
интерфейс объектов
protected:
    определение_защищенных_членов_класса
    недоступны для использования со стороны внешнего кода
    доступны наследникам
...
}
```

Private, protected, public —
модификаторы доступа.
Могут быть приведены в любом
порядке. Могут повторяться.

Обеспечивают инкапсуляцию.

```
};
```

Порядок следования областей доступа и их количество в классе – произвольны.

Служебное слово, определяющее первую область доступа, может отсутствовать. По умолчанию эта область считается `private`.

В закрытую (`private`) область обычно помещаются информационные члены, а в открытую (`public`) область – методы класса, реализующие интерфейс объектов класса с внешней средой.

Доступ к информационным членам и методам объекта:

- операции выбора члена класса «.». (доступ через объект или ссылку на объект)
- указателя на член класса «→» (доступ с помощью указателя на объект)

Пример

```
class X {
public:
    char c;
    int f(){...}
};

int main () {
    X x1;
    X &x2 = x1;
    X *p = &x1;
    int i, j, k;
    x1.c = '*';
    i = x1.f();
    x1.c = '+';
    j = x2.f();
    x1.c = '#';
    k = p -> f();
    ...
}
```

Объекты класса можно определять совместно с описанием класса:

```
class Y {...} y1, y2;
```

Класс C++ отличается от структуры C++ только определением по умолчанию первой области доступа в их описании (а также определением по умолчанию способа наследования.

- для структур умолчанием является открытый доступ (`public`)
- для классов умолчанием является закрытый доступ (`private`).

Структуры создавались для объединения и совместного использования разнородных типов данных, например записей файлов. Класс предназначен для определения полноценного типа данных.

Наличие информационных членов в открытой секции нарушает один из основных принципов ООП – принцип инкапсуляции.

Указатель this

- Явно получить доступ к самому объекту или к его членам.

В классах C++ неявно введен специальный указатель `this` – указатель на текущий объект. Через него методы класса могут получить доступ к другим членам класса. Каждый метод класса при обращении к нему получает данный указатель в качестве неявного параметра. При вызове член-функции ей передается неявный аргумент, содержащий адрес объекта, для которого эта функция вызывается.

Как это выглядит	Как это можно представить
<pre>class X{ int data; public: void foo() {this-> data = 42;} ... } x; c.foo();</pre>	<pre>class X { int data; ... } x; void foo(X *this){ this->data = 42;} foo(&x);</pre>

Во втором случае только нужно дополнительно организовать доступ ко всем членам класса `X` изнутри функции.

Указатель `this` можно рассматривать как локальную константу, имеющую тип `X*`, если `X` – имя описываемого класса. Нет необходимости использовать его явно. Он используется явно, например, в том случае, когда выходным значением для метода является текущий объект.

Данный указатель, как и другие указатели, может быть разыменован. При передаче возвращаемого значения метода класса в виде ссылки на текущий объект используется разыменованный указатель `this`, так как ссылка, какуже было указано, инициализируется непосредственным значением.

Пример:

```
class x {
  ...
public:
  x& f( . . . ){
    ...
    return *this;
  }
};
```

Пример.

```
class Human{
private:
  std::string name;
  long age;
public:
  std::string get_name() const {return name;}
```

```

    long get_age() const {return age;}
    void set_name(std::string name){
        this->name = name;
    }
    void set_age(long age){
        if (age > 0) this->age = age;
    }
};
class Student: public Human{
private:
    char year; // номер курса
public:
    void test(){
        // age=33; // Ошибка компиляции! поле age недоступно из этого класса.
    }
    char get_year() const {return year;}
    void set_year(char y) {if (y>0 and y<6) this->year = y;}
};
int main(){
    Human h;
    h.set_age(22);
    h.set_name("Вася");
    Student s;
    s.set_age(21); // не смотря на то, что age не доступен, он содержится в
классе
    cout << s.get_age() << endl;
    return 0;
}

```

Инкапсуляция

Манипуляции с состоянием объекта

Для доступа к внутренним информационным членам объекта, созданного на основе класса (чтение/запись), необходимо использовать специальные методы класса, называемые модификаторами (setters) и селекторами (getters). Они осуществляют подконтрольное считывание и изменение внутренних информационных членов.

Таким образом, изменение информационных полей объекта должно осуществляться специальными методами, производящими изменение требуемого информационного поля согласованно с одновременным изменением других информационных полей. Такие методы обеспечивают согласованность внутренних данных объекта.

Конструктор и деструктор

Конструкторы и деструкторы являются специальными методами класса.

Конструкторы вызываются при создании объектов класса и отведении памяти под них.

Деструкторы вызываются при уничтожении объектов и освобождении от веденной для них памяти.

В большинстве случаев конструкторы и деструкторы вызываются автоматически (неявно) соответственно при описании объекта (в момент отведения памяти под него) и при уничтожении объекта. Конструктор (как и деструктор) может вызываться и явно, например, при создании объекта в динамической области памяти с помощью операции `new`.

Так как конструкторы и деструкторы неявно входят в интерфейс объекта, их следует располагать в открытой области класса.

Примечание. Конструкторы и деструкторы могут располагаться и в закрытой области для блокирования возможности неявного создания объекта. Но в этом случае явное создание объекта возможно только при использовании статических методов, являющихся частью класса, а не конкретного объекта.

Отличия и особенности описания конструктора от обычной функции:

- 1) Имя конструктора совпадает с именем класса
- 2) При описании конструктора не указывается тип возвращаемого значения

Следует отметить, что и обычная процедура может не возвращать значения, а только перерабатывать имеющиеся данные. В этом случае при описании соответствующей функции указывается специальный тип возвращаемого значения `void`.

В описании конструктора тип возвращаемого значения не указывается не потому, что возвращаемого значения нет. Оно как раз есть. Ведь результатом работы конструктора в соответствии с его названием является созданный объект того типа, который описывается данным классом. Страуструп отмечал, что кон- структор – это то, что область памяти превращает в объект.

Конструкторы можно классифицировать разными способами:

- 1) по наличию параметров:
 - без параметров,
 - с параметрами;
- 2) по количеству и типу параметров:
 - конструктор умолчания,
 - конструктор преобразования,
 - конструктор копирования,
 - конструктор с двумя и более параметрами.

Набор и типы параметров зависят от того, на основе каких данных создается объект. В классе может быть несколько конструкторов. В соответствии с правилами языка C++ все они имеют одно имя, совпадающее с именем класса, что является одним из проявлений статического полиморфизма. Компилятор выбирает тот конструктор, который в зависимости

от ситуации, в которой происходит создание объекта, удовлетворяет ей по количеству и типам параметров. Естественным ограничением является то, что в классе не может быть двух конструкторов с одинаковым набором параметров.

Деструкторы применяются для корректного уничтожения объектов. Часто процесс уничтожения объектов включает в себя действия по освобождению выделенной для них по операциям new памяти.

Имя деструктора: ~имя_класса

У деструкторов нет параметров и возвращаемого значения.

В отличие от конструкторов деструктор в классе может быть только один.

Пример: Описание класса.

```
class box{
    int len, wid, hei;
public:
    box(int l, int w, int h){
        len = l; wid = w; hei = h;
    }
    24}
};
box(int s){
    len = wid = hei = s;
}
box(){
    len = 2; wid = hei = 1;
    int volume(){
        return len * wid * hei;
    }
}
```

Конструктор предназначен для инициализации объекта и вызывается автоматически при его создании. Ниже перечислены основные свойства конструкторов.

1. Конструктор не возвращает значения, даже типа void . Нельзя получить указатель на конструктор.
2. Класс может иметь несколько конструкторов с разными параметрами для разных видов инициализации (при этом используется механизм перегрузки).
3. Конструктор, который можно вызвать без параметров, называется конструктором по умолчанию.
4. Параметры конструктора могут иметь любой тип, кроме этого же класса. Можно задавать значения параметров по умолчанию. Их может содержать только один из конструкторов.
5. Если программист не указал ни одного конструктора, компилятор создает его автоматически (кроме случая, когда класс содержит константы и ссылки, поскольку их необходимо инициализировать). Такой конструктор вызывает конструкторы по умолчанию для полей класса и конструкторы базовых классов.

6. Конструкторы не наследуются.

7. Конструктор не может быть константным, статическим и виртуальным (нельзя использовать модификаторы `const` , `virtual` и `static`).

8. Конструкторы глобальных объектов вызываются до вызова функции `main` . Локальные объекты создаются, как только становится активной область их действия. Конструктор запускается и при создании временного объекта (например, при передаче объекта из функции).

При объявлении объектов вызывается один из конструкторов. При отсутствии инициализирующего выражения в объявлении объекта вызывается конструктор по умолчанию, при инициализации другим объектом того же типа – конструктор копирования (см. далее), при инициализации полей один из явно определенных конструкторов инициализации (т.е. конструкторов, которым передаются параметры для инициализации полей объекта).

Класс как область видимости

Класс является областью видимости описанных в нем членов класса. Идентификатор члена класса локален по отношению к данному классу. Классы могут быть вложенными. Одноименные идентификаторы членов класса закрывают видимость соответствующих внешних идентификаторов.

Операция `::` позволяет получить доступ к одноименным объектам, внешним по отношению к текущей области видимости, в частности, к глобальным функциям и переменным, следующим образом:

```
имя_класса :: имя_члена_класса
:: имя - для имен глобальных функций и переменных.
```

Пример:

```
int ia1;
void f1(int b1) {
    ia1 = ia1 + b1;
}
class x {
    int ia1;
public:
    x(){ia1 = 0;}
    void f1(int b1){
        ::f1(b1); // вызов глобальной функции
    }
};
int main(){
    x a2;
    a2.f1(2);
    return 0;
}
```

Объявление и определение методов класса. Спецификатор inline

Каждый метод класса, должен быть определен в программе. Определить метод класса можно либо непосредственно в классе (если тело метода не слишком сложно и громоздко), либо вынести определение вне класса, а в классе только объявить соответствующий метод, указав его прототип.

При определении метода класса вне класса для указания области видимости соответствующего имени метода используется операции '::':

Пример:

```
class x {
    int ia1;
public:
    x(){ia1 = 0;}
    int func1();
};
int x::func1(){ ... return ia1; }
```

Это позволяет повысить наглядность текста, особенно, в случае значительного объема кода в методах. При определении метода вне класса с использованием операции '::' прототипы объявления и определения функции должны совпадать.

Метод класса и любую функцию, не связанную ни с каким классом, можно определить со спецификатором inline:

```
inline int func1();
```

Такие функции называются встроенными.

Спецификатор inline указывает компилятору, что необходимо по возможности генерировать в точке вызова код функции, а не команды вызова функции, находящейся в отдельном месте кода модуля. Это позволяет уменьшить время выполнения программы за счет отсутствия команд вызова функции и возврата из функции, которые кроме передачи управления выполняют действия соответственно по сохранению и восстановлению контекста (содержимого основных регистров процессора). При этом размер модуля оказывается увеличенным по сравнению с программой без спецификаторов inline. Следует отметить, что спецификатор inline является рекомендацией компилятору. Данный спецификатор неприменим для функций со сложной логикой. В случае невозможности использования спецификатора для конкретной функции компилятор выдает предупреждающее сообщение и обрабатывает функции стандартным способом.

По определению методы класса, определенные непосредственно в классе, являются inline-функциям

Инициализация, конструкторы, оператор присваивания

<https://msdn.microsoft.com/ru-ru/library/s16xw1a8.aspx>

Конструктор — это особая функция-член, инициализирующая экземпляр своего класса. Конструкторы имеют имена, совпадающие с именами классов, и не имеют возвращаемых значений.

- У конструктора может быть любое число параметров,
- а у класса — любое число перегруженных конструкторов.
- Конструкторы могут иметь любой уровень доступа — открытый, защищенный или закрытый.
- Если вы не определили ни одного конструктора, компилятор создаст конструктор по умолчанию, не имеющий параметров.

Это поведение можно переопределить, объявив конструктор по умолчанию как удаленный.

Конструктор выполняет свою работу в следующем порядке.

1. Вызывает конструкторы базовых классов и членов в порядке объявления.
2. Если класс является производным от виртуальных базовых классов, конструктор инициализирует указатели виртуальных базовых классов объекта.
3. Если класс имеет или наследует виртуальные функции, конструктор инициализирует указатели виртуальных функций объекта. Указатели виртуальных функций указывают на таблицу виртуальных функций класса, чтобы обеспечить правильную привязку вызовов виртуальных функций к коду.
4. Выполняет весь код в теле функции.

В следующем примере показан порядок, в котором конструкторы базовых классов и конструкторы-члены вызываются в конструкторе производного класса. Сначала вызывается конструктор базового класса, затем инициализируются члены базового класса в порядке их появления в объявлении класса. После этого вызывается конструктор производного класса.

