

top

КОМПЬЮТЕРНАЯ
АКАДЕМИЯ

ОБЪЕКТНО -ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ НА C++

Урок 1

Введение
в объектно-
ориентированное
программирование
на C++

Содержание

1. Вступление	5
2. История и этапы развития C++. Сравнение с другими языками программирования	14
3. Три принципа объектно-ориентированного программирования	17
3.1. Инкапсуляция. Определение, примеры использования в повседневной среде.....	21
3.2. Наследование. Определение, примеры использования в повседневной среде.....	24
3.3. Полиморфизм. Определение, примеры использования в повседневной среде.....	26
4. Класс и объект.....	29
5. Классы.....	30
5.1. Понятие класса. Синтаксис объявления	30
6. Переменные-члены класса	35

7. Спецификаторы доступа	38
7.1. Public. Private. Protected.....	38
8. Методы-члены класса	41
8.1. Реализация тела метода внутри класса.....	41
8.2. Вынос тела метода за класс	45
9. Понятие аксессуара, инспектора, модификатора ..	52
9.1. Определение	52
9.2. Реализация	53
10. Встроенные (inline) методы в классах	57
10.1. Необходимость использования. Примеры объявления и использования. Ограничения при использовании inline методов.....	57
11. Сравнительный анализ структур и классов	59
12. Конструктор	61
12.1. Проблемы, возникающие при использовании неинициализированных переменных	61
12.2. Понятие конструктора. Синтаксис объявления	62
12.3. Конструктор, принимающий параметры.....	66
12.4. Конструктор по умолчанию	74
12.5. Перегруженные конструкторы	75
13. Деструктор.....	78
13.1. Утечки ресурсов. Причины их возникновения и плачевные последствия данного явления	78

13.2. Понятие деструктора.	
Синтаксис объявления.....	79
13.3. Примеры использования	80
13.4. Указатели на объекты. Массивы объектов	81
14. Резюме.....	86
15. Терминология.....	91
16. Домашнее задание	92

Материалы урока прикреплены к данному PDF-файлу. Для доступа к ним, урок необходимо открыть в программе Adobe Acrobat Reader.

1. Вступление

Самый лучший способ изучить новый язык программирования — это сразу начать писать на нем программы.

Брайан Керниган, Деннис Ритчи

Во вводном курсе нам часто приходилось решать задачу нахождения среднего арифметического нескольких величин. Давайте вспомним, как это делалось, и напишем программу вычисления среднего балла студента (код программы — в папке *Lesson01\les01_01*). Для хранения необходимых данных создадим структуру **Student**, элементами которой будет ФИО студента и массив его оценок.

```
// Вычислить среднюю оценку студента
#include <iostream>
#include <iomanip>
#include <conio.h>
using namespace std;

// максимальная длина имени
const int maxNameLength = 20;

// число оценок
const int markCount = 3;

// определение структуры данных
// -----
// Студент
struct Student
{
    // ФИО
    char name[maxNameLength];
```

```

        // оценки
        int marks[markCount];
};
// -----
// конец определения структуры данных

// функции обработки структуры данных
// -----
// заполнение данных о студенте
void initStudent(Student& student,
    const char* name,
    const int marks[])
{
    strcpy_s(student.name, maxNameLength, name);
    for (int i = 0; i < markCount; i++)
    {
        student.marks[i] = marks[i];
    }
}

// вычисление средней оценки
double averMark(Student student)
{
    double sum = 0;
    for (int i = 0; i < markCount; i++)
    {
        sum += student.marks[i];
    }
    return sum / markCount;
}

// вывод данных о студенте
void printStudent(Student student)
{
    cout << student.name << endl;
    cout << "Оценки: ";
    for (int i = 0; i < markCount; i++)

```

```

    {
        cout << setw(4) << student.marks[i];
    }
    cout << endl;
}
// -----
// конец списка функций обработки структуры данных

int main()
{
    setlocale(LC_ALL, "");

    cout << "Успеваемость студента."
         << endl << endl;

    Student student;
    const char* studentName{ "Петров А.О." };
    int studentMarks[] { 4,4,3 };

    initStudent(student,
                studentName, studentMarks);
    printStudent(student);
    cout << "Средняя оценка: "
         << fixed << setprecision(2)
         << averMark(student) << endl;

    _getch();
    return 0;
}

```

Давайте при запуске программы настроим экран консоли так, чтобы цвет фона был белым, а цвет текста — черным, тогда изображение снимков экрана и фрагментов снимков экрана будут более понятными. Для этого щелкнем мышью на заголовке окна консоли, выберем «Свойства».

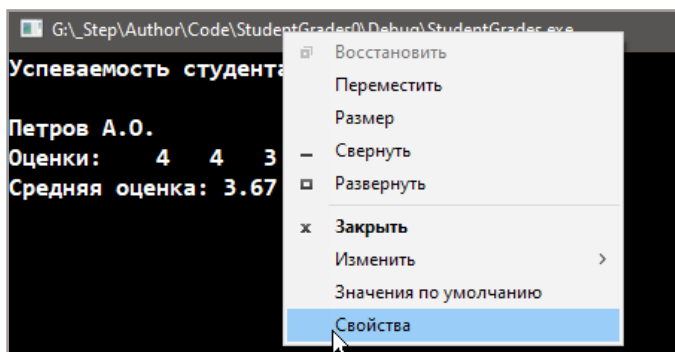


Рисунок 1

В появившемся диалоге установим «Текст на экране» черным, а «Фон экрана» — белым.

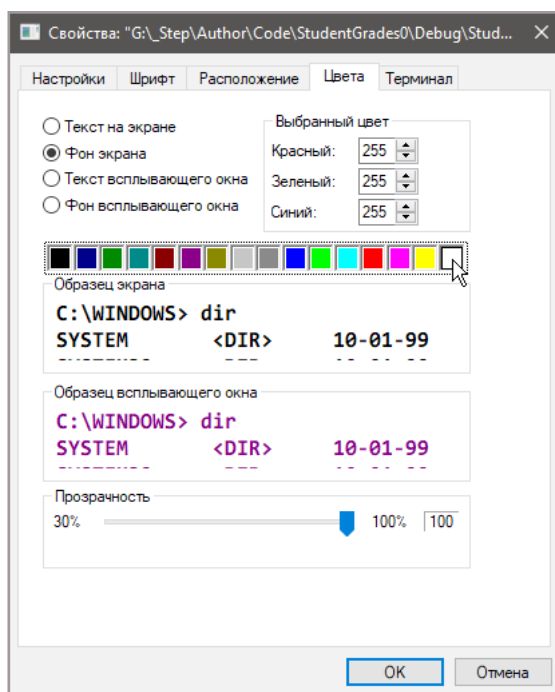


Рисунок 2

При следующем и последующих запусках программы сделанные изменения войдут в силу. Результат работы программы:

```

Успеваемость студента.
Петров А.О.
Оценки:      4      4      3
Средняя оценка: 3.67
  
```

Рисунок 3

Задача решена. Но пришлось написать и использовать несколько функций с длинными именами и длинными списками параметров. А теперь представьте, что потребуется расширить возможности программы. Для этого в структуру нужно будет добавить дополнительные элементы, а список необходимых функций значительно увеличится.

Написание такой программы, а, главное, — отладка ее, поиск и исправление ошибок, — будет трудом нелегким и неблагодарным. Можно ли сделать как-то проще и удобнее? Можно. Существует такой подход к разработке программ — объектно-ориентированное программирование (ООП), при котором значительно улучшается и процесс разработки, и полученный в результате код.

Объектно-ориентированная версия расчета среднего балла будет выглядеть следующим образом (код программы — в папке *Lesson01\les01_02*):

```

// Вычислить среднюю оценку студента
#include <iostream>
#include <iomanip>
#include <conio.h>
  
```

```

using namespace std;

// максимальная длина имени
const int maxNameLength = 20;

// число оценок
const int markCount = 3;

// определение класса
// -----
// Студент
class Student
{
    // ФИО
    char name[maxNameLength];

    // оценки
    int marks[markCount];

public:
    // конструктор: создание объекта Студент
    Student(const char* name, const int *marks)
    {
        strcpy_s(Student::name,
            maxNameLength, name);
        for (int i = 0; i < markCount; i++)
        {
            Student::marks[i] = marks[i];
        }
    }

    // вычисление средней оценки
    double getAver()
    {
        double sum = 0;
        for (int i = 0; i < markCount; i++)
        {

```

```

        sum += marks[i];
    }
    return sum / markCount;
}

// вывод данных о студенте
void print()
{
    cout << name << endl;
    cout << "Оценки: ";
    for (int i = 0; i < markCount; i++)
    {
        cout << setw(4) << marks[i];
    }
    cout << endl;
}

// вывод средней оценки
void printAver()
{
    cout << "Средняя оценка: "
         << fixed << setprecision(2)
         << getAver() << endl;
}
};
// -----
// конец определения класса

int main()
{
    setlocale(LC_ALL, "");

    cout << "Успеваемость студента."
         << endl << endl;

    // вызов конструктора класса:
    // создание объекта

```

```

Student student{ "Петров А.О.",
    new int[3]{ 4,4,3 } };

// вызовы методов класса:
// работа с данными объекта
student.print();
student.printAver();

_getch();
return 0;
}

```

В чем отличие этого варианта (объектно-ориентированного) от предыдущего (структурного)?

Вместо раздельно существующих элементов программы:

- *структура Student,*
- *функции по ее обработке (initStudent, averMark, printStudent)*

у нас теперь есть

- *класс Student,*
который описывает (объединяет в себе) и структуру данных, и функции их обработки. На основе этого класса в программе создается:
- *объект student,*
в котором содержатся данные конкретного студента, и который управляет этими данными с помощью методов класса (функций класса) —
- *student.print();*
- *student.printAver().*

Не правда ли, выглядит лучше? Так что, готовьтесь изучать ООП — долго, внимательно и обязательно с интересом.

Если вы наслаждаетесь используемыми инструментами, то работа будет выполнена успешно.

Дональд Кнут

Но сначала немного отвлечемся... развлечемся... посмотримся, как С++ пришел к ООП, и для чего это было нужно.

2. История и этапы развития C++. Сравнение с другими языками программирования

Если вы считаете, что C++ труден, попытайтесь выучить английский.

Бьерн Страуструп

История C++ насчитывает уже почти полсотни лет. В 1972 году программист из фирмы Bell Labs Деннис Ритчи создал язык программирования C. К этому времени в мире уже существовало немало языков программирования (Fortran, COBOL, Basic, Pascal), но C получился особенным, ни на какой из этих языков не похожим. Дело в том, что другие языки разрабатывались либо учеными — для ученых, либо преподавателями — для студентов.

