Лабораторная работа №3
Классы и инкапсуляция

Организация исходного кода в проекте

Вы могли заметить, что количество исходного кода при разработке программ растёт достаточно быстро, и в нём становится сложно ориентироваться. Для того, чтобы быстро находить требуемый блок кода, разработчики используют различные договоренности по организации проекта и разделению исходного кода на файлы.

Особенностью компиляторов Си++ является требование объявления функции до её вызова в коде. То есть, если вы хотите вызвать одну функцию внутри другой, вы должны убедиться, что вызываемая функция объявлена в исходном коде выше. Подобное ограничение приводит к сложностям организации исходного кода в программе: ведь с таким ограничением вы не можете сгруппировать функции так, как удобно вам – вам приходится также думать и о компиляции программы.

Обойти данное ограничение можно с помощью использования ***прототипов функций*** и ***заголовочных файлов***.

Прототип функции – это объявление функции, реализация которой может быть в другом участке кода.

Например, для функции суммирования чисел массива:

```
double Sum(double* values, int valuesCount)
{
   double sum = 0.0;
   for (int i = 0; i < valuesCount; i++)
   {
      sum += values[i];
   }
   return sum;
}
```

Прототип функции будет выглядеть следующим образом:

```
double Sum(double* values, int valuesCount);
```

То есть прототип полностью повторяет описание функции (название, возвращаемый тип данных, набор входных аргументов), но без тела самой функции в фигурных скобках.

Цель прототипа – объявить функцию до её вызова, переместив реализацию в другой участок кода. Это может быть ниже точки вызова функции, или даже в другом файле. Например:

```
double Sum(double* values, int valuesCount);

void main()
{
   double* values = new double[]{1.0, 15.6, 18.2, 45.2};
   double sum = Sum(values, 4);
}

double Sum(double* values, int valuesCount)
{
   double sum = 0.0;
   for (int i = 0; i < valuesCount; i++)
   {
      sum += values[i];
   }
   return sum;
}
```

В данном примере, реализация функции Sum находится ***после*** её вызова в функции Main(). Без размещения прототипа функции перед функцией main() компилятор Си++ выдал бы ошибку, что идентификатор Sum не

найден. Однако при написании прототипа такой ошибки не будет, так как мы заранее предупредили компилятор о существовании такой функции в программе с помощью прототипа.

Применение прототипов позволяет: а) повысить скорость изучения файлов исходного кода, написанного другими разработчиками; б) организовать структуру файлов нашей программы для повышения читаемости. Рассмотрим, как достигается каждое из этих преимуществ.

Рекомендуемая (для комфортного ориентирования в проекте) длина файлов исходного кода составляет 300 строк кода, а при количестве строк свыше 1000 файл рекомендуется разделить на несколько. Таким образом, при средней длине функций в 25 строк кода, в одном файле может быть от 12 до 40 функций. Без прототипов функций, разработчику придется просматривать весь файл целиком, находя объявления функций в коде и запоминая наборы входных аргументов. Это будет занимать много времени каждый раз, когда разработчику будет нужно найти функцию для вызова в своём коде.

Однако, если взять за правило писать прототипы всех функций в начале файла, а затем ниже оставлять их реализацию, поиск нужной функции в файле ускорится в разы. Так, все функции файла смогут поместиться на одном экране в виде удобного списка:

```
// Прототипы функций наглядно перечислены в начале файла
// Очень легко найти нужную функцию, чтобы вызвать её в своём коде
double Max(double* values, int count);
double Min(double* values, int count);
double Sum(double* values, int count);
double Average(double* values, int count);
void SortAscending(double* values, int count);
void SortDescending(double* values, int count);
int IndexOf(double* values, int count, double findedValue);
int IndexOfLast(double* values, int count, double findedValue);
...

// Ниже идут реализации всех функций, которые могут занять несколько сотен строк кода
// Разработчику не нужно тратить время, листая сотни строк реализации функций
double Max(double* values, int count)
{
   ...
}
double Min(double* values, int count)
{
   ...
}

double Sum(double* values, int count)
{
   ...
}

...
```

Разумеется, написание прототипов занимает время. Однако время, сэкономленное на поиске требуемых функций в исходном коде, значительно превышает время, затраченное на написание прототипов.