```
#include <iostream>
using namespace std;

class C1 {
public: C1() {cout << " C1 constructor." << endl; }
};

class C2 {
public: C2() { cout << " C2 constructor." << endl; }
};

class C3 {
```

```

public: C3() { cout << " C3 constructor." << endl; }
};

class BaseC {
public:
    BaseC() { cout << " BaseC constructor." << endl; }
private:
    C1 c1;
    C2 c2;
};

class DerivedC : public BaseC {
public:
    DerivedC() : BaseC() {cout << "DerivedContainer constructor." << endl;}
private: Contained3 c3;
};

int main() {
    DerivedContainer dc;
}

```

Выходные данные этого кода:

```

Contained1 constructor.
Contained2 constructor.
BaseContainer constructor.
Contained3 constructor.
DerivedContainer constructor.

```

Прямая инициализация (direct initialization)

<http://www.geeksforgeeks.org/when-do-we-use-initializer-list-in-c/>

Инициализируйте члены класса из аргументов конструктора, используя список инициализации членов. В этом методе применяется *прямая инициализация*, что более эффективно, чем использование операторов присваивания в теле конструктора.

```

class Box {
public:
    Box(int width, int length, int height)
        : m_width(width), m_length(length), m_height(height) // member init
list
    {}
    int Volume() {return m_width * m_length * m_height; }
private:
    int m_width;
    int m_length;
    int m_height;
};

```

```

Box b(42, 21, 12);
cout << "The volume is " << b.Volume();

```

Списки инициализации (braced initializer lists)

[Wiki](#)

<https://habrahabr.ru/post/330402/>

Концепция списков инициализации пришла в C++ из C. Идея состоит в том, что структура или массив могут быть созданы передачей списка аргументов в порядке, соответствующем порядку определения членов структуры. Списки инициализации рекурсивны, что позволяет их использовать для массивов структур и структур, содержащих вложенные структуры.

```
struct Object{
    float first;
    int second;
};

Object scalar = {0.43f, 10}; // один объект, с first=0.43f и second=10
Object anArray[] = {{13.4f, 3}, {43.28f, 29}, {5.934f, 17}}; // массив из
трёх объектов
struct S1{
    float x, y, z;
};

struct S2{
    float x;
    S1 s;};

S1 s1 = {1, 2, 3};
S2 s2 = {10, {1, 2, 3}};
S2 s22 = {42, s1};
```

Списки инициализации vs явные конструкторы.

С подобной разницей можно встретиться в случае `std::vector`. конструкция `std::vector<int> x(5);`

создаст вектор на 5 элементов с нулями (дефолтное значение для `int`), а `std::vector<int> x{5};` создаст вектор на один элемент со значением 5.

Конструкторы по умолчанию (Default constructor)

`MyClass()`

- являются *специальными функциями-членами*.
- не имеют параметров
- создаётся компилятором автоматически, если в классе не объявлено ни одного конструктора
- в случае явного объявления конструкторов компилятор не предоставляет конструктор по умолчанию.

- Когда вызываются:

```
MyClass c;
MyClass ccc[10];
```

Если при вызове конструктора по умолчанию вы пытаетесь использовать скобки, выводится предупреждение:

```
class myclass{};
int main(){
    myclass mc(); // warning C4930: prototyped function not called (was a
variable definition intended?)
}
```

Это пример проблемы Most Vexing Parse (наиболее неоднозначного анализа). Поскольку выражение примера можно интерпретировать как объявление функции или как вызов конструктора по умолчанию и в связи с тем, что средства синтаксического анализа C++ отдают предпочтение объявлениям перед другими действиями, данное выражение обрабатывается как объявление функции. Дополнительные сведения см. в статье Википедии [Most Vexing Parse \(Наиболее неоднозначный анализ\)](#).

Пример

```
// Класс без конструктора
class C_nc{
private:
    float x,y;
    float abs() {return 10;}
};

// Класс без конструктора по-умолчанию
class C{
public: C(float x, float y) {}
private:
    float x,y;
    float abs() {return 10;}
};

using namespace std;

int main(){
    C_nc c1;          // Конструктора нет, но компилятор об этом позаботится
    // C c2;          // Если есть любой конструктор, кроме к.по-умолчанию -
компилятор не будет создавать к. по-умолчанию! Ошибка!
    C ccc[3];         // Так тоже нельзя! Здесь нужен конструктор по умолчанию.
    vector<C> vc;      // А так можно! WTF!?!

    // но если использовать списки инициализации
    C cccc[3]{ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } }; // так снова можно
```

Если у параметра конструктора преобразования имеется априорное значение, и при описании объекта явно не задается фактический параметр, этот конструктор играет роль конструктора умолчания.

Пример:

```
class X {
```

```
int x1;
public:
X(int px1 = 0)
};
```

Для такого класса будут верны следующие объявления объектов:

```
int main(){
... X x1, x2(1); ...
}
```

Явные конструкторы (Explicit Constructor)

- Когда вызываются

```
MyClass c(1);
```

Ключевое слово `explicit` используется для создания явных конструкторов. Другое название - «неконвертирующиеся конструкторы» (`nonconverting constructors`).

Рассмотрим следующий класс:

```
class MyClass {
int i;
public:
MyClass(int j)
{i = j;}
// ...
};
```

Объекты этого класса могут быть объявлены двумя способами:

```
MyClass ob1(1);
MyClass ob2 = 10;
```

В данном случае инструкция:

```
MyClass ob2 = 10;
```

автоматически конвертируется в следующую форму:

```
MyClass ob2(10);
```

Однако, если объявить конструктор `MyClass` с ключевым словом `explicit`, то это автоматическое конвертирование не будет выполняться. Ниже объявление класса `MyClass` показано с использованием ключевого слова `explicit` при объявлении конструктора:

```
class MyClass {
int i;
public:
explicit MyClass(int j)
{i = j;}
// ...
};
```

Теперь допустимыми являются только конструкции следующего вида:

```
MyClass ob (110);
```

Если у класса имеется конструктор с одним параметром, или у всех параметров, кроме одного, имеются значения по умолчанию, тип параметра можно неявно преобразовать в тип класса. Например, если у класса `Box` имеется конструктор, подобный следующему:

```
Box(int size): m_width(size), m_length(size), m_height(size){}
```

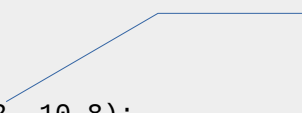
то возможно инициализировать объект `Box` следующим образом:

```
Box b = 42;
```

Или передать целое значение функции, принимающей объект `Box`:

```
class ShippingOrder {
public:
    ShippingOrder(Box b, double postage) : m_box(b), m_postage(postage){}

private:
    Box m_box;
    double m_postage;
}
//elsewhere...
ShippingOrder so(42, 10.8);
```



Подразумевается, что вместо 42 создается объект Box

В некоторых случаях подобные преобразования могут быть полезны, однако чаще всего они могут привести к незаметным, но серьезным ошибкам в вашем коде. Как правило, для конструктора (и пользовательских операторов) следует использовать ключевое слово `explicit`, чтобы избежать подобных неявных преобразований типа:

```
explicit Box(int size): m_width(size), m_length(size), m_height(size){}
```

Когда конструктор является явным, эта строка вызывает ошибку компилятора:

```
ShippingOrder so(42, 10.8);
```

Вот и все объяснение. Но есть и нюансы (подводные камни) о которых необходимо помнить. Вот выдержка из Википедии:

Операторы явного преобразования

Стандарт C++ предлагает ключевое слово `explicit` как модификатор конструкторов с одним параметром, чтобы такие конструкторы не функционировали как конструкторы неявного преобразования. Однако это никак не влияет на действительные операторы преобразования. Например, класс умного указателя может содержать оператор `bool()` для имитации обычного указателя. Такой оператор можно вызвать, например, так: `if(smart_ptr_variable)` (ветка выполняется, если указатель ненулевой). Проблемой является то, что такой оператор не защищает от других непредвиденных преобразований. Поскольку в C++ тип `bool` объявлен как арифметический, возможно неявное преобразование в любой целочисленный тип или даже в тип числа с плавающей точкой, что в свою очередь может привести к непредвиденным математическим операциям.

В C++11 ключевое слово `explicit` применимо и к операторам преобразования. По аналогии с конструкторами, оно защищает от непредвиденных неявных преобразований. Однако ситуации, когда язык по контексту ожидает булевый тип (например, в условных

выражениях, циклах и операндах логических операторов), считаются явными преобразованиями и оператор явного преобразования в `bool` вызывается непосредственно.

Конструктор преобразования (Conversion constructor) и конструкторы с двумя и более параметрами

```
X(T); // X и T – разные классы
X(T&);
X(const T&);
```

Если для создания объекта необходимы параметры, то они указываются в круглых скобках после идентификатора переменной:

- когда вызывается

```
box b2(1, 2, 3);
box b3(5);
```

Указываемые параметры являются параметрами конструктора класса. Если у конструктора имеется ровно один входной параметр, который не представляет собой ссылку на свой собственный класс, то соответствующий конструктор называется **конструктором преобразования**. Этот конструктор называется так в связи с тем, что в результате его работы на основе объекта одного типа создается объект другого типа (типа описываемого класса).

Если уже описан класс `T` и описывается новый класс `X`, то его конструкторы преобразования могут иметь любой из следующих прототипов:

```
X(T);
X(T&);
X(const T&);
```

Последний прототип служит для защиты от изменения передаваемого фактического параметра в теле конструктора, так как при получении ссылки на фактический параметр используется собственно передаваемый объект, а не его локальная копия.

Примечание. Выделение в отдельную группу **конструкторов с двумя и более параметрами**, независимо от их типа, является в некотором смысле, условным. Так, например, если есть два класса: `Vector` и `Matrix`, то для создания соответствующих объектов:

```
Vector v1(10);
```

```
Matrix m1(10,15);
```

используется в первом случае один параметр, а во втором случае – два параметра. Таким образом, в первом случае объект создается с помощью конструктора преобразования, а во втором случае, с формальной точки зрения, с помощью конструктора с двумя параметрами, хотя в обоих случаях фактически выполняется одна и та же процедура: создание объекта на основе заданных числовых параметров.

Конструктор копирования (copy constructor)

```
CopyMe(const CopyMe& c)
```

Конструктор копирования — это специальная функция-член, принимающая в качестве входных данных ссылку на объект того же типа и создающая его копию.

```
Point a, b;  
...  
a = b;
```

- Если конструктор копии не объявлен, компилятор создает конструктор копии для каждого члена.
- Если оператор присваивания копированием не объявлен, компилятор создает оператор присваивания копированием для каждого члена.
- Объявление конструктора копии не подавляет созданный компилятором оператор присваивания копий, и наоборот.
- Когда вызывается

```
box b = a;
```

```
// Конструктор копирования  
class CopyMe{  
public:  
    vector<int> v;  
    CopyMe(){}  
    CopyMe(const CopyMe& c){  
        v = c.v;  
        cout << "Copy constructor" << endl; }  
};  
  
CopyMe cm1;  
cm1.v = vector<int>(10,0);  
CopyMe cm2 = cm1; // Будет напечатано "Copy constructor"
```

Если тип аргумента для конструктора копии — не **const**, инициализация путем копирования объекта **const** возвращает ошибку.

Конструктору передаётся именно ссылка на объект, а не сам объект. Иначе пришлось бы при передаче объекта по значению в конструктор копирования вызывать конструктор копирования.

Важно. При копировании класса, если в нём содержатся поля ссылки или указатели следует копировать не сами указатели и области памяти, на которые они указывают.

Если класс не предусматривает создания внутренних динамических структур, например, массивов, создаваемых с использованием операции `new`, то в конструкторе копирования достаточно предусмотреть **поверхностное копирование**, то есть почленное копирование информационных членов класса.

Если же в классе предусмотрено создание внутренних динамических структур, использование только поверхностного копирования будет ошибочным, так как информационные члены-указатели, находящиеся в разных объектах, будут иметь одинаковые значения и указывать на одну и ту же размещенную в динамической памяти структуру. Автоматически сгенерированный конструктор копирования в данном классе не позволит корректно создавать объекты такого типа на основе других объектов этого же типа.

В подобных случаях необходимо **глубокое копирование**, осуществляющее не только копирование информационных членов объекта, но и самих динамических структур. При этом, естественно, информационные члены-указатели в создаваемом объекте должны не механически копироваться из объекта-инициализатора, а указывать на вновь созданные динамические структуры нового объекта.

Поверхностное копирование (shallow copy) - копирует только объект, а состояние является разделяемым. Автоматически сгенерированный конструктор копирования способен выполнить только поверхностное копирование.

Глубокое копирование (deep copy) - копирует объект и состояние, если нужно — рекурсивно.

Пример.