А язык C — был придуман программистом для программистов. Главные его достоинства — тексты программ получались очень короткими, а программы — очень быстрыми. C стал языком системного программирования — для разработки операционных систем, компиляторов, СУБД.

В 1980 году группой исследователей во главе с Аланом Кэйем был реализован первый по-настоящему успешный объектно-ориентированный язык программирования Smalltalk, и идеи ООП стали завоевывать программистское сообщество. Бьерн Страуструп из той же Bell Labs дополнил C классами и объектами, и в результате получил новый, мощный и эффективный ООП язык, совместимый с C. Этот язык в 1983 получил название C++.

C++ заменил C в качестве языка системного программирования, в частности, операционная система Windows (до Windows XP включительно), писалась именно на нем.

Основные достоинства C++:

- высокая совместимость с языком C, что позволило использовать накопленные за десятилетия наработки на этом языке;
- возможность использования различных подходов и технологий программирования (процедурное программирование, ООП, шаблоны, макросы);
- кроссплатформенность, — достаточно легко переносить программы с Windows на Unix и наоборот;
- эффективность — высокая скорость работы программ, эффективное использование памяти;
- возможность работы на уровне «железа» — с памятью, адресами, портами.

Основные недостатки C++:

- сложность языка. C++ собирался из разных частей и совершенствовался в разных направлениях: чтобы знать его в достаточной степени, надо знать очень много и учиться долго...;
- идеология (достижение максимальной эффективности) и синтаксис языка построены так, что очень легко сделать трудно находимые ошибки.

Преодоление недостатков C++ шло по двум направлениям.

Во-первых, на основе C++ создавались новые языки программирования. Они так и называются — Си-подобные

языки. К ним относятся, например, C#, Java, PHP, perl, Python, Swift и многие другие.

По сравнению с C++ эти языки, как правило:

- более безопасные;
- имеют более узкую область применения;
- легче для изучения.

Вторым направлением были постоянное развитие и стандартизация языка.

Первое описание языка вышло в 1985 году. В 1989 году состоялся выход C++ версии 2.0. Первый стандарт C++ был опубликован в 1998 (известен как C++98). В 2003 был опубликован стандарт языка C++03.

Наиболее серьезные изменения языка произошли в стандарте C++ 11, разработанном в 2011. Далее стандарт обновлялся каждые три года (C++14, C++17), но изменения были не такими существенными.

В феврале 2020 был утвержден стандарт C++20 и намечены направления разработки C++23. Так что язык программирования C++ — это вовсе не набор застывших правил, а живой и развивающийся проект.

3. Три принципа объектно-ориентированного программирования

Сложность программных объектов более зависит от их размеров, чем, возможно, для любых других создаваемых человеком конструкций, поскольку никакие две их части не схожи между собой.

Фредерик Брукс

Простота — замечательное свойство, но необходимы огромные усилия, чтобы ее достичь, и хорошее образование для того, чтобы оценить ее по достоинству.

Эдсгер Дейкстра

Объектно-ориентированный подход в программировании (включает в себя объектно-ориентированный анализ, объектно-ориентированное проектирование и объектно-ориентированное программирование), появился не потому, что кому-то захотелось сделать «красиво», а из суровой жизненной необходимости.

Пока компьютеры были большими и медленными, программисты писали небольшие программы, и все проблемы решались по мере их появления. Когда же компьютеры стали значительно меньше, а скорость их повысилась на порядки, программы сильно выросли в объеме, и в программировании вдруг наступил всемирный кризис, — кризис «первоначального» процедурного программирования.

Процедурное программирование основано на функциях, параметрах, глобальных переменных. В очень большой программе таких элементов становится слишком много, связи между ними оказываются хаотичными, особенно, при исправлении ошибок и добавлении новых возможностей. В результате появляется много новых ошибок, работа замедляется, что, в конце концов, приводит к перерасходу средств и срыву запланированных сроков разработки. В 1994 году исследование, проведенное фирмой IBM, показало, что 68% всех программных проектов отстают от графика, а среднее превышение бюджета разработки достигает 65%.

В не столь далеком 2014 году компания The Standish Group проанализировала более 8 тысяч проектов по разработке и внедрению программных продуктов и показала, что успешными из них были только 16,2%, а провальными (такими, разработку которых прекратили) были 31,1%. Остальные 52,7% проектов были признаны частично успешными, частично неудачными.

Смысл ООП в том, чтобы уменьшить сложность разрабатываемых проектов. Это достигается за счет того, что одно общее большое программное пространство разбивается на много отдельных комнат, внутренние объекты которых связаны только друг с другом (ну и комнаты, конечно, связаны между собой). Комнаты в данной метафоре олицетворяют собой данные, точнее — виды данных, типы данных. С каждым типом данных связывается набор функций, которые эти данные обрабатывают. Вместе — тип данных и связанные функции — представляют отдельную программную конструкцию — класс. Классы

взаимодействуют между собой, данные и функции взаимодействуют внутри класса.

Пусть нам в программе требуется 100 функций. Количество связей между N предметами равно примерно $N * N$, следовательно, общее количество связей программы можно оценить в $100 * 100 = 10000$.

Теперь разбиваем программу на 10 классов, внутри каждого по 10 функций. Общее количество связей теперь $10 * 10$ для каждого из 10 классов плюс связи между классами $= 10 * (10 * 10) + 10 * 10 = 1100$.

Понятно, что расчет очень приблизительный, но все же видно, что даже программа такого небольшого объема становится на порядок (в 10 раз) проще для понимания. А современные программы представляют собой комплексы из миллионов строк кода, и для них этот эффект возрастает многократно...

Еще раз сформулируем разницу между процедурным программированием (язык C) и объектно-ориентированным программированием (язык C++).

При процедурном программировании структура программы определяется набором функций для обработки данных. Сами данные при этом остаются на втором плане и определяются по ходу разработки функций.

В объектно-ориентированном программировании основную роль играют данные. Структура программы соответствует данным, которые она обрабатывает: сначала определяются виды данных и связи между ними, а уже потом — методы обработки (функции).

Нужно хорошо понимать, что программа — это модель, и разница в подходах — это разница в том, как мы

модель строим, с какой начальной точки начинаем, и как потом модель детализируем.

Например, пусть нам нужно построить программу «Филиал академии Шаг». При процедурном подходе мы начинаем со списка задач:

- вести списки набора студентов;
- составлять расписание;
- вести списки преподавателей;
- следить за успеваемостью;
- следить за оплатой занятий;
- рассчитывать зарплату.

И уже потом определяем необходимые данные:

- составить расписание (занятия);
- следить за успеваемостью(студенты);
- следить за оплатой(студенты).

При объектно-ориентированном подходе мы начинаем с перечня объектов, данные о которых мы должны обрабатывать:

- кандидаты в студенты;
- студенты;
- преподаватели;
- занятия;
- бюджет филиала.

И уже потом определяем необходимые методы:

- Занятия.СоставитьРасписание;
- Студенты.СледитьЗаУспеваемостью;
- Студенты.СледитьЗаОплатой.

При втором подходе у нас значительно меньше элементов первого уровня, они лучше определены, и от них проще создавать и изменять элементы второго уровня.

Для того, чтобы писать программы подобным образом, в языке должны существовать специальные механизмы. К основополагающим относятся так называемые принципы ООП: инкапсуляция, наследование и полиморфизм.

3.1. Инкапсуляция. Определение, примеры использования в повседневной среде

Omnia mea mecum porto (всё своё ношу с собой).

Цицерон

Почаще задавайте себе вопрос «Что мне скрыть?» и вы удивитесь, сколько проблем проектирования растает на ваших глазах.

Стивен Макконнелл

Инкапсуляция (с латинского «в ящике») — очень простой и самый главный принцип ООП.

Инкапсуляция — это свойство системы, позволяющее тесно связать данные и методы работы с ними внутри класса, и сделать данные и детали реализации недоступными для других частей системы.

Представьте себе — телефон из прошлого века. Для набора номера нужно было вращать диск... Нет, представьте себе телефон начала прошлого века: «Барышня, соедините меня с ...». И барышня на телефонной станции должна была отозваться, принять заказ, найти нужные гнезда в коммутаторе и соединить шнуры.



Рисунок 4

Наш мобильник делает то же самое, — запрашивает соединение, ищет адресата, соединяет двух абонентов, но только все детали реализации этого процесса скрыты от нас в изящной коробочке.

Инкапсулированы

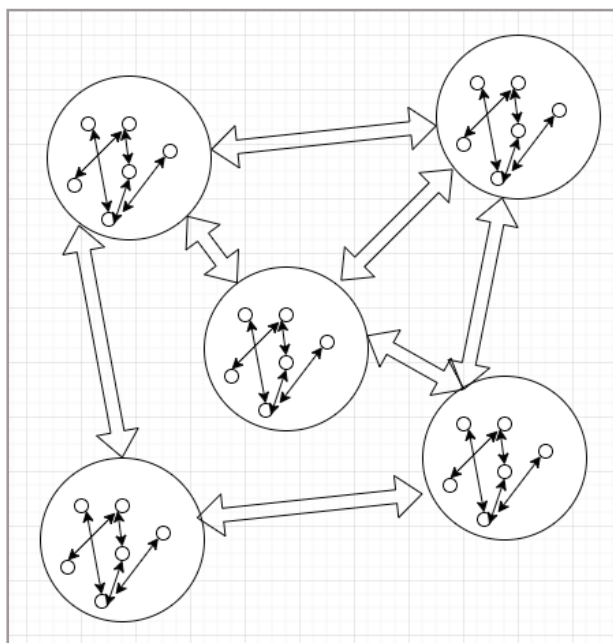


Рисунок 5

В окружающем нас мире самый впечатляющий пример инкапсуляции — клетки живых организмов (рис. 5). Они взаимодействуют как цельные и неделимые сущности, в то время, как внутри них имеется сложная и активная структура.

В приложении к программированию принцип инкапсуляции обозначает следующее:

- программа собирается из отдельных модулей (деталей и узлов деталей);
- для сборки и дальнейшей работы — достаточно знать о каждом модуле, — для чего он предназначен и как он подсоединяется. Из чего он состоит и каким образом работает — знать не нужно;
- модуль — это органичное, естественное соединение внутренних элементов — данных и способов их работы — функций. (Функции в ООП немного отличаются от обычных функций и называются в C++ — функциями-членами класса, а во многих других объектно-ориентированных языках — методами класса);
- модули изготавливаются по чертежам. Чертеж содержит всю детальную информацию, необходимую для использования и усовершенствования модуля.

Чертежи — это классы, из которых состоит текст программы. **Модули** — это объекты, которые создаются на основе классов во время работы программы. Классы описывают состав данных объектов, и методы объектов, работающие с этими данными. Объекты содержат данные и запускают на выполнение функции для обработки этих данных. На основе одного класса может быть создано

множество объектов, каждый из которых содержит свои собственные, уникальные данные.

Самое главное в инкапсуляции — объект изменяет данные другого объекта, используя только функции этого другого объекта, ничего не зная о структуре данных другого объекта. Это делает классы в высокой степени независимыми друг от друга, что позволяет легко модифицировать и развивать тексты программ. Единственное, что должно оставаться неизменным — заголовки функций класса, которые вызываются другими объектами.

3.2. Наследование. Определение, примеры использования в повседневной среде

*Если я видел дальше других, то потому,
что стоял на плечах гигантов.*

Исаак Ньютон

Большинство идей основываются на предыдущих идеях.

Алан Кей

Наследование — это отношение между классами, при котором один класс повторяет структуру и поведение другого класса (одиночное наследование) или других классов (множественное наследование).

Класс, который наследуется, называется суперклассом, надклассом, базовым или родительским классом.

Класс, производный от суперкласса, называется подклассом, производным или дочерним классом

Основной смысл наследования — повторное использование чего-то, с чем уже работали ранее, возможно,

с некоторой доработкой. Подходящим примером из жизни является подготовка документа по шаблону.

Пусть вы собираетесь устроиться на новую работу. Первым делом нужно разослать по интересующим вас адресам свое резюме.

Разумеется, вы не будете писать резюме с нуля, а возьмете существующий текст и на его основе подготовите несколько вариантов, соответствующих требованиям работодателей. И даже если у вас в архиве нет резюме, вы будете создавать его не с чистого листа, а на основании шаблона, взятого в интернете.

Шаблон или существующий текст — базовый класс, ваши резюме — дочерние классы.

Точно так же, автомобильная фирма при создании нового поколения автомобилей возьмет за основу существующую успешную модель, внесет изменения в конструкцию и выпустит новый модельный ряд. Все полученные модификации будут иметь большинство свойств прежней модели, при этом у каждой будет свои новые особенности.

В программировании ценность наследования увеличивается еще тем, что оно позволяет использовать и изменять поведение объектов класса даже без доступа к исходным текстам класса.

Пусть у нас есть в библиотеке (без исходного текста) класс `A`. C++ позволяет нам создать исходный текст класса `B` и оформить его, как наследника класса `A`. Наследование обозначает, что объекты класса `B` полностью обладают теми же характеристиками, что и объекты класса `A`. Теперь в исходном тексте класса `B` можно изменить поведение

объектов класса **B** — добавить какие-то методы, удалить, переписать. И в итоге мы имеем все нужные нам возможности класса **A** плюс новые возможности из класса **B**.

В основном, наследование используется для того, чтобы не повторять один и тот же код в разных классах. Представьте, что мы пишем программу, которая моделирует поведение разных видов транспорта: автомобиля, мотоцикла, самолета. У них всех есть общие свойства: скорость, вес, цвет и общие методы: заправиться, двигаться. Правильным подходом к разработке будет создание базового (родительского) класса **Транспорт**, и классов — наследников: **Мотоцикл**, **Автомобиль**, **Самолет**. В базовом классе будет сосредоточен код, общий для всех, а в классах-наследниках — только специализированный, уникальный для каждого класса код.

3.3. Полиморфизм. Определение, примеры использования в повседневной среде

*Долой канотье, вместо тросточки — стек,—
И шепчутся дамы:
Да это же просто другой человек!
А я — тот же самый.*

В.С. Высоцкий

Полиморфизм (с греческого — многообразность) — это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач.

Полиморфизм в программировании — это возможность применения одноименных методов с одинаковыми или различными наборами параметров в одном

классе или в группе классов, связанных отношением наследования.

В переводе на обыденный язык «полиморфизм» — это понятия, которые обозначают действия, похожие по назначению, но существенно различающиеся по исполнению, например, «пообщаться», «открыть ключом».

Мы можем пообщаться по телефону, по смартфону, пообщаться по скайпу, пообщаться через teams, другие мессенджеры, наконец, просто пообщаться без всяких технических средств.

Аналогично, — можно открыть ключом дверь в квартиру, открыть электронным ключом замок домофона, замок автомобиля, открыть ключом — паролем зашифрованный документ.

Смысл полиморфизма — экономия понятий. Не нужно придумывать и запоминать новые слова, не будет ошибок непонимания.

Полиморфизм в программировании — это наличие разных функций с одинаковыми именами. Чаще всего — это функции с одинаковыми именами у разных классов, которые все являются наследниками одного базового класса.

Если взять предыдущий транспортный пример, то при вызове метода Загрузить класса Транспорт:

- для объекта Мотоцикл выполнится посадка мотоциклиста;
- для объекта Автомобиль выполнится посадка водителя и нескольких пассажиров;
- для объекта Самолет выполнится посадка команды и пассажиров.

Мы показали основные принципы ООП, то есть правила и приемы построения объектно-ориентированной программы. Понятно, что для реализации этих принципов в языке С++ есть соответствующие синтаксические конструкции. Рассмотрим их.

4. Класс и объект

Мой опыт в математике заставил меня понять, что каждый объект может иметь несколько алгебр, они могут объединяться в семейства, и это может быть очень полезным.

Алан Кей

Программы пишутся для обработки данных. Данные в тексте программы представляются переменными. В ООП переменные — это не только простые (встроенные `int`, `char`) типы данных, но и объекты сложной структуры.

Для создания объекта необходим шаблон объекта, (для построения башенок из мокрого песка нужно ведро, не правда ли?).

Таким шаблоном, инструментом для создания объектов служит класс.

Класс является инструментальным описанием структуры и поведения однотипных объектов, а объект — это отдельный представитель (экземпляр) класса, имеющий свое собственное индивидуальное состояние.

Новые концепции легче понимать, если рассматривать их на конкретных примерах, поэтому давайте сделаем очень постепенное построение уже приведенной программы «Средний балл».

5.1. Понятие класса. Синтаксис объявления

Лично я лучше всего учусь на примерах. Думаю, это относится и к другим программистам.

Стивен Макконнелл

Итак, открываем Visual Studio (примеры в этом руководстве разработаны в среде Visual Studio 2019 Community), выбираем «[Create a new project](#)», в следующем окне устанавливаем тип проекта.

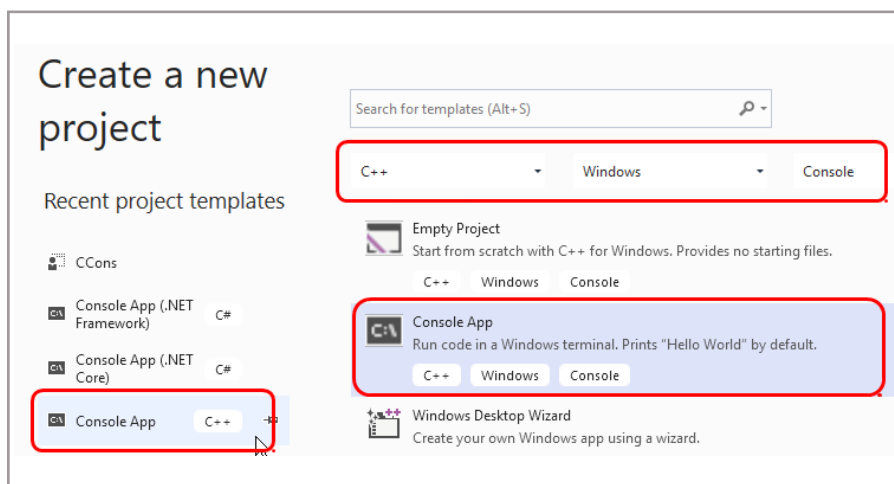


Рисунок 6

Жмем кнопку «[Next](#)». В открывшемся окне задаем имя приложения «[StudentGrades](#)», задаем расположение проекта, отмечаем «[Place solution and project in the same directory](#)».

Configure your new project

Console App C++ Windows Console

Project name
StudentGrades

Location
G:_Step\Author\Code\

Solution name ⓘ
StudentGrades

☒ Place solution and project in the same directory

Рисунок 7

Жмем «Create». Получаем текст приложения «*Hello World!*». Убираем лишнее, добавляем нужное, выводим заголовок (код программы — в папке *Lesson01\les01_03*).

```
// Вычислить среднюю оценку студента
#include <iostream>
#include <conio.h>
using namespace std;

int main()
{
    setlocale(LC_ALL, "");
    cout << "Успеваемость студента."
          << endl << endl;
    _getch();
    return 0;
}
```

Чтобы создавать объекты, сначала должен быть создан класс. Класс состоит из заголовка класса и тела класса.

```
class имя_класса
{
    список_элементов_класса
};
```

где:

- **class** — ключевое слово, по которому компилятор понимает, что дальше должно быть размещено описание класса;
- **имя_класса** — идентификатор, составленный по правилам C++. Имя каждого класса в программе должно быть уникальным;
- фигурные скобки заключают в себе тело класса, которое содержит описание данных класса и методы класса;
- точка с запятой после закрывающей скобки обязательна.

Объекты в нашей программе будут создаваться в функции **main**, в этой точке компилятору уже необходимо знать, что из себя представляет класс, поэтому текст класса должен быть расположен перед **main**.

Определение объекта

```
имя_класса имя_переменной;
```

где:

- **имя_класса** — имя класса, на основе которого создан объект (переменная);
- **имя_переменной** — просто имя переменной.

Создаем следующую версию программы (код программы — в папке *Lesson01\les01_04*)

```
// Вычислить среднюю оценку студента

#include <iostream>
#include <conio.h>
using namespace std;

// определение класса
class Student
{
};

int main()
{
    setlocale(LC_ALL, "");
    cout << "Успеваемость студента."
         << endl << endl;

    // определение объекта
    Student student;
    _getch();
    return 0;
}
```

Строки:

```
class Student
{
};
```

представляют собой определение класса, другими словами — описание объектов, которые могут быть созданы на основе этого класса.

В строке

```
...Student student;
```

определение объекта, другими словами — определение переменной, которая представляет конкретный объект, созданный на основе класса.

Класс — один. Класс — это описание, набор правил, схема.

Объектов — может быть много. Объекты — это переменные, которые хранят конкретные данные. Каждый объект хранит свои индивидуальные данные.

Мы видим, что определение (создание) объекта в точности соответствует определению (созданию) переменной встроенного типа данных (`int`, `double`, `char`). Из этого следует, что определение класса равносильно определению нового типа данных.

C++ — расширяемый язык программирования. Это означает, что в нем можно создавать новые типы данных, — просто создавая свои классы.

В приведенном выше примере:

- `class Student` — заголовок класса;
- `Student` — имя класса;
- `{ }` (пустые фигурные скобки) — тело класса (пока без элементов класса);
- `Student student;` — создание объекта класса `Student`, определение переменной `student` типа `Student`.