```
class X{
    int *p;}
```

Глубокое копирование заново выделит память для p и скопирует содержимое.

Поверхностное копирование скопирует только сам указатель. Т.о. будет создано два класса с указателями на общую область памяти.

Обращение к закрытым членам передаваемого в конструктор объекта

Входной параметр является внешним объектом по отношению к создаваемому объекту. Тем не менее, имеется возможность прямого обращения к закрытым членам этого внешнего объекта. Это возможно только потому, что входной параметр имеет тип, совпадающий с типом создаваемого в результате работы конструктора копирования объекта. Если бы на вход конструктора поступал бы объект другого типа (например, в конструкторе преобразования класса vector входным параметром был бы объект, созданный на основе класса matrix), то для доступа к закрытым членам объекта-параметра необходимо было бы применять специальные средства.

Это связано с тем, что **единицей защиты является не объект, а тип**, то есть методы объекта могут обращаться к закрытым членам не только данного объекта, но и к закрытым членам любого объекта данного типа.

Метод swap

Помимо к. копирования бывает полезно реализовать метод swap, который обменивает объекты местами. В реализации этого метода полезно использовать алгоритм их стандартной библиотеки std::swap. <http://ru.cppreference.com/w/cpp/algorithm/swap>

```
template< class T >  
void swap( T& a, T& b );
```

Конструктор перемещений (move constructor)

c(C&& other)

Конструктор перемещений (move constructor)— это тоже специальная функция-член, передающая право на владение текущим объектом новой переменной без копирования исходных данных.

Проблема

```
string func(){
string s;
//do something with s
return s;
}
string mystr=func()
```

В функции func будет создан временный объект s. Далее этот объект будет *копирован* в переменную mystr, затем уничтожен. Происходит лишнее копирование, вместо того, чтобы переместить содержимое.

- Конструктор перемещения повышает производительность

конструктор перемещения применяется в местах, где объявление совпадает с определением (инициализацией) *rvalue-ссылкой* на экземпляр этого же класса, либо посредством `direct initialization` в конструкторе класса/структуры (если же определение произойдет с помощью *lvalue-ссылки*, то вызовется конструктор копирования)

- принимают параметр T&&, который является rvalue.

Для создания конструкторов перемещения часто используется функция **std::move**

```
std::string str = "Hello";
std::vector<std::string> v;

// uses the push_back(const T&) overload, which means
// we'll incur the cost of copying str
v.push_back(str);
std::cout << "After copy, str is \"" << str << "\"\n";

// uses the rvalue reference push_back(T&&) overload,
// which means no strings will copied; instead, the contents
// of str will be moved into the vector. This is less
// expensive, but also means str might now be empty.
v.push_back(std::move(str));
std::cout << "After move, str is \"" << str << "\"\n";

std::cout << "The contents of the vector are \"" << v[0]
          << "\", \"" << v[1] << "\"\n";
```

Вывод:

```
After copy, str is "Hello"
After move, str is ""
The contents of the vector are "Hello", "Hello"
```

Конструктор копирования описывается следующим образом:

```
C::C(C&& other); //C++11 move constructor
```

- Когда вызывается

```
B b1;
B b2 = std::move(b1); // Явный вызов конструктора копирования
```

```
A foo(A a){return a;}
```

```
A a1 = f(A()); // неявный вызов конструктора копирования. Но сначала
вызовется конструктор по умолчанию.
```

Обмен значений переменных

Пусть T некоторый класс.

```
T tmp(a); // сейчас мы имеем две копии объекта a
a = b;    // теперь у нас есть две копии объекта b
b = tmp;   // а теперь у нас две копии объекта tmp (т.е. a)
```

Правильно

```
T tmp(std::move(a)); // В a теперь пусто
a = std::move(b);    // теперь a есть tmp
b = std::move(tmp);  // s
```

Поэтому для обмена значений двух объектов из STL следует использовать специальный метод swap;

```
vector<int> v1 = {1};
vector<int> v2 = {4,5,6};
v2.swap(v1); // меняет местами v2 и v
```

Конструктор копирования и операция присваивания

Если объект уже создан, то операция присваивания '=' осуществляет не инициализацию создаваемого объекта, а копирование данных, то есть передачу данных между существующими объектами.

Пример:

```
Box b3(4,1,1); // создание объекта b3
Box b2;
// создание объекта b2
b2 = b3;
// операция присваивания: копирование объекта
// b3 в существующий объект b2.
```

В операции присваивания, так же, как и в конструкторе копирования, по умолчанию осуществляется поверхностное копирование. Если требуется глубокое копирование, то необходимо перегрузить (описать нужный алгоритм) операцию присваивания. Перегрузка операций рассматривается в следующих разделах.

Операторы присваивания

Оператор присваивания перемещением (move assignment operator)

T& operator=(T&& data)

- генерируется автоматически компилятором в случае, если нет явного объявления программистом
- Код, сгенерированный компилятором, выполняет побитовое копирование.
- отличается от конструктора копирования тем, что должна очищать члены-данные цели присваивания (и правильно обрабатывать самоприсваивание), тогда как конструктор копирования присваивает значения неинициализированным членам-данным

move assignment operator = is used for transferring a temporary object to an existing object

```
class Resource {
public:
    Resource& operator=(Resource&& other) {
        if (this != &other) { // If the object isn't being called on itself
            delete this->data; // Delete the object's data
            this->data = other.data; // "Move" other's data into the
current object
            other.data = nullptr; // Mark the other object as "empty"
        }
        return *this; // return *this
    }
    void* data;
};
```

A move constructor is executed only when you construct an object. A move assignment operator is executed on a previously constructed object. It is exactly the same scenario as in the copy case.

```
Foo foo = std::move(bar); // construction, invokes move constructor
foo = std::move(other); // assignment, invokes move assignment operator
```

Оператор присваивания копированием (assignment operator)

MyClass & operator = (const MyClass & other)

```
class My_Array
{
    int * array;
    int count;

public:
    My_Array & operator = (const My_Array & other)
    {
        if (this != &other) // защита от неправильного самоприсваивания
        {
            // 1: выделяем "новую" память и копируем элементы
            int * new_array = new int[other.count];
            std::copy(other.array, other.array + other.count, new_array);

            // 2: освобождаем "старую" память
            delete [] array;

            // 3: присваиваем значения в "новой" памяти объекту
            array = new_array;
            count = other.count;
        }
        // по соглашению всегда возвращаем *this
        return *this;
    }

    ...
};
```

Причина, по которой операция = возвращает My_Array& вместо void, проста. Он разрешён для объединения присваиваний, как например:

```
array_1 = array_2 = array_3; // значение array_3 присваивается array_2
                             // затем значение array_2 присваивается array_1
```


Правило трёх (Закон Большой Тройки) → Правило пяти

С выходом [одиннадцатого стандарта](#) правило расширилось и теперь называется правило пяти.

Если класс или структура определяет один из следующих методов, то они должны явным образом определить все методы:

- Деструктор
- Конструктор копирования
- Оператор присваивания копированием
- Конструктор перемещения
- Оператор присваивания перемещением

```
class Rfive{
    char* cstring;

public:
    RFive(const char* arg) // Конструктор со списком инициализации и телом
    : cstring(new char[std::strlen(arg)+1]) {
        std::strcpy(cstring, arg);}

    ~RFive() // Деструктор
    {
        delete[] cstring;}

    RFive(const RFive& other) // Конструктор копирования
    {
        cstring = new char[std::strlen(other.cstring) + 1];
        std::strcpy(cstring, other.cstring);    }

    RFive(RFive&& other) noexcept // Конструктор перемещения, noexcept - для
оптимизации при использовании стандартных контейнеров
    {
        delete[] cstring;
        cstring = other.cstring;
        other.cstring = nullptr;    }

    RFive& operator=(const RFive& other) // Оператор присваивания
копированием (copy assignment)
    {
        char* tmp_cstring = new char[std::strlen(other.cstring) + 1];
        std::strcpy(tmp_cstring, other.cstring);
        delete[] cstring;
        cstring = tmp_cstring;
        return *this;    }

    RFive& operator=(RFive&& other) noexcept // Оператор присваивания
перемещением (move assignment)
    {
        delete[] cstring;
        cstring = other.cstring;
```

```

        other.cstring = nullptr;
        return *this;    }

// Также можно заменить оба оператора присваивания следующим оператором
// RFive& operator=(RFive other)
// {
//     std::swap(cstring, other.cstring);
//     return *this;
// }
};

```

Спецификаторы default и delete

Проблема: если есть конструкторы, но например нет конструктора по умолчанию его нужно определять самостоятельно. Это утомительно.

Спецификаторы default и delete можно указывать вместо тела метода.

```

class Foo
{
public:
    Foo() = default;
    Foo(int x) { /* ... */ }
};

```

Спецификатор **default** означает реализацию по умолчанию и может применяться только к специальным функциям-членам:

- конструктор по-умолчанию;
- конструктор копий;
- конструктор перемещения;
- оператор присваивания;
- оператор перемещения;
- деструктор.

Спецификатор **delete** помечают те методы, работать с которыми нельзя. Раньше приходилось объявлять такие конструкторы в приватной области класса.

```

class Foo
{
public:
    Foo() = default;
    Foo(const Foo&) = delete;
    void bar(int) = delete;
    void bar(double) {}
};
// ...
Foo obj;
obj.bar(5);      // ошибка!
obj.bar(5.42);  // ok

```

Если программа ссылается явно или неявно на delete функцию — ошибка на этапе компиляции. Запрещается даже создавать указатели на такие функции.

Можно также запретить оператор new:

```
class Foo
{
public:
    void *operator new(std::size_t) = delete;
    void *operator new[](std::size_t) = delete;
};
// ...
Foo* ptr = new Foo; // ошибка!
```

<https://stackoverflow.com/questions/6502828/what-does-default-mean-after-a-class-function-declaration>

Порядок вызова конструкторов и деструкторов

В классе может быть описано несколько конструкторов преобразования и конструкторов с двумя и более параметрами. Единственное требование к ним: не может быть двух конструкторов с одинаковым (по типам) набором параметров, в частности, не может быть двух конструкторов умолчания.

При создании объекта конструкторы вызываются в следующем порядке:

1) Конструкторы базовых классов, если класс для создаваемого объекта является наследником других классов в порядке их появления в описании класса. Если в списке инициализации описываемого класса присутствует вызов конструктора преобразования (или конструктора с двумя и более параметрами) базового класса, то вызывается конструктор преобразования (или конструктор с двумя и более параметрами), иначе вызывается конструктор умолчания базового класса.

2) Конструкторы умолчания всех вложенных информационных членов, которые не перечислены в списке инициализации, и конструкторы преобразования, копирования и конструкторы с двумя и более параметрами всех вложенных информационных членов, которые перечислены в списке инициализации. Все перечисленные в данном пункте конструкторы (умолчания, преобразования, копирования, с двумя и более параметрами) вызываются в порядке описания соответствующих информационных членов в классе.

3) Собственный конструктор. Такая последовательность вызова конструкторов логически обосновывается тем, что в момент выполнения собственного конструктора все информационные поля должны быть уже проинициализированы.

Деструкторы вызываются в обратном порядке:

1) Собственный деструктор. В момент начала его работы поля класса еще не очищены, и их значения могут быть использованы в теле деструктора.

2) Деструкторы вложенных объектов в порядке, обратном порядку их описания.

3) Деструкторы базовых классов в обратном порядке их задания.

Свод ситуаций, при которых вызываются конструкторы и деструкторы

Обычный конструктор (не копирования) вызывается:

1) при создании объекта (при обработке описания объекта);

2) при создании объекта в динамической памяти (с использованием операции new). При этом в динамической памяти предварительно отводится необходимый по объему фрагмент памяти;

3) при композиции объектов;

4) при создании объекта производного класса.

Замечание. Если в программе создается не единичный (скалярный) объект, а массив объектов, то в соответствующем классе (структуре) должен быть конструктор умолчания (явно или неявно сгенерированный). Действительно, при объявлении массива объектов или создании массива в динамической памяти указывается размерность массива, что несовместимо с заданием параметров для конструктора преобразования или конструктора с двумя и более параметрами:

```
class X{ . . . };
X* x1 = new X[10];
```

Как уже было указано, если в классе явно описан хотя бы один конструктор, то конструктор умолчания не генерируется системой неявно. Для удаления такого массива должна применяться векторная форма операции delete, при использовании которой деструктор вызывается для каждого элемента массива:

```
delete[] x1;
```

Конструктор копирования вызывается:

1) при инициализации создаваемого объекта:

```
box a(1,2,3); // Конструктор преобразования
box b = a;
// вызов конструктора с тремя параметрами
// инициализация
```

2) при инициализации временным объектом:

```
box c = box(3,4,5);
```

3) при передаче параметров-объектов в функцию по значению:

```
int f(box b);
```

4) при возвращении результата работы функции в виде объекта.