6. Переменные-члены класса

Лучше, чтобы в 100 функциях использовалась одна структура данных, чем в 10 функциях — 10 структур.

Алан Перлис

Мы научились создавать классы и объекты, вот только наши объекты пока ничего не умеют делать и даже ничего не хранят.

Для того, чтобы в объекте хранились значения, нужно в классе определить соответствующие поля объекта посредством объявления переменных-членов класса (код программы — в папке *Lesson01\les01_05*).

```
// Вычислить среднюю оценку студента

#include <iostream>
#include <conio.h>
using namespace std;

// определение класса Студент
class Student
{
    public:
        // ФИО
        char name[21];
        // оценки
        int marks[3];
};

int main()
{
    setlocale(LC_ALL, "");
```

```

cout << "Успеваемость студента."
      << endl << endl;

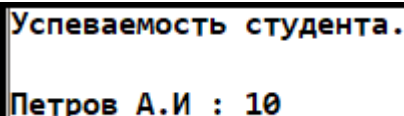
// определение объекта
Student student;

// присвоение значений объекту
strcpy_s(student.name, 20, "Петров А.И");
student.marks[0] = 10;

// получение значений объекта
cout << student.name << " : "
      << student.marks[0] << endl;
_getch();
return 0;
}

```

Результаты работы программы:



```

Успеваемость студента.

Петров А.И : 10

```

Рисунок 8

Добавленные в теле класса строки

```

char name[21];
int marks[3];

```

приводят к тому, что объекты класса `Student` (каждая переменная типа `Student`) уже не будут пустыми, — каждый такой объект будет содержать в себе массив символов длиной 21 и массив целых длиной 3.

Переменные, объявленные в теле класса, в оригинале документации по C++ называются **data members**. Чаще всего это переводится как переменные-члены класса. Другие переводы: данные-члены класса, элементы данных, атрибуты класса, поля класса.

Мы в дальнейшем будем использовать термин переменные-члены класса, поскольку именно такое название чаще всего используется в литературе по C++, а в применении к объектам — термин поля объекта

После того, как переменная-член класса объявлена в классе, она входит в состав каждого объекта класса, и доступ к ней осуществляется посредством оператора «.» (точка — выбор элемента) через конструкцию **имя_объекта.имя_поля_класса**, например,

```
student.name,
```

где:

- **student** — имя переменной типа **Student** (объекта класса **Student**);
- **.** — оператор-точка выбор элемента;
- **name** — имя поля класса **Student**.

Как мы видим в строках

```
strcpy_s(student.name, 20, "Петров А.И");
student.marks[0] = 10;
```

доступ к каждой переменной-члену класса осуществляется индивидуально, причем можно, как получать значения переменных, так и присваивать новые значения этим переменным.

7. Спецификаторы доступа

7.1. Public. Private. Protected

*«Добро пожаловать, или
Посторонним вход воспрещён»
Название фильма.*

Обратим внимание на строку

```
public:
```

Это модификатор доступа (другое имя — спецификатор доступа). Модификаторы доступа управляют доступом к членам класса. В C++ есть три уровня доступа:

- **public** делает члены класса открытыми. Это значит, что доступ к членам класса возможен из любой части программы;
- **private** делает члены класса закрытыми, то есть доступ к ним возможен только внутри класса, для других частей программы эти члены недоступны.
- **protected** открывает доступ к членам класса только для дочерних классов (и дружественных функций, но об этом потом). Такие члены класса называют защищенными.

Рассмотрим следующий пример:

```
class AccessLevels  
{  
    int privateByDefault;  
public:  
    int publicMember;
```

```
protected:
    int protectedMember;

private:
    int privateMember;
};
```

Согласно принципам инкапсуляции, все, что находится внутри класса, — по умолчанию недоступно за его пределами. Поэтому переменная — член класса `privateBy-Default`, перед которой не указан никакой модификатор доступа является закрытым членом класса.

Далее модификатор `public`: действует на следующие за ним строки, поэтому `publicMember` является открытым членом класса.

`Protected`: отменяет действие предыдущего модификатора и делает `protectedMember` защищенным.

`Private`: отменяет действие предыдущего модификатора и делает `privateMember` закрытым.

Модификаторы могут ставиться в любом порядке и повторяться несколько раз, каждый модификатор действует на все следующие за ним члены класса (до следующего модификатора). По умолчанию (если не объявлен никакой модификатор доступа) члены класса являются закрытыми.

Если мы прокомментируем строку `public`: в определении класса и попытаемся скомпилировать программу, то получим следующие сообщения (рис. 9).

Поскольку модификатора `public` перед переменными-членами класса больше нет, все переменные стали

закрытымі. Компілятар сообщае, што нельзя абрацца к закрытым пераменным-членам класа, паэтому праграма не кампілюецца. Убрав камментарыі з `public`, вернем праграму ў рабочае становішча.









	E0265	member "Student::name" (declared at line 12) is inaccessible
	E0265	member "Student::name" (declared at line 12) is inaccessible
	E0265	member "Student::marks" (declared at line 14) is inaccessible
	E0265	member "Student::marks" (declared at line 14) is inaccessible
	C2248	'Student::name': cannot access private member declared in class 'Student'
	C2248	'Student::name': cannot access private member declared in class 'Student'
	C2248	'Student::marks': cannot access private member declared in class 'Student'
	C2248	'Student::marks': cannot access private member declared in class 'Student'

Рисунок 9

8. Методы-члены класса

8.1. Реализация тела метода внутри класса

Наша цель в том, чтобы каждый метод эффективно решал одну задачу и больше ничего не делал.

Стивен Макконнелл

Мы знаем, что поля объектов класса определяются в теле класса (в нашем примере — переменные-члены класса: `name`, `marks` — внутри класса `Student`). Таким же образом определяются и функции класса, работающие с этими полями.

Добавим в класс `Student` функцию `getAver`, которая будет считать средний балл студента на основе его оценок, хранящихся в полях класса (код программы — в папке `Lesson01\les01_06`).

```
// определение класса Студент
class Student
{
public:
    // ФИО
    char name[21];
    // оценки
    int marks[3];
    // вычисление среднего балла
    double getAver()
    {
        double sum = 0;
        for (int i = 0; i < 3; i++)
        {
```

```

        sum += marks[i];
    }
    return sum / 3;
}
};

```

Функции, принадлежащие классу, в оригинальной документации по C++ называются **member function**. Переводится как функции-члены класса, элемент-функции, методы (класса).

Мы в дальнейшем будем использовать термины функции-члены класса (основной термин C++) и методы (общий термин для многих языков ООП).

Определение функции-члена класса помещается в любом месте в теле класса (нельзя, конечно, размещать одну функцию внутри другой).

Определение функции-члена класса синтаксически ничем не отличается от определения обычной функции C++: заголовок функции, тело функции, тип возвращаемого значения, параметры...

Но есть одно важное отличие: в коде функции-члена класса можно обращаться не только к локальным переменным и параметрам функции, но и к переменным-членам класса.

```

// marks — переменная-член класса
sum += marks[i];

```

Мы знаем, что физически (в памяти компьютера) переменных-членов класса не существует, — это просто описание структуры полей объекта класса. Таким

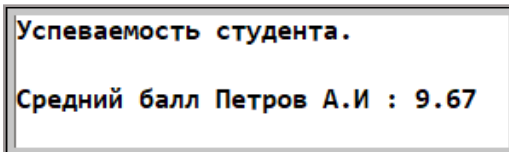
образом, обращение к переменной-члену класса в функции-члене класса представляет собой обращение к соответствующему полю соответствующего объекта.

Рассмотрим использование функции `getAver` (код программы — в папке *Lesson01\les01_06*).

```
int main()
{
    setlocale(LC_ALL, "");
    cout << "Успеваемость студента."
         << endl << endl;
    // определение объекта
    Student student;
    // присвоение значений объекту
    strcpy_s(student.name, 20, "Петров А.И");
    student.marks[0] = 10;
    student.marks[1] = 10;
    student.marks[2] = 9;
    // выполнение метода объекта
    double aver = student.getAver();

    cout << "Средний балл " << student.name
         << " : " << fixed << setprecision(2)
         << aver << endl;
    _getch();
    return 0;
}
```

Результаты работы программы:



```
Успеваемость студента.
Средний балл Петров А.И : 9.67
```

Рисунок 10

Вызов функции-члена класса отличается от вызова обычной функции. Синтаксис вызова функции-члена: `имя_объекта.имя_функции(список_параметров)`. Например:

```
double aver = student.getAver();
```

где:

- `student` — имя переменной типа `Student` (объекта класса `Student`);
- `.` — оператор «точка»;
- `getAver` — имя метода класса `Student`.

Об обычной функции нельзя сказать, что ее кто-то вызывает, — она просто анонимно «вызывается». Функцию-член класса всегда вызывает существующий объект класса. И члены-переменные класса, к которым обращается функция на самом деле обозначают поля того объекта, который вызывает функцию.

Вызов метода `student.getAver` считает среднее значение полей `student.marks` (полей `marks` объекта `student` — в нашем примере им присвоены значения 10, 10 и 9).

Таким образом, мы можем (упрощенно) представить себе каждый объект класса как отдельную маленькую программу, которая хранит свои собственные значения полей класса и свои методы класса, которые работают именно с этими значениями.

Функции-члены класса, как и обычные функции, могут быть с параметрами, без параметров, перегруженные, шаблонные...

Для функции-члена класса переменные-члены класса являются примерно тем же, чем для обычных функций

являются глобальные переменные. Однако, надо понимать, что переменные класса — это НЕ глобальные переменные. Глобальная переменная существует в одном экземпляре на всю программу, переменная-член класса — в одном экземпляре на каждый объект класса в программе. К глобальной переменной можно обратиться из любой функции программы, к переменной класса (без указания имени объекта) — только из функций-членов класса.

Поскольку методы класса могут работать непосредственно с полями класса, у методов класса значительно сокращается число параметров по сравнению с обычными функциями.

Вызов метода класса можно еще представить себе, как вызов обычной функции, у которой есть скрытый параметр — ссылка на объект, вызывающий метод.

Обычные функции C++ — глобальны, то есть они все существуют в одном общем пространстве и поэтому не может быть двух одинаковых (с учетом правил перегрузки) функций. Функции-члены каждого класса находятся в отдельном пространстве своего класса, поэтому совершенно одинаковые по названию и параметрам функции-члены разных классов допустимы, они должны быть уникальными только в пределах класса.

8.2. Вынос тела метода за класс

Снимите шляпы, обнажите головы. Сейчас состоится вынос тела.