```
box f ();
```

Примечание 1. Если используется оптимизирующий компилятор, то при обработке инициализации вида:

```
box c = box(3,4,5)
```

временный объект не создается, и вместо конструктора копирования используется конструктор с тремя параметрами:

```
( box c(3,4,5) ).
```

Примечание 2. Если при возвращении результата работы функции в виде объекта тип возвращаемого значения не совпадает с типом результата работы функции, то вызывается не конструктор копирования, а конструктор преобразования или функция преобразования (описывается ниже). Данное преобразование выполняется, если оно однозначно. Иначе фиксируется ошибка.

Пример:

```

class Y;
class X{
. . .
public:
X(const Y& y1);
. . .};

class Y{
. . .
public:
X f(){
. . .
return *this;}
};

X::X(const Y& y1){. . .}

```

Деструктор вызывается:

1) при свертке стека – при выходе из блока описания объекта, в частности, при обработке исключений (при выходе из try-блока по оператору throw, try-блоки описываются далее), завершении работы функций;

2) при уничтожении временных объектов – сразу, как только завершится конструкция, в которой они использовались;

3) при выполнении операции delete для указателя, получившего значение в результате выполнения операции new. После выполнения деструктора освобождается выделенный для объекта участок памяти;

4) при завершении работы программы при уничтожении глобальных и статических объектов.

Примечание. Все правила описания и использования конструкторов и деструкторов применимы и для структур.

Можно вызывать один конструктор из другого.

Вопросы

Как запретить создание объекта на основе уже существующего?

Как запретить любой другой способ создания объекта?

Зачем нужны конструкторы перемещения? В чём их отличие от к. копирования?

Статические переменные и функции

Статические переменные являются долговременными переменными, существующими на протяжении функции или файла. Они отличаются от глобальных переменных, поскольку не известны за пределами функции или файла, но могут хранить свои значения между вызовами. Данная возможность оказывается очень полезной тогда, когда необходимо написать универсальные функции и библиотеки функций, которые могут использоваться программистами. Поскольку эффект использования `static` на локальных переменных отличается от эффекта на глобальных переменных, то мы рассмотрим их по отдельности.

Статические члены класса

- Как хранить информацию о числе объектов класса?
- Как организовать доступ к общему ресурсу для всех классов?

Эти задачи можно решить используя обычные глобальные переменные. Но тогда будет нарушен принцип инкапсуляции. Переменные можно менять как угодно, класс их не контролирует. Выход — поместить их в класс.

Информационные члены класса представленные в одном экземпляре для всех объектов данного класса называются **статическими**. Объявление таких членов класса начинается с ключевого слова `static`.

Помечены как статические могут быть и методы. Такие методы не могут использовать не статические информационные члены класса.

Статические члены класса имеют отношение не к объектам, а к классам.

Часто статические переменные называют переменными класса, а не статические переменные - переменными экземпляра.

Информационные члены класса, которые могут быть представлены в единственном экземпляре для всех объектов данного типа, в случае такого представления называются статическими членами. Они не являются частью объектов этого класса и размещаются в статической памяти.

Для всех объектов, созданных на основе класса, содержащего статический член, существует только одна копия этого члена. *Примером такого статического члена является счетчик числа созданных объектов данного класса.* Такой счетчик может существовать только в отрыве от всех экземпляров объектов данного класса, в то же время работать с ним обычно приходится одновременно с вызовом обычных методов, например, конструкторов объектов, поэтому описывать счетчик удобнее именно как элемент класса.

Статическими могут быть не только информационные члены класса, но и его методы. Статический метод не может использовать никакие нестатические члены класса, так как, являясь частью класса, а не объекта, он не имеет неявного параметра `this`.

Обращение к статическим членам. Поскольку статический метод не использует специфического содержимого конкретного объекта, то обращение к нему может осуществляться не только с использованием идентификатора объекта, но и с использованием идентификатора класса:

```
имя_класса :: имя_функции (фактические_параметры);
или
имя_объекта.имя_члена_класса
```

Одно из очевидных применений статических методов – манипуляция глобальными объектами и статическими полями данных соответствующего класса.

Примечание 1. Статические методы класса не могут вызывать нестатические, так как последние имеют доступ к данным конкретных объектов. Обратное допустимо: нестатические методы могут вызывать статические методы.

Примечание 2 . Статический метод класса может создавать объекты данного и любого другого класса. Это можно использовать, в частности, если необходимо в программе запретить создание объектов простым объявлением или с использованием операции `new` . В этом случае конструктор и деструктор помещаются в закрытую область класса, а для создания и уничтожения объекта используются специальные статические методы, которые можно вызывать, не имея ни одного объекта.

Примечание 3. Статические методы класса не могут быть виртуальными и константными (`inline` –функциями быть могут).

Пример:

```
#include <iostream>
using namespace std;
class X{
    X(){}
    ~X(){}
public:
    static X& createX(){
        X* x1 = new X;
        cout << "X created" << endl;
        return *x1;
    }
    static void destroyX(X& x1){
        delete &x1;
        cout << "X destroyed" << endl;
    }
};

int main(){
    X& xx1 = X::createX();
    . . .
    X::destroyX(xx1);
}
```



```
    return 0;
}
```

Статические информационные члены класса, даже находящиеся в закрытой области (а это характерно для информационных членов класса в соответствии с принципом инкапсуляции), **необходимо объявить дополнительно вне класса** (с возможной инициализацией):

```
тип_переменной имя_класса :: идентификатор = инициализатор;
```

Это связано с тем, что память для статического объекта должна быть выделена до начала работы программы. В то же время при обработке описания класса до создания конкретных объектов никакие области памяти не отводятся. В дальнейшем прямое обращение к статическим информационным членам, находящимся в закрытой секции, недопустимо. Если инициализация не нужна, то все равно необходимо дополнительное объявление статического члена вне класса для резервирования памяти для статического члена. В противном случае на этапе сборки исполняемого модуля будет выдана ошибка о неразрешенной внешней ссылке.

Пример:

```
class B {
    static int i;
};
// статический информационный член
// класса
public:
    static void f(int j){ // статический метод
        i = j;
    }
int B::i = 10; // Нужно указывать тип!

дополнительное внешнее определение
статической переменной с
инициализацией статического
информационного члена класса b.

int main(){
    B a;
    B::f(1);
    return 0;
}
```

Примеры

Пример с количеством экземпляров класса. С запретом на создание большого их числа?

Контроль доступа к общим ресурсам.

Access.h

```
enum status {shared, in_use, locked, unlocked};
class Access{
```

```

        static enum status acs;
        int exmpl;
    public:
        public:
            static void set_access (enum status a) {
                // this->acs = a; // Неправильно! this недоступен в статических
методах
                // exmpl = 10; // Нельзя обращается в стат. методах к не статическим
полям
                acs = a;
            }
            static enum status get_access() {return acs;}
            void foo();
};

```

Access.cpp

```

void Access::foo(){
    if (acs == unlocked){ // Можно обращается к статическим членам класса
        // ...
    }
}
// Нужно обязательно объявить вне класса, как глобальную переменную.
// static указывать при объявлении не нужно.
enum status Access::acs = unlocked;

```

Вы пишете класс логер. С помощью метода getLogger() вы возвратили логер. Но при этом все логеры должны писать в один и тот же файл, как быть? Можно сделать static HANDLE hFile, и все объекты будут иметь доступ к этому приватному полю.

[https://ru.wikipedia.org/wiki/Одиночка_\(шаблон_проектирования\)](https://ru.wikipedia.org/wiki/Одиночка_(шаблон_проектирования))

Статическое поле класса — глобальная переменная на которую наложены ограничения области видимости класса.

Константные члены класса. Модификатор const

- Делают код более строгим → меньше ошибок.

Все информационные члены класса, не являющиеся статическими информационными членами, можно представлять, как данные, доступные методу класса через указатель `this` (в случае необходимости этот указатель можно употреблять явно).

Если необходимо запретить методу изменять информационные члены объектов класса, то при его описании используется дополнительный модификатор `const`:

```
Тип_возвр_знач Имя_функции ( формальные_параметры ) const { тело _ функции }
```

Описанные таким образом методы класса называются константными.

Тем не менее, статические члены класса могут изменяться такой функцией, так как они являются частью класса, но не объекта.

Если же статические информационные члены класса имеют дополнительный модификатор `const`, то они не могут изменяться никакими методами класса.

Примечание. В некоторых изданиях данные, доступные через указатель `this`, рассматриваются как неявные аргументы метода класса. Конечно, можно рассматривать глобальные переменные также в качестве неявных параметров для всех функций, а не только методов класса. Тем не менее, если дать определение неявных аргументов метода класса как данных, доступных через указатель `this`, то вышеописанное можно сформулировать следующим образом: константные методы не могут изменять свои неявные аргументы.

Таким образом, если объект типа описанного класса является константным объектом, то есть он объявлен с модификатором `const`, это означает, что изменение его состояния недопустимо. В таком случае все применяемые к этому объекту методы (кроме конструкторов и деструктора) должны иметь модификатор `const`. Данное требование является обязательным независимо от наличия или отсутствия информационных членов в классе.

Для защиты от изменения передаваемых фактических параметров в теле функции соответствующие формальные параметры также объявляются с модификатором `const`:

```
const Тип_параметра идентификатор
```

Объявление методов класса и формальных параметров с модификатором `const` называется **контролем постоянства**.

Если необходимо запретить изменение объекта в пределах его области видимости, то при объявлении объекта используется ключевое слово `const`, например:

```
const X x2 = x1;
```

Примечание. При объявлении объекта с модификатором `const` объект должен быть обязательно инициализирован. Объект пользовательского типа может быть инициализирован неявно (например, с помощью конструктора класса), если в описании типа объекта указаны

параметры, принимаемые по умолчанию при отсутствии в объявлении объекта явных параметров.

Пример:

```
#include <iostream>
using namespace std;

class A {
    static int i;
    void f() const { // модификатор const, запрещающий
        // изменять неявные аргументы,
        // необходим в связи с тем, что
        // имеется объект данного класса с
        // описателем const
        if (i < 0) g(i);
        cout << "f()" << endl;
    }
public:
    void g(const int & n) const {
        // модификатор const для
        // параметра-ссылки необходим в
        // связи с использованием
        // числовой константы 2 при вызове
        // данного метода для объекта:
        // a.g(2)
        i = n;
        f();
        cout << "g()" << endl;
    }
};

int A::i = 1;
// инициализация статической переменной

int main(){
    const A a = A();
    a.g(2);
    return 0;
}
```

Друзья классов

- Что если нужно из одного класса\функции получить доступ к закрытым членам другого?

Имеется ряд ситуаций, когда объекту одного класса необходимо иметь прямой доступ к закрытым членам объекта другого класса без использования методов-селекторов. Для этого в языке C++ введена концепция друзей и специальное ключевое слово **friend**.

Друг класса – это функция, не являющаяся членом класса, но имеющая доступ к его закрытым и защищенным членам.

Друзья класса объявляются в самом классе с помощью служебного слова `friend`. в любой области доступа.

Другом класса может быть обычная функция, метод другого класса или другой класс (при этом каждый его метод становится другом класса).

Пример:

```
class B;
// предварительное объявление идентификатора
// b как идентификатора типа данных

class X {
    int ial;
public:
    X(){
        ial = 0;
    }
    int func1(B& bb);
};

class B {
    int b1;
public:
    friend int X::func1(B & bb);
    B(){
        b1 = 1;
    }
};

int X::func1(B & bb){
    ial = ial + bb.b1;
    return ial;
}

int main(){
    int i1;
    B b2;
    X a2;
    i1 = a2.func1(b2);
    return 0;
}
```

Примечание. Несмотря на предварительное объявление идентификатора B, его можно использовать в описании класса X, находящемся перед описанием класса B, только в описании формального параметра в прототипе функции (func1). Саму функцию func1 необходимо описывать вне класса X после описания классов B и X, используя операцию разрешения области видимости '::' с квалификатором X. Неправильным будет следующее описание функции func1:

```
class B;
class X {
    int ia1;
public:

    X(){
        ia1 = 0;}

    int func1(B & bb){
        ia1 = ia1 + bb.b1;
        return ia1;
    }
};
// ОШИБКА!

class B {
    int b1;
public:
    friend int X::func1(B & bb);

    B(){
        b1 = 1;}
};

int main(){
    int i1;
    B b2;
    X a2;
    i1 = a2.func1(b2);
    return 0;
}
```

Другом можно объявить и весь класс: friend class X;

На друзей не действуют модификаторы доступа.

Другом класса может быть не только метод другого класса, но и внешняя функция. Кроме того, возможна дружелюбность сразу для нескольких классов. Это необходимо, например, в случае организации взаимодействия нескольких объектов разных классов, когда функция, обеспечивающая взаимодействие, должна иметь доступ к закрытым компонентам одновременно нескольких объектов. Объявить функцию методом одновременно нескольких классов невозможно, поэтому в стандарте языка C++ предусмотрена возможность объявлять внешнюю по отношению к классу функцию дружелюбной данному классу. Для этого необходимо в теле класса объявить некоторую внешнюю по отношению к классу функцию с использованием ключевого слова friend:

friend имя_функции (список_формальных_параметров);

Пример:

```
class B;

class D {
    int x;
    . . .
    friend void func(B &, D &); //функция дружелюбна классу D
    . . .
};

class B {
    int y;
    . . .
    friend void func(B &, D &); // функция дружелюбна классу B
    . . .
};

void func(B & b1, D & d1) {
    cout << d1.x + b1.y; // дружелюбная функция имеет
    // доступ к закрытым
    // компонентам обоих классов
}
```

Наследование

- Упорядочивает классы
- Упрощает повторное использование кода
- Позволяет преобразование одного класса в другой.

Базовый класс (предок) — класс на основе которого строится определение нового класса - **производного класса (потомка)**.

В C++ существуют три способа создать структурированный тип данных:

- структура `struct` (следует использовать для хранения данных.)
- объединение `union` (когда нужен доступ к данным как различным типам)
- класс `class` (хранение и работа с данными)

Определение производного класса

```
class Имя_Производного_Класса : спецификатор доступа Имя_Базового_Класса
{...}
```

Можно записывать в объект базового класса объект производного, лишнее будет срезано.

`Protected` — разумно для методов, а не полей.

Перегрузка vs переопределение

Переопределение — метод имеет ту же сигнатуру.

Нельзя использовать метод базового класса если он был переопределён в наследнике. Однако он всё ещё содержится в наследнике. Чтобы сделать это возможным — явно подключаем методы из базового класса:

```
class D : public Base{
...
public:
    void foo();
    using Y::foo(int);
...}

D d;
d.Base::foo()
```

Структуры тоже можно наследовать как и классы.

Конструкторы по умолчанию не наследуются. Чтобы наследовать нужно в производном классе указать:

```
using BaseClass::BaseClass
BaseClass::BaseClass(); // или вызывать явно
```


Не наследуются:

- Конструкторы
- Деструктор
- Операция присваивания

Если программист не описал в производном классе деструктор, он формируется по умолчанию и вызывает деструкторы всех базовых классов.

Единственный способ использования конструктора базового класса – список инициализации. Он записывается при описании конструктора производного класса:

```
DeliveredClass (formal_params) : BaseClass (actual_params) {...}
```

Преобразования

Преобразование объектов

Можно выполнить неявное преобразование производного класса в базовый, но нельзя наоборот.

```
class A {
    public:
        int x;
        A() {}
};

class B: public A{
    public:
        int y;
        B() {}
};

A a;
B b;
a = b;
a.y = 10; // Ошибка! В классе A нет поля Y
b = a;    // Так нельзя.
b = (B)a; // Так тоже нельзя
```

Преобразование указателей на объекты

Безопасным приведением является приведение указателя на объект производного класса к указателю на объект базового типа.

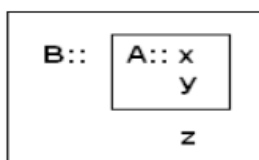
Пусть A и B – типы из предыдущего примера. Тогда

```
A a1;
A* pa;
B b1;
B* pb;
```

```

pb = &b1;
pa = pb; // Указателю pa присваивается адрес объекта
// b. Т.к. указатель pa описан как указатель
// на объект типа A, то с его помощью видна
// только та часть объекта b, которая
// структурно соответствует типу A.
pa = &a1;
pb = (B*)pa; // Допустимое, но небезопасное явное преобразование указателя
на объект базового типа к указателю на объект производного типа. Через указатель
на
    объект типа B можно обращаться к информации, которая присутствует в
    // типе B. Однако, в структуре объекта a1
    // базового типа A отсутствуют
    // дополнительные члены производного
    // типа.
pb -> y; // Ошибочное обращение к таким членам после указанного
преобразования
    //(ошибка во время исполнения программы).

```



Модификаторы доступа

Для классов наследование по умолчанию — private

struct — public

Доступ к членам базового класса в производном.

	Модификатор наследования		
Модификатор базового класса	public	protected	private
public	public	protected	нет
protected	protected	protected	нет
private	private	private	нет

Стоит использовать private protected наследование с осторожностью. Это полезно когда нужно подменить виртуальные методы.

Если не указан тип наследования, то тип наследования по умолчанию определяется описанием типа наследника. Если тип-наследник описывается классом, то тип наследования – закрытый (private), если же это структура, то наследование по умолчанию будет открытым (public).

Следует по возможности скрывать поля предка в потомке, используя для работы с ними соответствующие методы.

Перекрытие имен (overriding)

В производном классе могут использоваться имена членов класса, перекрывающие видимость таких же имен в базовом классе (overriding). При перекрытии имен при работе с объектом через указатель будет исполняться тот метод, который содержится в классе, используемом в объявлении указателя, независимо от типа объекта, на который указывает указатель. Это статическое связывание.

Пример:

```
class A {
public:
    void f(int x){
        cout << "A::f" << '\n'; }
};

class C: public A{
public:
    void f(int x){
        cout << "C::f" << '\n';}
};

A a1;
A* pa;
C c1;
C* pc;
pc = &c1;
// Вызов определяется по типу указателя
pc -> f(1); // C::f
pa = pc;
pa -> f(1); // A::f – несмотря на то, что pa указывает
на типа класс C.

pc = (C*)& a1;
pc -> f(1); // C::f
```

Члены базового класса с именами, совпадающими с именами членов производного класса, доступны в производном классе. Для доступа к ним необходимо указывать квалификатор (имя базового класса) с использованием операции '::', так как данные члены находятся в доступной области видимости, которая не совпадает с текущей областью видимости. Также метод базового класса доступен через указатель класса-наследника при условии использования квалификатора.

Пример:

```
class A {
public:
    void f(int x){cout<<"A::f"<<"\n";}
    int x = 42;
};
```

```

class C: public A{
public:
    int x = 99;
    void f(int x){
        cout << "C::f" << '\n';}

    void g(){
        f(1);
        A::f(1);}
};

C c1;
c1.x      // 99
c1::A.x   // 42
C* pc;
pc=&c1;
pc->A::f(1); // метод базового класса можно вызывать либо так
pc->f(1);
pc->g();      // Либо поместив вызов в один из методов

```

Таким образом, при перекрытии методы базового класса не «затираются» в классе-наследнике. Они доступны через квалификатор.

Множественное наследование

Создание нового класса на основе нескольких. Особенно полезно если наследование происходит от интерфейсов.

Следует избегать конфликтов и неоднозначностей при наследовании. Рекомендуется наследоваться от интерфейсов и классов с данными.

Наследование vs агрегация

Наследование - сложный механизм. Особенно, наследование многоуровневое и множественное. Поэтому везде, где можно применить наследование рассматривать ещё и агрегацию. Например агрегация может быть выглядеть логичной для случая если нужно описать сущность «автомобиль», включающую «двигатель». К тому же, относительно легко будет заменить один «двигатель» на другой, если это потребуется. Наследование может как упростить, так и усложнить интерфейс.

Множественное наследование возникает, когда имеется несколько базовых типов и один класс – наследник..

```
class X { . . . };
class Y { . . . };
class Z: public X, public Y { . . . };
```

При описании производного класса каждый базовый класс имеет свой собственный описатель типа наследования (явно указанный или неявно предполагаемый).

Видимость при множественном наследовании.

При множественном наследовании возникает проблема неоднозначности из-за совпадающих имен в базовых классах. Именно поэтому лучше наследоваться от интерфейсов и классов-контейнеров.

Пример:

```
struct X {
    int i1;
    int jx;
};
struct Y {
    int i1;
    int jy;
};
struct Z: X, Y {
    int jz;
};

...
Z z1;
z1.i1 = 5;
// ошибка – неоднозначность: член i1 наследуется как из базового типа X,
так и из базового типа Y.
}
```

Тем не менее, **данная неоднозначность проявляется не при объявлении объекта типа-наследника, а при использовании его членов**, имеющих в нескольких базовых типах. Эту неоднозначность можно обойти при помощи операции разрешения области видимости:

```
z1.X::i1 = 5
```

Таким образом, в тип-наследник попадают все члены базовых типов. Но повторяющиеся имена необходимо сопровождать квалификатором базового типа.

Полиморфизм

Полиморфизм - обработка разных типов данных одним способом.

Механизмы:

- перегрузка (стат. Полиморфизм — на этапе компиляции)
- виртуальные методы (динамический П, П времени выполнения)
- универсальный
 - параметрический

представляет собой свойство семантики системы типов, позволяющее обрабатывать значения разных типов *идентичным* образом, то есть исполнять физически *один и тот же* код для данных разных типов
 - включения (или подтипов)
- ad hoc
 - перегрузка (статический?)
 - приведение типов

Ad hoc — латинская фраза, означающая «к этому, для данного случая, для этой цели»

Ad hoc, *ад хок* (от лат. *ad hoc* — к этому, для данного случая, для этой цели) — способ решения специфической проблемы или задачи, который не адаптируется для решения других задач.

Статический полиморфизм

Статический полиморфизм реализуется с помощью перегрузки функций и операций. Под перегрузкой функций в C++ понимается описание в одной области видимости нескольких функций с одним и тем же именем. О перегрузке операций в C++ говорят в том случае, если в некоторой области видимости появляется описание функции с именем `operator <обозначение_операции_C++>`, задающее еще одну интерпретацию заданной операции.

Перегрузка операторов (Operator overloading)

Относится к полиморфизму. Синтаксический сахар.

<https://habrahabr.ru/post/308890/>

https://ru.wikipedia.org/wiki/Перегрузка_операторов — пример.

- Что такое перегрузка? Примеры.
- Что такое оператор? Примеры.
- В чём отличия оператора от функции?
 - Запись вида «<операнд1> <знакОперации> <операнд2>» принципиально аналогична вызову функции «<знакОперации> (<операнд1>, <операнд2>)

Использование операторов позволяет сделать код более естественным. Например для сложения двух объектов не нужно будет вызывать отдельной функции или метода:

```
o3 = plus(o1, o2); // структурный стиль
o3 = o1.plus(o2);
```

Предполагается, что метод `plus` не изменяет сам объект, а возвращает новый, сумму текущего и переданного в параметре.

Такая запись выглядит лаконичнее и понятнее:

```
o3 = o1 + o2;
```

Но оператор сложения определён только для числовых типов. Поэтому нужно реализовать новую версию этого оператора, что будет работать с нужными классам.

Перегрузка оператора - один из способов реализации полиморфизма, заключающийся в возможности одновременного существования в одной области видимости нескольких различных вариантов применения оператора, имеющих одно и то же имя, но различающихся типами параметров, к которым они применяются.

Два способа реализовать перегрузку:

- Реализовать оператор как функцию. Операнды — параметры функции.
- Реализовать оператор как метод. Левым операндом (аргументом оператора) всегда выступает объект `*this`. Остальные — параметры метода.

Перегрузка оператора как функции

В C++ для перегрузки оператора нужно описать функцию специального вида:

```
возвр.тип operator X (параметры);
```

X — символ, которым обозначается оператор, например +.

Число параметров функции должно соответствовать аности оператора. Например для бинарных операторов должно быть два параметра.

Перегрузка оператора как метода

```
возвр.тип Имякласса::operator X (параметры);
```

Когда перегруженный оператор является методом класса, тип первого операнда должен быть этим классом(всегда *this), а второй должен быть объявлен в списке параметров.

Следующий код

```
T a, b, c;  
c = a + b
```

будет транслирован компилятором в

```
c = a.operator+(b)
```

Операторы-методы не статичны, за исключением операторов управления памятью.

В C++ можно выделить четыре типа перегрузок операторов:

1. Перегрузка обычных операторов

1. Арифметические

1. Унарные: префиксные + - ++ --; постфиксные: ++ --
2. Бинарные + - * / % += -= *= /= %=

2. Битовые

1. Унарные ~
2. Бинарные & | ^ >> << &= |= ^= <=> >=>

3. Логические

1. Унарные !
2. Бинарные && ||
3. Сравнения > < >= <= == !=

4. Остальные = & * , . :: ->* -> () []

(type) оператор приведения типа

Тернарные оператор x

2. Перегрузка операторов преобразования типа (type)

3. Перегрузка операторов аллокации и деаллокации new и delete, new [], delete []

4. Перегрузка литералов operator""

, - оператор
последовательного
выполнения

. :: - нельзя
перегружать

Ограничения перегрузки

- 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`.
- Операторы `=` `->` `[]` `()` могут быть перегружены только как методы (функции-члены), но не как функции.

Рекомендации

Перегружать операторы стоит только в том случае, если их смысл интуитивно понятен. Например перегрузить оператор `+` для класса `Клиент_банка` будет плохой идеей. Не очевидно, что в результате этого «сложения» должно получиться и как будет происходить процесс сложения.

Реализуйте унарные операторы и бинарные операторы типа “`X=`” в виде методов класса, а прочие бинарные операторы — в виде свободных функций. Так стоит делать потому, что оператор-метод всегда вызывается для левого операнда.

<https://stackoverflow.com/questions/4622330/operator-overloading-member-function-vs-non-member-function>

Примеры

Операторы `++` (постфиксный и префиксный)

```
class Number {
public:
    Number& operator++ ();    // (++x) prefix ++: no parameter, returns a
reference
    Number operator++ (int); // (x++) postfix ++: dummy parameter, returns
a value
};
```

```
#include <iostream>
using namespace std;

class Time {
private:
    int hours;           // 0 to 23
    int minutes;         // 0 to 59

public:
    ...

    // overloaded prefix ++ operator
```

```

    // Должен возвращать текущий, изменённый объект
    Time& operator++ () {
        ++minutes; // increment this object
        if(minutes >= 60) {
            ++hours;
            minutes -= 60;
        }
        return *this;
    }

    // overloaded postfix ++ operator
    Time operator++( int ) {

        // save the original value
        Time T(hours, minutes);

        // increment this object
        ++minutes;

        if(minutes >= 60) {
            ++hours;
            minutes -= 60;
        }

        // return old original value
        return T;
    }
};

int main() {
    Time T1(11, 59), T2(10,40);

    Time ++T1; // increment T1
    T1.displayTime(); // H: 12 M:0

    ++T1; // increment T1 again
    T1.displayTime(); // H: 12 M:1

    T2++; // increment T2
    T2.displayTime(); // display T2
    T2++; // increment T2 again
    T2.displayTime(); // display T2
    return 0;
}

```

Указатели на методы

Необходимо объявить указатель на метод класса

```
class MyClass{
};
```

Определить тип указателя на метод класса можно двумя способами. Понятным и не очень. Эти способы аналогичны объявлению типа указателей на функции. Начнём со не очень понятного способа. MethPtr — объявляемый тип.

```
typedef возвр.тип (ИмяКласса:: * MethPtr) (типы параметров) [const];
```

Этот способ неудобен тем, что тип (MethPtr) записывается внутри трудночитаемого выражения. Модификатор const может отсутствовать.

Использование оператора using для создания синонима (читай: нового типа) делает объявление понятнее:

```
using MethPtr = возвр.тип (ИмяКласса:: * ) (типы параметров) [const];
```

Теперь можно объявлять переменные типа «указатель на метод конкретного класса». Подойдёт любой метод удовлетворяющий объявленной сигнатуре. Чтобы получить адрес метода нужно использовать оператор &.

При вызове метода адрес нужно разыменовывать с помощью операторов .* или ->.*.

Указатель на константный метод без параметров класса string из стандартной библиотеки

```
using MethPtr = size_t (string::*)() const;

MethPtr mp1 = &string::size;

string s = "what is my size?";

size_t len = (s.*mp1)();
```

Самый полезный способ использования типа «указатель на метод» - это в параметрах функции.

Пример

Статический полиморфизм

Виртуальные методы

Виртуальный деструктор

Абстрактные классы

Виртуальные методы

stepik

- Реализуют динамический полиморфизм.
- Позволяют выбирать программе (во время компиляции), какой из методов в иерархии наследования вызывать
- Упрощает код: один набор методов на иерархию классов

https://ru.wikipedia.org/wiki/Виртуальный_метод

Виртуальный метод - в объектно-ориентированном программировании метод (функция) класса, который может быть переопределён в классах-наследниках так, что конкретная реализация метода для вызова будет определяться во время исполнения.

Базовый класс может и не предоставлять реализации виртуального метода, а только декларировать его существование. Такие методы без реализации называются «**чистыми виртуальными**» ([англ. pure virtual](#)) или **абстрактными**.

Класс, содержащий хотя бы один такой метод, тоже будет **абстрактным**.

Нельзя создавать экземпляры класса, но можно создавать указатели и ссылки на такие объекты. Такой класс нужен только для того, чтобы определить на его основе производные классы.

В произвольных классах виртуальные методы должны быть определены.

Чистые виртуальные методы могут иметь реализацию. Но в производном классе, при вызове, нужно явно указывать что вызывается метод базового класса.

```
baseclass::my_abstract_method()
```

Виртуальный деструктор

Во время компиляции будет определён и вызван нужный деструктор.

Если виртуальные деструкторы не используются, то может быть утечка памяти: удалятся только поля базового класса. Рассмотрен случай, когда используется указатель типа базового класса на экземпляр производного класса.

Если есть наследование, то скорее всего нужен виртуальный деструктор.

Интерфейс — класс без полей, с абстрактными методами. Деструктор в таких классах должен иметь реализацию.

Как определить какой метод вызывать, если для всей иерархии классов они одинаковы?

Для каждого класса, имеющего хотя бы один виртуальный метод, создаётся *таблица виртуальных методов*.

Каждый объект хранит указатель на таблицу своего класса. Для вызова виртуального метода используется такой механизм: из объекта берётся указатель на соответствующую таблицу виртуальных методов, а из неё, по фиксированному смещению,— указатель на реализацию метода, используемого для данного класса. При использовании множественного наследования ситуация несколько усложняется за счёт того, что таблица виртуальных методов становится нелинейной.

override и final

Виртуальные базовые классы

При многоуровневом множественном наследовании базовые классы могут быть получены от общего предка. В этом случае итоговый производный класс будет содержать несколько подобъектов общего предка:

```
class W
{ . . . };
class X: public W
{ . . . };
class Y: public W
{ . . . };
class Z: public X, public Y
{ . . . };
```

Если необходимо, чтобы общий предок присутствовал в итоговом производном классе в единственном экземпляре (например, если необходимо, чтобы функции классов X и Y в классе Z использовали общие информационные члены класса W, или для экономии оперативной памяти), то наследование базовых классов от общего предка описывается с использованием виртуального наследования:

```
class W
{ . . . };
class X: public virtual W
{ . . . };
class Y: public virtual W
{ . . . };
class Z: public X, public Y
{ . . . };
```

Интерфейсы

Указанная неоднозначность при множественном наследовании отсутствует, если все базовые классы являются абстрактными классами без информационных членов и содержат только открытые чистые виртуальные функции. Такие базовые классы называются интерфейсами. Действительно, если с объектом работать через указатель такого класса, то набор чистых виртуальных функций данного класса определяет, какие методы объекта доступны через этот указатель.

Класс-наследник классов-интерфейсов, если он не является абстрактным классом (то есть, не содержит ни одной чистой виртуальной функции), называется **классом реализации**. На основе класса реализации создаются конкретные объекты.

Работа с такими объектами осуществляется с использованием указателей типа классов-интерфейсов.

Раннее и позднее связывание.

То же, что и статическая и динамическая типизация.

Статическая типизация — определение типа на этапе компиляции. Динамическая — во время выполнения.

```
class Base{
public: void f(){ cout<< "base" << endl; }
};

class Delivered {
public: void f(){ cout<< "delivered" << endl; }
};

// Очевидно какой метод будет вызван.
Base b;
b.f()
Delivered d;
d.f();

void foo(Base b){
b.f()};
```

Раннее связывание	Позднее связывание
Delivered d; foo(d); // base	Delivered d; foo(d); // delivered
Компилятор уже всё решил. Раз параметр типа Base, значит вызываем соответствующий метод	Компилятор не стал ничего решать: вдруг передадут производный класс? Пусть программа сама решает всё на этапе выполнения.

Java, PHP, JavaScript используют позднее связывание.

C++ использует и раннюю и позднюю типизацию. Объявление методов как virtual даёт знать компилятору, что нужно использовать позднее связывание.

Динамическая информация о типе (RTTI).

RTTI (Run Time Type Identification) – механизм безопасного преобразования типов объектов. Этот механизм включает:

- `dynamic_cast` – операция преобразования типа указателя или ссылки на полиморфные объекты
- `typeid` – операция определения типа объекта
- `type_info` – структура, содержащая информацию о типе объекта (`typeinfo.h`).

Для текстового представления имени типа объекта в классе `type_info` имеется функция `name()`.

Примечание. Так как операция `typeid` возвращает значение типа `const type_info &`, то нецелесообразно явно создавать объект такого типа. Обычно такой объект используется неявно, то есть, используется создаваемый временный объект.

Синтаксис операции динамического приведения типа (`dynamic_cast`):

```
dynamic_cast < тип_результата > ( выражение );
```

тип_результата – указатель или ссылка,

выражение – указатель, если тип_результата является указателем, или объект (или ссылка на объект), если тип_результата является ссылкой.

Пример:

```
class A{
public:
    virtual void fx (){
        cout << "A::fx" << '\n';
    }
};

class B: public A{
public:
    void fx (){
        cout << "B::fx" << '\n';
    }
};

void f2(A* ptr){
    B* dptr = dynamic_cast<B*> (ptr);
    cout<<"type of pointer dptr: " <<typeid(dptr).name()<<'\n';
}

A* p1 = new A;
f2 (p1);
```

В результате работы данной программы будет выведена следующая строка:

type of pointer dptr: class B *

Примечание. Для обеспечения возможности использования механизма RTTI в компиляторе, как правило, необходимо указать специальный параметр, так как обычно в целях повышения эффективности работы программы механизм RTTI по умолчанию отключен. Так, в компиляторе Microsoft Visual C++ 6.0 таким параметром является /GR.

Динамическое приведение типов с помощью операции динамического приведения типа (`dynamic_cast`) возможно только для объектов родственных полиморфных классов, относящихся к одной иерархии классов. Указатель может иметь нулевое значение, поэтому при динамическом приведении указателя в случае возникновения ошибки может возвращаться это нулевое значение, которое затем может быть проверено в программе. Ошибка при приведении ссылки всегда приводит к возбуждению исключительной ситуации `bad_cast`, так как никакого выделенного значения для ссылок не существует. Проверка правильности динамического приведения ссылок всегда выполняется перехватом исключительной ситуации. `Bad_cast` – класс, описывающий исключительную ситуацию. Так же, как и класс `type_info`, он содержится в библиотечном файле `<typeinfo>`.

Пример:

```
#include <iostream>
#include <typeinfo>
using namespace std;
class A{
public:
    virtual void fx () {
        cout << "A::fx" << '\n';}
};

class B: public A{
public:
    void fx () {
        cout << "B::fx" << '\n';}
};

class C: public A{
public:
    void fx () {
        cout << "C::fx" << '\n'; }
};

void f (A* p, A& r){
    if (B* pp = dynamic_cast<B*> (p)){
        cout << "using of pp" << '\n';
        /* использование указателя pp */
    }
    else {
        cout << "NULL" << '\n';
        /* указатель pp не принадлежит нужному типу */
    }

    B& pr = dynamic_cast<B&> (r);
    /* использование ссылки pr */
}

void g(){
    try {
```

```

    cout << "f (new B, *
    f (new B, *new B);
    cout << "f (new C,
    cout << '\n';
    f (new C, *new C);
    new B) - correct using" << '\n';
// правильный вызов
* new C) - incorrect using";
// выход в перехватчик (C - из
// другой иерархии, основанной
// на том же базовом классе)
}
catch (bad_cast){
    cout << "Bad_cast" << '\n';
    // обработка исключительной ситуации
}
}

int main() {
    g ();
    return 0;
}

```

Однако корректность преобразования, зависит не только от типов указателей и ссылок, но и от типов соответствующих объектов. Пусть наследование классов представлено следующей схемой:

Тогда, если объект создан на основе класса E, то указатель на него, имеющий тип указателя на базовый класс B, может быть преобразован в указатель на базовый класс C. Кроме того, если объект создан на основе класса E, то указатель на него, имеющий тип указателя на базовый класс D, может быть преобразован в указатель на базовый класс B, несмотря на то, что классы B и D не имеют общего базового класса. Это связано с тем, что объект, созданный на основе класса, находящегося на нижней ступени иерархии, содержит в своей структуре структуру всех выше расположенных классов.

Пример:

```

using namespace std;
class A{
public:
    virtual void fx () { cout << "A::fx" << '\n';}
};

class B: public A{
public:
    void fx () { cout << "B::fx" << '\n';}
};

class C: public A{
public:
    void fx () {cout << "C::fx" << '\n';}
};

class D {
public:
    virtual void fd () {
        cout << "D::fd" << '\n';
    }
};

```

```

    }
};

class E: public B, public C, public D {};
void f (C* p, C& r){
    if (B* pp = dynamic_cast<B*> (p)){
        cout << "using of pp in f" << '\n'; }
    else { cout << "NULL in f" << '\n'; }

    B& pr = dynamic_cast<B&> (r);
    /* использование ссылки pr */
}

void f2 (D* p, D& r){
    if (B* pp = dynamic_cast<B*> (p)){
        cout << "using of pp in f2" << '\n'; }
    else {
        cout << "NULL in f2" << '\n'; }
    B& pr = dynamic_cast <B&> (r);
    /* использование ссылки pr */
}

void g(){
    try {
        cout << "f(new E, *new E)" << '\n';
        f (new E, *new E); }
    catch (bad_cast){
        cout << "Bad_cast in f" << '\n';
        // обработка исключительной ситуации
    };
    try {
        cout << " f2 (new E, *new E)" << '\n';
        f2 (new E, *new E);
    }
    catch (bad_cast){
        cout << "Bad_cast in f2" << '\n';
        // обработка исключительной ситуации
    }
};

int main(){
    g();
    return 0;
}

```

В приведенном примере не будет возбуждена ни одна из двух исключительных ситуаций, а указатели будут иметь ненулевые значения. Статическое приведение типов (операция `static_cast`) возможно для объектов родственных классов (полиморфность типов, участвующих в операции, при этом может отсутствовать), относящихся к одной иерархии классов. Статическое приведение возможно также для свободных указателей (`void*`), которые могут преобразовываться в значения любых типов указателей, и для преобразований между арифметическими типами.

Однако при этом невозможно проверить корректность преобразований с использованием исключительной ситуации `bad_cast`. Кроме того, невозможно преобразование

указателя к указателю из другой ветви иерархии наследования, основанной на том же базовом классе, даже если объект создан на основе класса, являющегося наследником классов рассмотренных указателей. Так, если в приведенном выше примере динамическое преобразование позволяет преобразовать указатель типа C^* к указателю B^* , если объект является объектом типа класс E , то статическое преобразование не дает возможности преобразовать указатель типа C^* к указателю B^* .

Примечание. Кроме динамического и статического преобразований имеются и другие преобразования. Например, преобразование `reinterpret_cast` (управляет преобразованиями между произвольными несвязанными типами. Результатом является значение нового типа, состоящее из той же последовательности битов, что и аргумент преобразования). Константное (`const_cast`) преобразование аннулирует действие модификатора `const` (ни статическое, ни динамическое преобразования действие модификатора `const` аннулировать не могут). Подробнее о преобразованиях см. в литературе по $C++$, например, в [12].

Динамическая информация о типе (RTTI)

ООА и ООР

Как составлять диаграммы классов и диаграммы объектов?

UML

Виды диаграмм, применяемых в ООР:

- диаграмма классов [[wiki](#)]
- диаграмма объектов

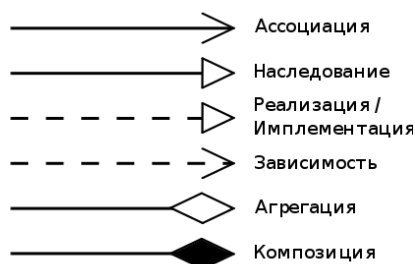


Рисунок 4: UML: Обозначение отношений между классами

Композиция - более строгий вариант агрегации. *Композиция* имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.

Различия между композицией и агрегацией

Приведём наглядный пример. Комната является частью квартиры, следовательно здесь подходит композиция, потому что комната без квартиры существовать не может. А, например, мебель не является неотъемлемой частью квартиры, но в то же время, квартира содержит мебель, поэтому следует использовать агрегацию.

Мощность отношений (Кратность)

Мощность отношения (мультипликатор) означает число связей между каждым экземпляром класса (объектом) в начале линии с экземпляром класса в её конце. Различают следующие типичные случаи:

нотация	объяснение	пример
0..1	Ноль или один экземпляр	кошка имеет или не имеет хозяина
1	Обязательно один экземпляр	у кошки одна мать
0..* или *	Ноль или более экземпляров	у кошки могут быть, а может и не быть котят
1..*	Один или более экземпляров	у кошки есть хотя бы одно место, где она спит

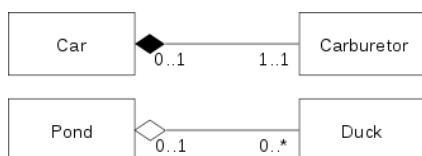


Рисунок 5: Диаграмма классов: Агрегация

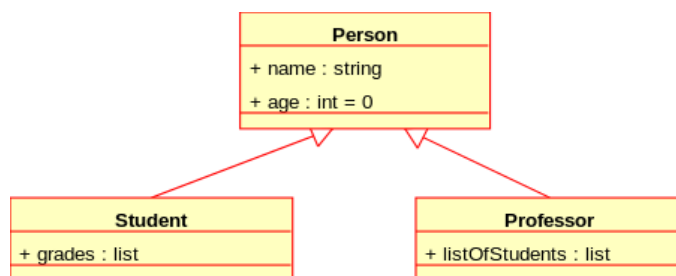


Рисунок 6: Диаграмма классов: Наследование

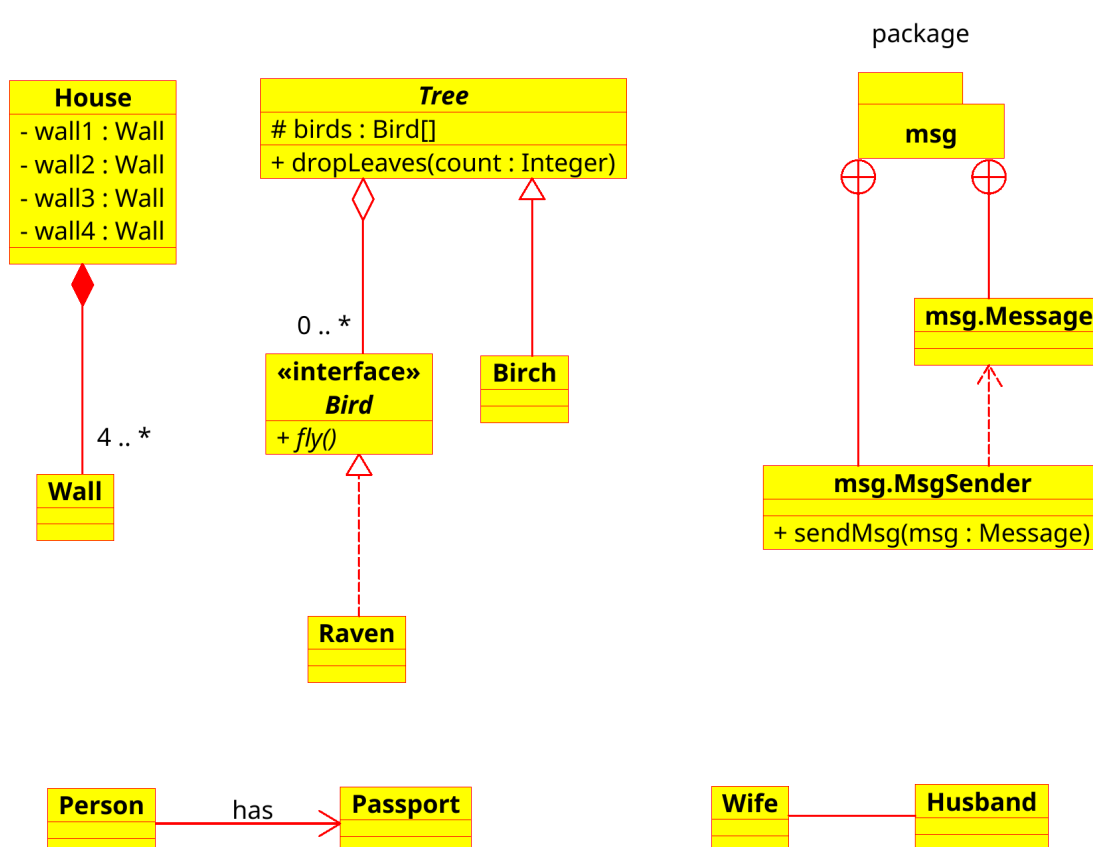


Рисунок 7: Пример диаграммы классов

ПО и сервисы

Какой выбрать - stackoverflow.com

[umbrello](#) Создание UML диаграмм. Генерация кода из UML диаграммы. Агрегация и наследование не генерируются? Плохой интерфейс.

[Modellio](#) Продвинутый инструмент. Относительно сложный интерфейс.

Microsoft visio

Использование стандартных исключений

Qt Quick и QML

Идея создания интерфейса пользователя (UI) как набора классов оправдывает себя с точки зрения программиста, но не с точки зрения дизайнера интерфейсов. Такой подход усложняет проектирование UI, ведь UI — забота дизайнера, а не программиста. К тому же подходу не хватает гибкости, если ставить задачу создания динамичного UI, с нестандартными элементами.

Решение этой проблемы — описание интерфейса на декларативном языке, который потом будет привязан к коду программы. Одна из реализаций данного подхода декларативный язык QML и фреймворк QT Quick.

QML (Qt Meta Language or Qt Modeling Language) — декларативный язык программирования, основанный на JavaScript, предназначенный для дизайна приложений, делающих основной упор на пользовательский интерфейс.

QML элементы могут быть дополнены стандартными JavaScript вставками путём встраивания .js файлов. Также они могут быть расширены C++ компонентами через Qt framework.

Рассмотреть

ОО языки

MDI и SDI

Лабораторные работы

Вместо лабораторных работ (кроме первой) можно разрабатывать собственный проект.

Семестр I

Задание. Диаграмма классов

Составить UML диаграмму классов. Не менее 4 классов. Каждый взаимодействует хотя бы с одним другим. Взаимодействие не должно быть только последовательным. Объекты не должны быть однотипными. Должно быть как минимум по одному отношению: ассоциация, агрегация (композиция), наследование. Указать мощность связей. Диаграмму оформить в электронном и твёрдом формате. Сохранение диаграммы исключительно в формат растровых изображений не допускается. Презентация схемы на доске или проекторе. Обсуждение в группе.

+ использование UML

Необходимо знать: ОО декомпозицию. Виды отношений между классами. UML диаграмму классов.

Задание. Реализовать диаграмму классов на C++

Реализовать предыдущую лабораторную работу. Как минимум одно поле и метод должны быть переопределены. Каждый класс должен располагаться в отдельном модуле.

Представить состояние и изменение объектов наглядно.

Рекомендации: вести разработку «сверху вниз», использовать систему контроля версий.

Необходимо знать: Описание классов в C++. Конструкторы. Методы. Простое наследование. Некоторые stl контейнеры (list, vector, string). Динамический полиморфизм.

Задание. Класс «матрица»

Создать класс представляющий матрицу. В качестве основы использовать класс vector.

Реализовать:

- доступ к отдельным элементам матрицы,
- доступ к строкам матрицы?
- сложение, вычитание
- умножение на число
- умножение матрицы на матрицу
- транспонирование,
- вычисление определителя,
- вычисление обратной матрицы.
- заполнение матрицы одним значением
- заполнение матрицы случайными числами,

- Создание диагональной матрицы.
- Операторы $\ast=$, $-$, $+=$?

Наглядно продемонстрировать работу всех методов. Недопустимые или невозможные операции над матрицами обрабатывать с помощью механизма генерации исключений.

Рекомендуется использование системы контроля версий при разработке.

Задание.

Матричный калькулятор с графическим интерфейсом пользователя?

Диаграмма классов на C++ с графическим интерфейсом пользователя?

Фильтрация ввода некорректных данных. Удобный UI.

Работа с Git.

1. Создать репозиторий.
2. Добавить файлы к отслеживанию.
3. Сделать коммит (зафиксировать изменения).
4. Исправить предыдущий коммит.
5. Создать новую ветку.
6. Переключиться на неё.
7. Внести изменения. Зафиксировать их.
8. Посмотреть что содержится в файлах на ветке master.
9. Объединить ветки.
10. Решить конфликт.
11. Клонировать удалённый репозиторий?
12. Отправить изменения в удалённый репозиторий?

Семестр II

Задание.

Игра «Жизнь» Конвея, Сапёр, Тетрис, свой вариант.

Задание

Эмуляции процессов какой-либо предметной области?

Реализовать АСУТП с эмуляцией устройств (датчиков и т.п.). Состояние датчиков и органов управления представить наглядно.

Задание

???

Погода. Поиск города. Текущая погода и погода на несколько дней. Загрузка погоды из Интернета.

Задание 3. Калькулятор.

Калькулятор должен корректно обрабатывать любые входные данные. Сделать обработку исключительных ситуаций. Хранить историю вычислений. Помимо арифметических операций и возведения в любую степень калькулятор должен вычислять функции: \sin , \cos , \tan , \ln , \exp .

Темы

Задание.

Реализовать класс «умный указатель»

Задание

Посвященное стандартной библиотеке?

Бонус

Хранить лабораторные работы в удалённом репозитории.

Сгенерировать диаграмму классов по одной из лабораторных?

Задание

???

Составить на любом языке программирования консольное приложение, которое содержит описание класса **Time** (время), который должен содержать:

Класс должен включать:

- Закрытые свойства для хранения часов и минут
- Методы доступа к закрытым свойствам
- Конструктор или несколько конструкторов, для создания экземпляров класса
- Метод отображения на экране времени в формате (чч:мм)

Программа должна делать следующее:

1. В функции `main()` нужно объявить и создать массив из 3 объектов описанного класса
2. Задать им следующие значения (2ч 30м, 5ч 15м, 3ч 45м)
3. Вывести на экран время, хранящееся во всех объектах.
4. Рассчитать разницу в днях между 1 и 2 объектами и вывести ее на экран.

Задание

???

Составить на любом языке программирования консольную программу, которая содержит описание класса

Date - дата (год, месяц, день)

Класс должен включать:

- Закрытые свойства для хранения год, месяц, день.
- Методы доступа к зарытым свойствам.
- Конструктор или несколько конструкторов, для создания объектов класса.

- Метод - показать на экране время в формате (дд/мм/гг)
- Метод - рассчитать количество дней с начала года до даты
public int Days()

Программа должна делать следующее:

1. В функции main() нужно объявить и создать массив из 3 объектов описанного класса
2. Задать им следующие значения (1.5.2001 5.2.2002 13.7.2001)
3. Вывести на экран даты, хранящиеся во всех объектах.
4. Рассчитать разницу в днях между 1 и 3 объектами и вывести ее на экран.

Задание.

Игра «Жизнь», Сапёр, Тетрис, свой вариант.

<https://docs.google.com/viewer?>

[a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbmVudC53Vvb3B8Z3g6N2Y1ZjM0YjY0YmJhY2I5NQ](https://docs.google.com/viewer?)

Задание

Piant

Файловая БД библиографии. Оформление библиографических ссылок. Задание шаблона библиографической записи. Поиск по БД. Работа с интернетом — поиск обложек и пр. информации по ISBN. Sqlite?

Рисование графов?

Работа с Интернетом?

Создание БД для поиска файлов?

Файловый менеджер?

Программа для резервного копирования?

Эмуляция?

Эмулятор работы фабрики?

Экзамен 1

Сдаётся в осеннем семестре #3

Экзамен 2

Сдаётся в весеннем семестре #4

Вопросы

<http://www.geeksforgeeks.org/g-fact-13/>

Вопрос