И. Ильф, Е. Петров

Мы видим, что класс C++, так же, как и функция C++ является отдельной независимой программной единицей

и, в принципе, может быть использован в нескольких разных программах.

Для этого, как и в случае с функцией, должна быть возможность реализации класса в отдельном файле. И так же, как с функцией, нужен отдельный заголовочный файл, включающий в себя необходимую для компиляции информацию о классе. Что должно входить в этот заголовочный файл? — То, что должен проверить компилятор: имена полей класса и прототипы методов класса.

Перенесем код класса в отдельный файл за два шага. На первом шаге вынесем код функции-члена класса `getAver` из описания класса и поместим его отдельно в том же файле (код программы — в папке *Lesson01\les01_07*).

В теле класса должен остаться прототип функции.

```
// определение класса Студент
class Student
{
public:
    // ФИО
    char name[21];

    // оценки
    int marks[3];

    // вычисление среднего балла
    double getAver();
};
```

Определение класса `Student` (с прототипом `getAver`) расположено перед функцией `main`, следовательно,

реализацию функции можно поместить как перед, так и после `main`.

Разместим код функции `getAver` после кода `main`:

```
// реализация метода вычисления среднего балла
double Student::getAver()
{
    double sum = 0;
    for (int i = 0; i < 3; i++)
    {
        sum += marks[i];
    }
    return sum / 3;
}
```

Обратите внимание, что перед именем функции-члена класса `getAver` в заголовке функции записано имя класса `Student` и операция разрешения области действия «`::`» (двойное двоеточие). Таким образом функция-член класса `getAver` «присоединяется» к отделенному от нее определению класса `Student`. Без этого префикса «`Student::`» перед именем функция была бы распознана компилятором просто как обычная функция, не имеющая никакого отношения к классу.

На втором шаге поместим в нашем проекте класс `Student` в отдельном файле исходного кода.

Сначала удалим (или закомментируем) код класса `Student`. Если этого не сделать, то нельзя будет добавить к проекту еще один класс с таким же именем `Student`.

Щелкаем на имени проекта в браузере решений, выбираем команду «`Add`» > «`New Item...`».

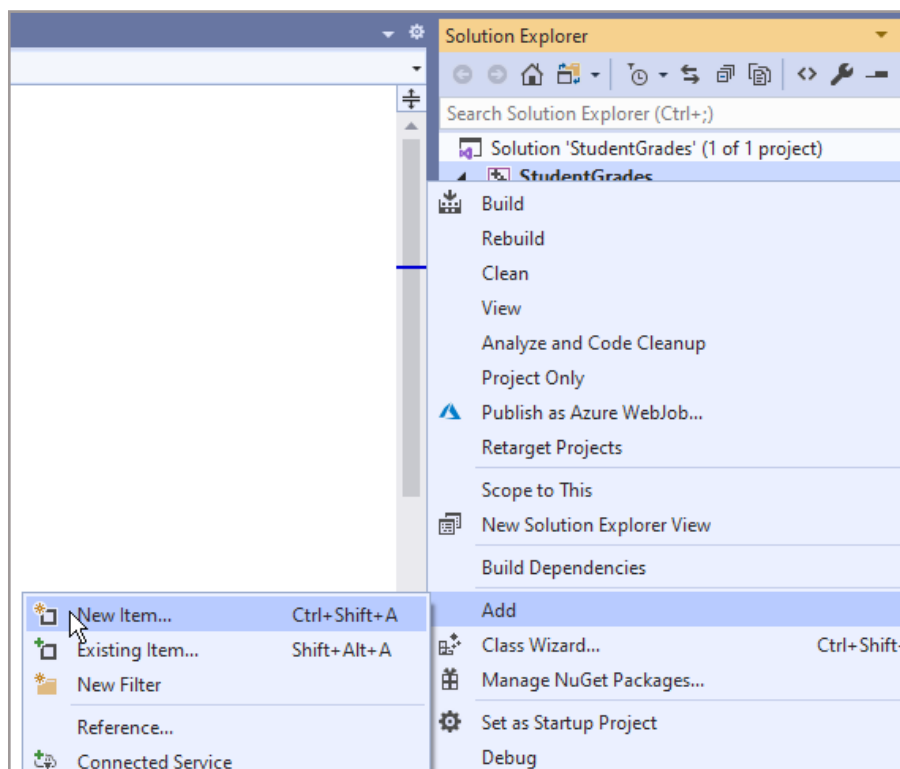


Рисунок 11

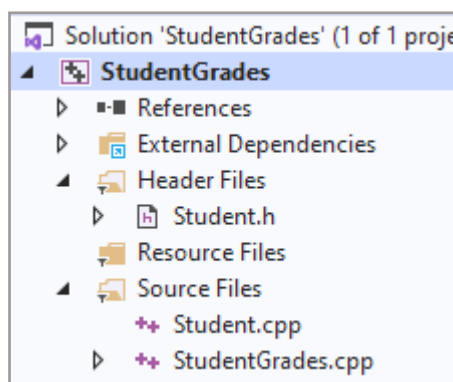


Рисунок 12

В появившемся диалоге выбираем «C++ Class» и вводим в поле **Name** название класса «Student». Нажимаем «Add».

В появившемся диалоге «Add Class» просто нажимаем «Ok». В проекте появляются файлы **Student.h** и **Student.cpp**. В файл **Student.h** переносим определение класса **Student** (код программы — в папке *Lesson01\les01_08*).

Файл Student.h

```
#pragma once
class Student
{
    public:
        // ФИО
        char name[21];
        // оценки
        int marks[3];
        // вычисление среднего балла
        double getAver();
};
```

В файл **Student.cpp** переносим определение метода **getAver**.

Файл Student.cpp

```
#include "Student.h"
using namespace std;
// реализация метода вычисления среднего балла
double Student::getAver()
{
    double sum = 0;
    for (int i = 0; i < 3; i++)
    {
```

```

        sum += marks[i];
    }
    return sum / 3;
}

```

В файл добавлен заголовок класса `Student.h`.

Этот же заголовок добавляем и в файл `StudentGrades.cpp`, функция `main` которого использует класс `Student`.

Файл `StudentGrades.cpp`

```

// Вычислить среднюю оценку студента

#include <iostream>
#include <iomanip>
#include <conio.h>
#include "Student.h"
using namespace std;

int main()
{
    setlocale(LC_ALL, "");
    cout << "Успеваемость студента."
          << endl << endl;

    // определение объекта
    Student student;

    // присвоение значений объекту
    strcpy_s(student.name, 20, "Петров А.И");
    student.marks[0] = 10;
    student.marks[1] = 10;
    student.marks[2] = 9;

    // выполнение метода объекта
    double aver = student.getAver();
}

```

```
cout << "Средний балл " << student.name  
    << " : " << fixed << setprecision(2)  
    << aver << endl;  
  
_getch();  
return 0;  
}
```

Больше никаких изменений в основную программу вносить не нужно.

Программа компилируется, запускается и выдает правильный результат.

9. Понятие аксессора, инспектора, модификатора

Подальше положишь — поближе возьмешь.

Пословица

9.1. Определение

В нашей объектно-ориентированной программе есть один серьезный объектно-ориентированный недочет: класс `Student` полностью открыт для любых частей программы, в которой он используется, что противоречит принципу инкапсуляции. Для того, чтобы убрать этот недостаток, нужно сделать следующее:

- закрыть переменные-члены класса;
- обеспечить возможность получения значений этих переменных (доступ для чтения);
- обеспечить возможность задания значений этих переменных (доступ для записи).

Для решения первой проблемы нужно поставить модификатор доступа `private` перед переменными-членами класса. Для доступа к переменным нужно добавить к классу соответствующие функции-члены, через вызов которых можно будет читать и записывать значения полей объектов.

В классах часто предусматриваются открытые функции-члены, предназначенные для установки значения (`set`) или получения значения (`get`) закрытых переменных-членов. Имена этих функций могут быть какими угод-

но, но обычно их начинают, соответственно, с `set` или `get`: `setName`, `getMarks`.

`Set`-функции принято называть мутаторами (**mutators**) или модификаторами, поскольку они изменяют значения полей объекта.

`Get`-функции, соответственно принято называть аксессуарами (**accessors**) или инспекторами (**inspectors**), потому что они осуществляют доступ (**access**) к значениям, просмотр значений.

Использование аксессуаров и мутаторов дает следующие преимущества:

- данные защищаются от неправильного использования;
- проверка корректности данных концентрируется в одном месте (в мутаторе);
- изменение структуры данных класса не влечет за собой изменение других частей программы;
- программу легче отлаживать (задание точки останова в мутаторе позволит отследить все изменения переменной-члена класса).

9.2. Реализация

Вносим необходимые изменения (код программы — в папке *Lesson01\les01_09*). Заголовок класса **Student**:

Файл Student.h

```
class Student
{
private:
    // ФИО
    char name[21];
```

```

    // оценки
    int marks[3];
public:
    // вычисление среднего балла
    double getAver();

    // доступ к полю name
    const char* getName();
    void setName(const char* studentName);
    // доступ к элементам массива marks
    int getMark(int index);
    void setMark(int mark, int index);
};

```

Добавлен модификатор доступа **private** и прототипы аксессоров и мутаторов.

Файл функций-членов класса **Student**:

Файл Student.cpp

```

#include <iostream>
#include "Student.h"
using namespace std;

// реализация метода вычисления среднего балла
double Student::getAver()
{
    double sum = 0;
    for (int i = 0; i < 3; i++)
    {
        sum += marks[i];
    }
    return sum / 3;
}
// доступ к полю name
// без возможности его изменения

```

```

const char* Student::getName()
{
    return name;
}

void Student::setName(const char* studentName)
{
    // присваивание с проверкой длины
    strcpy_s(name, 20, studentName);
}

// доступ к элементам массива marks
int Student::getMark(int index)
{
    return marks[index];
}

void Student::setMark(int mark, int index)
{
    // присваивание с проверкой оценки
    if (mark < 1 or mark > 12)
    {
        mark = 0;
    }
    marks[index] = mark;
}

```

Добавлены: включение файла `<iostream>` и код аксессуаров и мутаторов. Обратите внимание:

- функция `getName` возвращает константный указатель на строку. Это сделано для того, чтобы через этот указатель нельзя было изменить значение поля объекта;
- функции `setName` (неявно) и `setMark` проверяют параметры и не допускают присвоения переменным-членам класса некорректных значений.

Файл программы, использующей класс `Student`.

Файл `StudentGrades.cpp`

```
// Вычислить среднюю оценку студента
#include <iostream>
#include <iomanip>
#include <conio.h>
#include "Student.h"
using namespace std;

int main()
{
    setlocale(LC_ALL, "");
    cout << "Успеваемость студента."
         << endl << endl;

    // определение объекта
    Student student;

    // присвоение значений объекту
    student.setName("Петров А.И.");
    student.setMark(10, 0);
    student.setMark(10, 1);
    student.setMark(9, 2);

    // выполнение метода объекта
    cout << "Средний балл " << student.getName()
         << " : " << fixed << setprecision(2)
         << student.getAver() << endl;
    _getch();
    return 0;
}
```

Вместо прямого доступа к полям объектов используются аксессоры и мутаторы класса `Student`.

Программа компилируется, запускается и выдает правильный результат.

10. Встроенные (inline) методы в классах

Преждевременная оптимизация — корень всех зол.

Дональд Кнут

10.1. Необходимость использования.

Примеры объявления и использования.

Ограничения при использовании inline методов

Вызов функции C++ требует дополнительных расходов памяти и процессорного времени по сравнению с последовательным выполнением операторов. В некоторых случаях (там, где требуется очень высокая скорость обработки) это может быть критичным. Для снижения дополнительных расходов в C++ предусмотрены встроенные функции (**inline-functions**).

Объявление ключевого слова **inline** в определении функции рекомендует компилятору вместо обычного вызова функции «встроить» в этом месте копию кода функции. Компилятор не обязан следовать рекомендации, и обычно он делает встроенными только самые небольшие функции (следует заметить, что рекурсивные функции принципиально не могут быть встроенными).

Определение функции-члена класса в теле класса тоже делает ее inline-функцией.

Это — еще одна из причин, по которым функции класса, как правило, определяются вне тела класса.

Внутри класса пишутся только самые маленькие по размеру функции-члены класса, обычно — аксессоры и мутаторы.

Так что вернем в нашей программе часть функций-членов класса в тело класса (код программы — в папке *Lesson01\les01_10*).

Файл Student.h

```
class Student
{
private:
    // ФИО
    char name[21];
    // оценки
    int marks[3];

public:
    // чтение закрытых членов класса
    const char* getName()
    {
        return name;
    }

    int getMark(int index)
    {
        return marks[index];
    }

    // запись закрытых членов класса
    void setName(const char* studentName);
    void setMark(int mark, int index);

    // вычисление среднего балла
    double getAver();
};
```

11. Сравнительный анализ структур и классов

При изучении основ C++ мы уже встречались с конструкцией вида `a.b`. Это — обращение к полям (элементам) структуры, например,

```
struct date
{
    int day;
    char* month;
    int year;
};

date birthday;
birthday.year = 2004;
```

В этом фрагменте:

- `date` — имя структуры;
- `birthday` — переменная, которая имеет тип структуры `date`;
- `birthday.year` — обращение к полю `year` переменной `birthday`.

Напоминает — имя класса, объект класса, поле объекта, не правда ли?

Действительно, концепцию структуры, взятую из языка C, в C++ дополнили методами, наследованием, другими полезными возможностями и назвали классом.

Само понятие структуры в C++ также изменилось: `struct` и `class` обозначают, практически, одно и то же.

Структуры C++ так же, как и классы C++, включают в себя поля, методы, модификаторы доступа.

Единственное отличие структуры от класса, — это то, что элементы структуры по умолчанию (без объявления модификаторов доступа) открыты (считаются `public`), тогда как элементы класса по умолчанию закрыты (считаются `private`).

12. Конструктор

12.1. Проблемы, возникающие при использовании неинициализированных переменных

Один из способов, чтобы научиться делать что-то правильно — это сделать сначала неправильно.

Джим Рон

Давайте посмотрим, какое значение будет иметь объект нашего класса `Student` сразу после создания. Для этого прокомментируем в функции `main` строки, в которых полям объекта `student` присваиваются значения:

```
// определение объекта
Student student;
// присвоение значений объекту
//student.setName("Петров А.И.");
//student.setMark(10, 0);
//student.setMark(10, 1);
//student.setMark(9, 2);

// выполнение метода объекта
cout << "Средний балл " << student.getName()
    << " : " << fixed << setprecision(2)
    << student.getAver() << endl;
```

Результат запуска программы — обескураживающий:

Успеваемость студента.

[illegible]

Рисунок 13

Мы видим, что поля объекта имеют бессмысленные значения. Для того, чтобы работать с объектом, его полям (переменным-членам класса) нужно присвоить значения осмысленные. Как мы знаем, это можно сделать с помощью мутаторов (закомментированные строки в примере вверху). Однако, это не слишком удобно, особенно, если полей много, — легко что-то забыть, перепутать... Было бы гораздо правильней задать сразу значения всем переменным-членам класса, которые в этом нуждаются, в одной операции, при создании объекта.

12.2. Понятие конструктора.

Синтаксис объявления

Если ты хочешь что-то в жизни сделать правильно, делай это сам.

Мэттью МакКонахи

В каждом классе, который мы создаем, можно предусмотреть специальную функцию-член класса, предназначенную для создания объекта уже с заданными значениями определенных полей. Этот процесс называется инициализацией объектов, а специальная функция — конструктором (**constructor**). Иногда конструктор в литературе по C++ называют сокращенно **ctor**.

Конструктор — это функция C++, он имеет заголовок и тело, может иметь или не иметь параметры, но у него есть несколько важных отличий от других функций-членов класса:

- конструктор имеет в точности то же имя, что и класс;
- конструктор не возвращает значений (он просто создает объект);

- перед именем конструктора не указывается тип возвращаемого значения, в том числе не ставится и `void`.

Обычно конструкторы объявляются открытыми. Конструктор автоматически вызывается при создании (объявлении) объекта.

Синтаксис объявления конструктора:

```
имя_класса(список_параметров)
{
    список_операторов
}
```

Список параметром может быть пустым.

Добавим в наш класс `Student` необходимые конструкторы. Сначала создадим конструктор, который просто сообщает о своем существовании, — для того, чтобы отследить момент вызова конструктора (код программы — в папке `Lesson01\les01_11`).

Файл `Student.h`

```
class Student
{
private:
    // ФИО
    char name[21];

    // оценки
    int marks[3];

public:
    // конструктор
    Student()
    {
```

```

        std::cout << "Конструктор Student"
                    << std::endl;
    }

    // чтение закрытых членов класса
    const char* getName()
    {
        return name;
    }
    int getMark(int index)
    {
        return marks[index];
    }
    // запись закрытых членов класса
    void setName(const char* studentName);
    void setMark(int mark, int index);

    // вычисление среднего балла
    double getAver();
};

```

В приведенном выше коде непосредственно в теле класса `Student` добавляем конструктор без параметров `Student()`.

Файл `StudentGrades.cpp`

```

// Вычислить среднюю оценку студента

#include <iostream>
#include <iomanip>
#include <conio.h>
#include "Student.h"

using namespace std;

```



```

int main()
{
    setlocale(LC_ALL, "");
    cout << "Успеваемость студента."
         << endl << endl;

    // определение объекта
    Student student;
    cout << "После вызова конструктора" << endl;

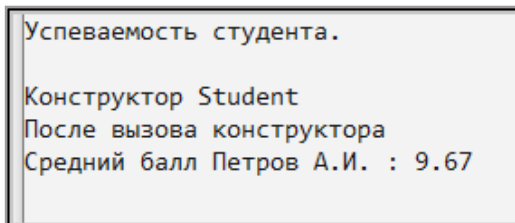
    // присвоение значений объекту
    student.setName("Петров А.И.");
    student.setMark(10, 0);
    student.setMark(10, 1);
    student.setMark(9, 2);

    // выполнение метода объекта
    cout << "Средний балл " << student.getName()
         << " : " << fixed << setprecision(2)
         << student.getAver() << endl;
    _getch();
    return 0;
}

```

В функцию `main` добавляем вывод сообщения на экран после определения объекта `Student`.

Результаты работы программы:



```

Успеваемость студента.

Конструктор Student
После вызова конструктора
Средний балл Петров А.И. : 9.67

```

Рисунок 14

Мы видим, что конструктор вызывается в момент создания объекта `student` в строке: «`Student student;`».

12.3. Конструктор, принимающий параметры

Изменим теперь конструктор в классе `Student` так, чтобы он принимал параметры. Имеет смысл при создании объекта `student` сразу задавать имя (фамилию) студента. Поэтому добавим в существующий конструктор параметр — имя студента.

Уберем также сообщения из конструктора и функции `main`, — они нужны были только для визуализации вызова конструктора. Код программы — в папке `Lesson01\les01_12`).

Файл `Student.h`

```
. . .  
    // конструктор  
    Student(const char* studentName)  
    {  
        setName(studentName);  
    }  
. . .
```

Для запоминания имени студента в переменной-члене класса `name` используем в коде конструктора вызов мутатора `setName`.

Файл `StudentGrades.cpp`

```
. . .  
int main()  
{
```

```

setlocale(LC_ALL, "");
cout << "Успеваемость студента."
      << endl << endl;

// определение объекта
Student student("Петров А.И.");

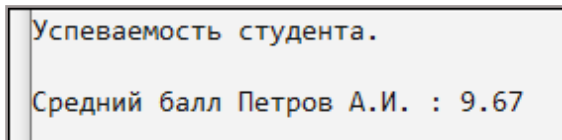
student.setMark(10, 0);
student.setMark(10, 1);
student.setMark(9, 2);

// выполнение метода объекта
cout << "Средний балл " << student.getName()
      << " : " << fixed << setprecision(2)
      << student.getAver() << endl;

_getch();
return 0;
}

```

Результаты работы программы:



```

Успеваемость студента.

Средний балл Петров А.И. : 9.67

```

Рисунок 15

Однако, еще удобнее было бы создавать объект `Student` сразу с имеющимися оценками студента. Так что добавим к конструктору параметры: массив оценок и размер этого массива.

При этом, для большей эффективности класса, немного изменим определение переменных-членов класса:

```
// ФИО
char* name;
// число оценок
int markCount;
// оценки
int* marks;
```

Теперь наши данные (имя и оценки студента) будут храниться в динамической памяти, а количество оценок можно будет изменять.

Соответственно, несколько изменится и реализация функций-членов класса. Код новой версии программы — в папке *Lesson01\les01_13*.

Файл Student.h

```
class Student
{
private:
    // ФИО
    char* name;

    // число оценок
    int markCount;

    // оценки
    int* marks;

    // служебные функции
    void createName(const char* studentName);

public:
    // конструктор
    Student(const char* studentName,
            const int studentMarkCount,
            const int* studentMarks);
```

```

// чтение закрытых членам класса
const char* getName()
{
    return name;
}
int getMark(int index)
{
    return marks[index];
}
// запись закрытых членам класса
void setName(const char* studentName);
void setMark(int mark, int index);
// вычисление среднего балла
double getAver();
};

```

Конструктор вынесен из тела класса, поскольку его размер значительно увеличился. Добавлен прототип закрытой (служебной) функции `createName`. Эта функция запрашивает необходимую динамическую память и копирует параметр «Имя студента» в поле объекта «Имя студента». Такое действие выполняется в конструкторе класса и в мутаторе `setName`, поэтому возникла необходимость в создании функции, чтобы не было дублирования кода.