```
#include<iostream>
using namespace std;
class Point {
Point() { cout << "Constructor called"; }
};

int main()
{
Point t1;
return 0;
}
```

Compiler Error

B Runtime Error

Constructor called

By default all members of a class are private. Since no access specifier is there for Point(), it becomes private and it is called outside the class when t1 is constructed in main.

Вопрос

```
#include<iostream>
using namespace std;
class Point {
public:
Point() { cout << "Constructor called"; }
};

int main()
{
Point t1, *t2;
return 0;
}
```

A Compiler Error

Constructor called

B Constructor called

**Constructor
called**

Экзамен

Теоретические вопросы

1. Исключительные ситуации
2. ОО декомпозиция
3. Отношения между классами. Диаграмма классов
4. Алгоритмы стандартной библиотеки
5. Стандартная библиотека
6. Создание и использование объектов класса. Работа с указателем `this`. Создание массивов объектов.
7. Абстрагирование
8. Агрегирование. Композиция
9. Друзья классов
10. Наследование. Виды
11. Приведение типов
12. Виртуальные методы и классы
13. Перегрузка методов, переопределение методов
14. Перегрузка операторов
15. Полиморфизм. Виды. Примеры
16. Динамический полиморфизм
17. Средства преобразования типов. Явные преобразования `static_cast`, `dynamic_cast`, `reinterpret_cast`.
18. Абстрактные классы.
19. Статический полиморфизм
20. Указатели на функции и методы.
21. Лямбда функции.
22. Статические члены класса
23. Шаблоны
24. Паттерны проектирования
25. Liskov Substitution Principle

- 26. SOLID
- 27. Фреймвокс Qt
- 28. QML
- 29. Фреймвокс MFC
- 30. ATL, (WTL)
- 31. WPF
- 32. .NET
- 33. Фреймворк wxWindgets

Литература

1. Г. Буч. Объектно-ориентированный анализ и проектирование с примерами приложений. 720 с. 2010 г. ~700 страниц. Теория. Примеры на C++. Картинки! Вторая половина книги - примеры ООА и ООД с UML диаграммами.
2. Бадд Т. Объектно-ориентированное программирование в действии
3. Б. Страуструп Язык программирования C++. 1136 с. 2015 г. 1000 страниц. Учебник по языку. Шаблоны. ООП. Проектирование.
4. Б. Страуструп Дизайн и эволюция языка C++
5. Скотт Мейерс, Эффективное использование C++. 50 рекомендаций по улучшению ваших программ и проектов
6. Скотт Мейерс, Эффективное использование C++. 35 новых способов улучшить стиль программирования
7. Эффективный и современный C++: 42 рекомендации по использованию C++ 11 и C++14. Скотт Мейерс. 2016. ~300 страниц. Просмотреть. Изучить. Использовать как справочник. Неформальный стиль. Много примеров. Хорошее знание C++.
8. UML???
9. <http://www.cplusplus.com> - Информация по языку и стандартной библиотеке C++
10. www.stackoverflow.com — система вопросов и ответов. На английском языке.

Графические фреймворки

11. <https://wiki.qt.io/Main> Qt wiki
12. Qt 5.X. Профессиональное программирование на C++. Макс Шлее. 2015 г. 928 с. Книга периодически обновляется с выходом новых версий фреймворка Qt.
13. Qt. Профессиональное программирование. Разработка кроссплатформенных приложений на C++. Марк Саммерфилд. Нет новых изданий?

Литература для РП

Основная литература

1. Объектно-ориентированное программирование в C++: лекции и упражнения [Электронный ресурс] : Учебное пособие для вузов / Ашарина И.В. - М. : Горячая линия - Телеком, 2012. - <http://www.studentlibrary.ru/book/ISBN9785991270014.html>
2. Объектно Ориентированное Программирование. Хорошая книга для Хороших Людей [Электронный ресурс] / Комлев Н.Ю. - М. : СОЛОН-ПРЕСС, 2015. - <http://www.studentlibrary.ru/book/ISBN9785913591388.html>

Дополнительная литература

3. Огнева, М. В. Программирование на языке с++: практический курс : учебное пособие для бакалавриата и специалитета / М. В. Огнева, Е. В. Кудрина. — М. : Издательство Юрайт, 2017. — 335 с. — (Серия : Бакалавр и специалист). — ISBN 978-5-534-05123-0. <https://www.biblio-online.ru/book/7670D7EC-AC37-4675-8EAE-DD671BC6D0E4>
4. "Программирование на C++ [Электронный ресурс] / Дейл Н., Уимз Ч., Хедингтон М. Пер. с англ. - М. : ДМК Пресс, 2000. - (Серия "Учебник")" - <http://www.studentlibrary.ru/book/ISBN5937000080.html>

Источники

1. Волкова И. А., Иванов А. В., Карпов Л. Е. Основы объектно-ориентированного программирования. Язык программирования C++. Учебное пособие для студентов 2 курса. – М.: Издательский отдел факультета ВМК МГУ (лицензия ИД No 05899 от 24.09.2001), 2011 – 112 с.
2. Радченко, Г.И. Объектно-ориентированное программирование / Г.И. Радченко, Е.А. Захаров. – Челябинск: Издательский центр ЮурГУ, 2013. – 167 с.

Курсы

[Программирование на языке C++](#)

[Программирование на языке C++ \(продолжение\)](#)

Некоторые статьи

<https://habrahabr.ru/post/87119/>

Рекомендованное ПО и онлайн-сервисы

IDE

- Visual Studio
- QtCreator + Qt
<https://ru.wikipedia.org/wiki/Qt>
<https://www.qt.io/download-qt-for-application-development> OpenSource версия.
+ Gcc (в составе MinGW) <https://sourceforge.net/projects/mingw/>
- Code Blocks
- RAD Studio

[draw.io](#)— создание диаграмм.

Microsoft Visio

[Dia](#) — создание диаграмм. UML, блок-схемы, и т.д.

modelio — продвинутое CASE средства. Относительно сложная программа.

Добавить

Стандартная библиотека std

- шпаргалка

- примеры

Установка IDE и т. д.

Ссылка на оффлан и онлайн версии Qt

<https://www1.qt.io/download-open-source/#section-2>

Если используется онлайн-установщик — не забыть поставить галочку напротив последней версии Qt (выбрать версию Qt только для одного компилятора - MinGW) и в Tools QtCreator и MinGW.

Оффлайн версия для Windows — 2+ Gb