Файл Student.cpp

```

#include <iostream>
#include "Student.h"
using namespace std;
// конструктор
Student::Student(const char* studentName,
    const int studentMarkCount,
    const int* studentMarks)
{

```

```
// присваивание имени
createName(studentName);

// присваивание списка оценок
marks = new int[studentMarkCount];
for (int i = 0; i < studentMarkCount; i++)
{
    marks[i] = studentMarks[i];
}
markCount = studentMarkCount;
}

// присваивание имени
void Student::createName(const char* studentName)
{
    int nameLength = strlen(studentName);
    name = new char[nameLength + 1];
    for (int i = 0; i <= nameLength; i++)
    {
        name[i] = studentName[i];
    }
}

// запись имени
void Student::setName(const char* studentName)
{
    delete[] name;
    createName(studentName);
}

// запись элементов массива marks
void Student::setMark(int mark, int index)
{
    // проверка индекса
    if (index < 0 or index >= markCount)
    {
```

```

        return;
    }

    // присваивание с проверкой оценки
    if (mark < 1 or mark > 12)
    {
        mark = 0;
    }
    marks[index] = mark;
}

// реализация метода вычисления среднего балла
double Student::getAver()
{
    double sum = 0;
    for (int i = 0; i < markCount; i++)
    {
        sum += marks[i];
    }
    return sum / markCount;
}

```

Конструктор

```

Student::Student(const char* studentName,
                 const int studentMarkCount,
                 const int* studentMarks)
{
    // присваивание имени
    createName(studentName);

    // присваивание списка оценок
    marks = new int[studentMarkCount];

    for (int i = 0; i < studentMarkCount; i++)
    {

```

```

        marks[i] = studentMarks[i];
    }
    markCount = studentMarkCount;
}

```

просто переносит значения параметров «Имя студента», «Количество оценок», «Массив оценок» в соответствующие поля объектов класса `Student`, выделяя при этом для них динамическую память.

Мутатор

```

void Student::setName(const char* studentName)
{
    delete[] name;
    createName(studentName);
}

```

освобождает динамическую память, занятую под имя студента и запоминает в объекте новое имя.

В мутаторе `setMark` добавлена проверка индекса оценки.

```

void Student::setMark(int mark, int index)
{
    // проверка индекса
    if (index < 0 or index >= markCount)
    {
        return;
    }

    . . .
}

```


В методе `getAver` используется добавленное поле `markCount` (количество оценок).

```
for (int i = 0; i < markCount; i++)
{
    sum += marks[i];
}
```

Файл StudentGrades.cpp

```
// Вычислить среднюю оценку студента

#include <iostream>
#include <iomanip>
#include <conio.h>
#include "Student.h"
using namespace std;

int main()
{
    setlocale(LC_ALL, "");
    cout << "Успеваемость студента."
         << endl << endl;

    // создание объекта с заданными значениями
    Student student("Петров А.И.", 3,
new int[3]{ 10, 10, 9 });

    // выполнение метода объекта
    cout << "Средний балл " << student.getName()
         << " : " << fixed << setprecision(2)
         << student.getAver() << endl;

    _getch();
    return 0;
}
```

В функции `main` вместо задания значений объекта `student` с помощью мутаторов `setName`, `setMark` используется конструктор

```
Student student("Петров А.И.", 3,  
new int[3]{ 10, 10, 9 });
```

который заносит необходимые значения в поля объекта сразу при создании объекта. Программа компилируется, запускается, выдает правильный результат.

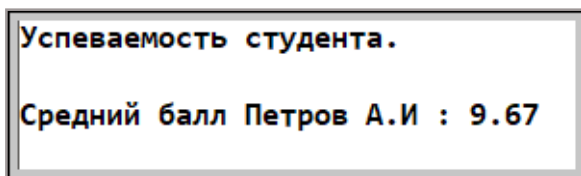


Рисунок 16

12.4. Конструктор по умолчанию

При создании объектов в C++ всегда автоматически вызывается конструктор класса.

Если в классе не определен конструктор (как это было в первых версиях нашей программы), то компилятор создает конструктор по умолчанию, который представляет собой конструктор без параметров.

Разработчик класса может, при необходимости создать свой конструктор по умолчанию (без параметров). Следует обратить внимание, что, если в классе определен какой-то конструктор (хоть без параметров, хоть с параметрами), то компилятором конструктор по умолчанию уже не создается.

12.5. Перегруженные конструкторы

Конструктор, как и любая другая функция C++, может быть перегружен, — то есть в классе может быть несколько конструкторов с разными наборами параметров (в том числе и без параметров). Это бывает весьма полезно, поскольку могут быть разные сценарии создания объектов одного класса.

Например, в нашей программе может возникнуть необходимость создания объекта «Студент» с неизвестными пока оценками, и даже необходимость создания пустого объекта «Студент» (без оценок и фамилии).

Для этого потребуется добавить в класс `Student` еще два конструктора (код программы — в папке `Lesson01\les01_14`):

Файл *Student.h*

```

. . .
// конструкторы
Student();
Student(const char* studentName,
        const int studentMarkCount);
Student(const char* studentName,
        const int studentMarkCount,
        const int* studentMarks);
. . .

```

Файл *Student.cpp*

```

. . .
// конструктор
Student::Student()
{

```

```

    // присваивание имени
    name = nullptr;

    // присваивание списка оценок
    marks = nullptr;
    markCount = 0;
}

// конструктор
Student::Student(const char* studentName,
    const int studentMarkCount)
{
    // присваивание имени
    createName(studentName);

    // присваивание списка оценок
    marks = new int[studentMarkCount];
    for (int i = 0; i < studentMarkCount; i++)
    {
        marks[i] = 0;
    }
    markCount = studentMarkCount;
}

// конструктор
Student::Student(const char* studentName,
    const int studentMarkCount,
    const int* studentMarks)
. . .

```

Конструктор

```

Student::Student(const char* studentName,
    const int studentMarkCount)

```

создает объект «студент» с нулевыми оценками.

Конструктор

```
Student::Student()
```

создает объект «студент» с нулевыми указателями («пустой» объект).

13. Деструктор

Если, возвратясь ночью домой, ты по ошибке выпил вместо воды проявитель, выпей и закрепитель, иначе дело не будет доведено до конца. К. Прутков-инженер. Мысль № 21..

Владимир Савченко

13.1. Утечки ресурсов.

Причины их возникновения и плачевные последствия данного явления

После сделанных изменений в нашем классе `Student` опять появились недоработки (такое часто бывает при попытке улучшить программу...). Дело в том, что объекты класса используют динамическую память. А освобождением ее не занимаются... Значит, происходит утечка памяти, и возможна такая ситуация, когда при многократных вызовах конструктора свободной памяти в компьютере станет недостаточно.

Аналогично, в других ситуациях (в других классах) может происходить утечка и других ресурсов, например, файловых дескрипторов, графических ресурсов, соединений с базой данных. Независимо от вида ресурсов, они должны быть освобождены по окончании использования.

Правильной стратегией использования ресурсов является привязка ресурсов к времени жизни объектов: ресурсы должны выделяться при создании объектов, и освобождаться — при уничтожении объектов.

13.2. Понятие деструктора. Синтаксис объявления

Для выполнения этой задачи в C++ существует пара специальных методов: конструктор и деструктор. С конструктором мы уже познакомились, а деструктор (**destructor**) — это специальная функция-член класса, которая автоматически вызывается при уничтожении объекта.

Синтаксис объявления деструктора:

```
~имя_класса()  
{  
    список_операторов  
}
```

Имя деструктора класса представляет собой имя класса, перед которым находится знак тильды (~), например, деструктор класса **Student** должен называться **~Student**. Деструктор иногда в литературе по C++ сокращенно называют **dtor**.

В отличие от конструктора, деструктор вызывается неявным образом при уничтожении объекта, например, когда функция заканчивает свою работу и все ее локальные переменные, в том числе и объекты классов, уничтожаются. При этом не деструктор освобождает память, занятую объектом — это делает система. Деструктор должен освободить дополнительные ресурсы, используемые объектом, — динамическую память, файл, соединение с базой данных.

Деструктор в классе не обязателен — пока в нашем классе **Student** не использовалась динамическая память, деструктор не был нужен (в этом случае компилятор все же создает «пустой» деструктор).

Деструктор не может иметь никаких параметров и ничего не возвращает. Деструктор не может быть перегружен, в классе может быть только один деструктор. Обычно деструктор объявляется открытым.

13.3. Примеры использования

Добавим необходимый деструктор в класс `Student` (код программы — в папке `Lesson01\les01_15`).

Файл *Student.h*

```

. . .
    // деструктор
    ~Student();
. . .

```

Заметьте: деструктор должен быть объявлен `public`.

Файл *Student.cpp*

```

. . .
// деструктор
Student::~~Student()
{
    // освобождение памяти для имени
    if (name != nullptr)
    {
        delete[] name;
    }

    // освобождение памяти для списка оценок
    if (marks != nullptr)
    {
        delete[] marks;
    }
}

```



```
cout << "Отработал деструктор" << endl;
system("pause");
}
. . .
```

Поскольку в классе есть конструктор, который не запрашивает динамическую память, мы должны сделать необходимые проверки перед освобождением.

Перед выходом выдается отладочное сообщение, чтобы можно было увидеть, когда отработывает деструктор.

(Сообщение от деструктора выдается после того, как пользователь завершает работу программы нажатием любой клавиши — после выдачи данных о среднем балле).

Успеваемость студента.

Средний балл Петров А.И. : 9.67

Отработал деструктор

Для продолжения нажмите любую клавишу . . . ■

Рисунок 17

Мы видим, что деструктор был вызван в процессе завершения функции `main`, когда локальная переменная `Student student` прекращает свое существование.

13.4. Указатели на объекты. Массивы объектов

Бог всегда на стороне больших батальонов.

Маршал XVII века Жак д'Эстамп

Созданный новый класс в C++, как мы знаем, — это новый тип данных, который обладает тем же набором

возможностей, что и встроенные типы данных. Следовательно, доступ к объектам класса возможен так же и через ссылки, и через указатели, и объекты могут быть элементами массива.

Рассмотрим новый вариант программы, использующей класс `Student`, в котором объекты класса будут организованы в виде массива (код программы — в папке `Lesson01\les01_16`).

```
// Вычислить среднюю оценку студентов
#include <iostream>
#include <iomanip>
#include <conio.h>
#include "Student.h"
using namespace std;

int main()
{
    setlocale(LC_ALL, "");
    cout << "Успеваемость студентов."
          << endl << endl;

    // размер массива объектов
    const int size = 2;

    // создание и инициализация
    // динамического массива объектов
    Student* students = new Student[size]
    {
        {"Студент 1", 3, new int[3]{ 10, 10, 9 }},
        {"Студент 2", 3, new int[3]{ 8, 10, 8 }}
    };

    // работа с массивом объектов
    double sum = 0;
```

```

for (Student* stud = students;
     stud < students + size; stud++)
{
    double aver = stud->getAver();
    cout << "Средний балл " << stud->getName()
         << " : " << fixed << setprecision(2)
         << aver << endl;
    sum += aver;
}

cout << endl;
cout << "Средний балл по группе: "
     << " : " << fixed << setprecision(2)
     << sum / size << endl;

delete[] students;
_getch();
return 0;
}

```

Мы создаем динамический массив объектов класса и инициализируем его, неявно вызывая конструктор для каждого объекта точно так же, как мы инициализировали отдельные переменные — объекты.

```

Student* students = new Student[size]
{
    {"Студент 1", 3, new int[3]{ 10, 10, 9 }},
    {"Студент 2", 3, new int[3]{ 8, 10, 8 }}
};

```

Далее мы обрабатываем элементы массива, обращаясь к ним через указатель.

```

for (Student* stud = students;
     stud < students + size; stud++)
{
    double aver = stud->getAver();
    cout << "Средний балл " << stud->getName()
         << " : " << fixed << setprecision(2)
         << aver << endl;
    sum += aver;
}

```

Для доступа через указатель на объект к элементам объекта (переменным-членам класса и функциям-членам класса) используется оператор-стрелка выбора элемента (->) между именем объекта и именем элемента, например `stud->getAver()`.

Не забываем освободить память

```
delete[] students;
```

В деструктор внесем небольшие изменения, чтобы удобнее было наблюдать его работу.

```

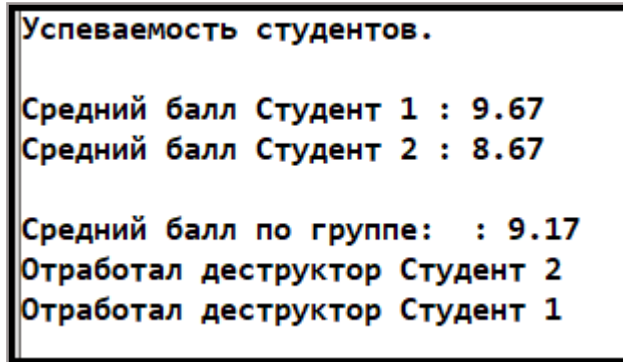
Student::~~Student()
{
    cout << "Отработал деструктор " << name << endl;
    // освобождение памяти для имени
    if (name != nullptr)
    {
        delete[] name;
    }

    // освобождение памяти для списка оценок
    if (marks != nullptr)
    {

```

```
        delete[] marks;  
    }  
}
```

Результат работы программы:



Успеваемость студентов.

Средний балл Студент 1 : 9.67
Средний балл Студент 2 : 8.67

Средний балл по группе: : 9.17
Отработал деструктор Студент 2
Отработал деструктор Студент 1

Рисунок 18

Обратите внимание, деструкторы класса вызываются в порядке, обратном к порядку вызовов конструкторов.

14. Резюме

Объектно-ориентированное программирование (ООП) представляет собой подход к созданию программ, при котором значительно улучшается и процесс разработки, и полученный в результате код.

В объектно-ориентированном программировании основную роль играют данные. Структура программы соответствует данным, которые она обрабатывает: сначала определяются виды данных и связи между ними, а уже потом — методы обработки (функции).

Три принципа ООП: инкапсуляция, наследование и полиморфизм.

Инкапсуляция — это свойство системы, позволяющее тесно связать данные и методы работы с ними внутри класса, и сделать данные и детали реализации недоступными для других частей системы.

Наследование — это отношение между классами, при котором один класс повторяет структуру и поведение другого класса (одиночное наследование) или других (множественное наследование) классов.

Класс, который наследуется, называется суперклассом, надклассом, базовым или родительским классом.

Класс, производный от суперкласса, называется подклассом, производным или дочерним классом.

Полиморфизм — это возможность применения одноименных методов с одинаковыми или различными наборами параметров в одном классе или в группе классов, связанных отношением наследования.

Класс является инструментальным описанием структуры и поведения однотипных объектов, а объект — это отдельный представитель (экземпляр) класса, имеющий свое собственное индивидуальное состояние.

Класс состоит из заголовка класса и тела класса. Определение класса равносильно определению нового типа данных. Для того, чтобы в объекте хранились значения, нужно в классе определить соответствующие поля объекта посредством объявления переменных-членов класса.

Синтаксис обращения к переменной-члену класса: `имя_объекта.имя_переменной`, например,

```
student.name = "Johnson";
```

Спецификаторы доступа:

- **public** делает члены класса открытыми. Это значит, что доступ к членам класса возможен из любой части программы;
- **private** делает члены класса закрытыми, то есть доступ к ним возможен только внутри класса, для других частей программы эти члены недоступны.
- **protected** открывает доступ к членам класса только для дочерних классов (и дружественных функций, но об этом потом). Такие члены класса называют защищенными.

Функции, принадлежащие классу, называются функциями-членами класса.

В коде функции-члена класса можно обращаться не только к локальным переменным и параметрам функции, но и к переменным-членам класса.

Обращение к переменной-члену класса в функции-члене класса представляет собой обращение к соответствующему полю соответствующего объекта.

Синтаксис вызова функции-члена: `имя_объекта.имя_функции(список_параметров)`, например,

```
student.getAver();
```

Функция-член класса может быть вынесена за пределы объявления класса. При этом в теле класса объявляется прототип функции, а в реализации функции в заголовке перед именем функции пишется имя класса и оператор разрешения области действия, например

```
Student::getAver()
```

Переменные-члены класса обычно объявляются как закрытые (`private`).

В классах часто предусматриваются открытые функции-члены, предназначенные для установки значения (`set`) или получения значения (`get`) закрытых переменных-членов. Имена этих функций могут быть какими угодно, но обычно их начинают, соответственно, с `set` или `get`: `setName`, `getMarks`.

Set-функции принято называть мутаторами (`mutators`) или модификаторами, поскольку они изменяют значения полей объекта.

Get-функции, соответственно принято называть аксессорами (`accessors`) или инспекторами (`inspectors`), потому что они осуществляют доступ (`access`) к значениям, просмотр значений.

Определение функции-члена класса в теле класса делает ее inline-функцией.

Структуры C++ так же, как и классы C++, включают в себя поля, методы, модификаторы доступа. Единственное отличие структуры от класса, — это то, что элементы структуры по умолчанию (без объявления модификаторов доступа) открыты (считаются **public**), тогда как элементы класса по умолчанию закрыты (считаются **private**).

Каждый класс предусматривает конструктор для инициализации объекта класса при его создании. **Конструктор** — это специальная функция-член класса, которая должна определяться с тем же именем, что и класс, например **Student()**. Конструкторы не могут возвращать значений и потому перед ними не указывается тип возвращаемого значения (даже **void**).

Конструкторы могут быть перегружены — в классе может быть несколько конструкторов.

Конструктор, не принимающий аргументов, является конструктором по умолчанию. В классе, не имеющим конструктора, компилятор генерирует конструктор по умолчанию. Разработчик класса может также явно определить конструктор по умолчанию. Если в классе определен какой-либо конструктор, то конструктор по умолчанию компилятором не создается.

Деструктор — это специальная функция-член класса, которая автоматически вызывается при уничтожении объекта.

Имя деструктора класса представляет собой имя класса, перед которым находится знак тильды (~), например, деструктор класса **Student** должен называться **~Student()**.

Доступ к объектам класса возможен так же и через ссылки, и через указатели, объекты могут быть элементами массива.

Для доступа через указатель на объект к элементам объекта (переменным-членам класса и функциям-членам класса) используется оператор-стрелка выбора элемента (->) между именем объекта и именем элемента, например `stud->getAver()`.

15. Терминология

- class (класс);
- private: (закрытый, приватный);
- protected: (защищенный);
- public: (открытый, публичный);
- аксессор (accessor);
- деструктор (destructor, dtor);
- класс (class);
- конструктор (constructor, ctor);
- инкапсуляция (encapsulation);
- модификатор доступа (access specifier);
- мутатор (mutators);
- наследование (inheritance);
- объект (object);
- объектно-ориентированное программирование (ООП);
- оператор-стрелка выбор элемента (->) (arrow operator);
- оператор-точка выбор элемента (.) (dot operator);
- оператор разрешения области (::) (scope resolution operator);
- переменная-член класса, метод класса, атрибут класса) (data member);
- полиморфизм (polymorphism);
- процедурное программирование;
- функция-член класса, метод класса (member function).

16. Домашнее задание

1. **Создать класс «Лифт», представляющий собой предельно упрощенную модель лифта.** Класс должен обеспечить:

- ▷ установку диапазона движения лифта (нижний и верхний этаж);
- ▷ включение / выключение лифта;
- ▷ возвращение текущего состояния лифта (работает / не работает);
- ▷ возвращение текущего положения лифта (этаж);
- ▷ обработку вызова лифта (этаж).

Написать программу, тестирующую класс «Лифт».

2. **Написать программу «Стоимость обоев».** Программа запрашивает:

- ▷ количество комнат в квартире, в которых планируется клеить обои,
- ▷ параметры каждой комнаты,
- ▷ параметры каждого вида обоев, которые планируется использовать.

В результате расчета программа выдает: необходимое количество рулонов каждого вида, общую стоимость закупки обоев.

Разработать и использовать в программе классы: «Квартира», «Комната», «РулонОбоев».

Атрибуты (поля) квартиры: список комнат.

Атрибуты комнаты: название, размеры, клеить потолок или нет. Атрибуты рулона: название, размеры, цена.

3. Написать программу «Группа студентов».

Программа получает следующие данные:

- ▷ список студентов группы,
- ▷ список предметов для группы,
- ▷ таблицу оценок студентов группы по всем предметам.

Данные вводятся из соответствующих файлов. Имена файлов запрашивает программа. Программа выводит следующие данные:

- ▷ таблицу оценок студентов по предметам,
- ▷ средние оценки студентов,
- ▷ средние оценки по предметам,
- ▷ средний балл группы,
- ▷ максимальные и минимальные оценки по предметам с указанием студентов.

Разработать и использовать в программе классы: «Группа», «Студент», «Предмет».

Атрибуты группы: название, список студентов, список предметов.

Атрибуты студента: имя, список оценок.

Атрибуты предмета: название.