Теперь перейдем к разделению проекта на файлы и организации файлов проекта в удобную для работы структуру. Как сказано выше, компилятор Си++ накладывает ограничения на организацию функций в программе. Чтобы сделать использование функций в программе более гибким, в Си++ выделяют два типа файлов – заголовочные файлы с расширением *.h и файлы исходного кода с расширением *.cpp. В заголовочных файлах (Header files) хранятся объявления структур, перечислений и других пользовательских типов данных, а также прототипы функций для работы с ними. В файлах исходного кода (Source files) хранятся реализации функций, описанных в заголовочных файлах. Следует придерживаться следующим правилам:

1) Для каждой структуры, перечисления или класса создается отдельный заголовочный файл, имя которого совпадает с именем типа данных. Например, для структуры Movie должен быть создан файл

Movie.h, в котором хранится сама структура, а ниже перечислены прототипы функций для работы с этой структурой: ReadMovie(), WriteMovie(), MakeMovie(), CopyMovie() и другие.

2) Каждому заголовочному файлу *.h соответствует собственный файл исходного кода с таким же названием. Например, для файла Movie.h должен быть создан файл Movie.cpp, в котором будут храниться реализации всех функций (ReadMovie(), WriteMovie(), MakeMovie(), CopyMovie() и др.)
3) Если функции не относятся к определенной структуре, но имеют общее назначение, их также можно выделить в отдельные файлы. Например, функции, отвечающие за пользовательский интерфейс, чтение/запись в файл, работу с базой данных, можно вынести в отдельные файлы соответственно UserInterface.h/UserInterface.cpp, File.h/File.cpp, Database.h/Database.cpp. Прототипы также размещаются в заголовочных файлах, а реализации функций в файлах исходного кода.
4) Если файлов в проекте слишком много, их можно поместить в подпапки. Например, в проекте можно создать подпапки logic и interface, чтобы отдельно хранить файлы с логикой программы и пользовательским интерфейсом. Можно создать подпапки lab1, lab2 и т.д. для хранения файлов отдельных лабораторных работ.

Разделение программы на файлы также требует определенных знаний о подключении файлов друг в друга для возможности их совместной работы. Подключение файлов между собой осуществляется с помощью директивы include. Так, если вы хотите использовать функции из файла Movie.h в файле main.h, то в начале файла main.h вам необходимо прописать строку:

```
#include "Movie.h"
```

Обратите внимание, что имя файла указано в кавычках, в то время как подключение стандартных библиотек типа iostream осуществляется с помощью угловых скобок:

```
#include <iostream>
```

Угловые скобки указывают компилятору искать файл в системной папке самой среды разработки. Кавычки указывают компилятору искать файл в папке вашего проекта. Также учтите, что в кавычках пишется не просто название файла, а относительный путь до файла:

```
#include "Movie.h"         // файл в той же папке, что и main.h
#include "lab3\Movie.h"    // файл лежит в подпапке lab3 рядом с main.h
#include "..\Movie.h"      // файл лежит на одну папку выше, чем и main.h
```

Если вы укажете неверный путь файла, то компилятор может сообщить об ошибке, что указанный файл не найден.

Также директиву include необходимо писать в начале файлов исходного кода. Например, в файле Movie.cpp в начале файла нужно также написать:

```
#include "Movie.h"
```

Это будет указанием компилятору, что функции в файле Movie.cpp реализуют прототипы, описанные в файле Movie.h. В противном случае компилятор может выдать ошибку об объявлении двух функций с одинаковыми именами.

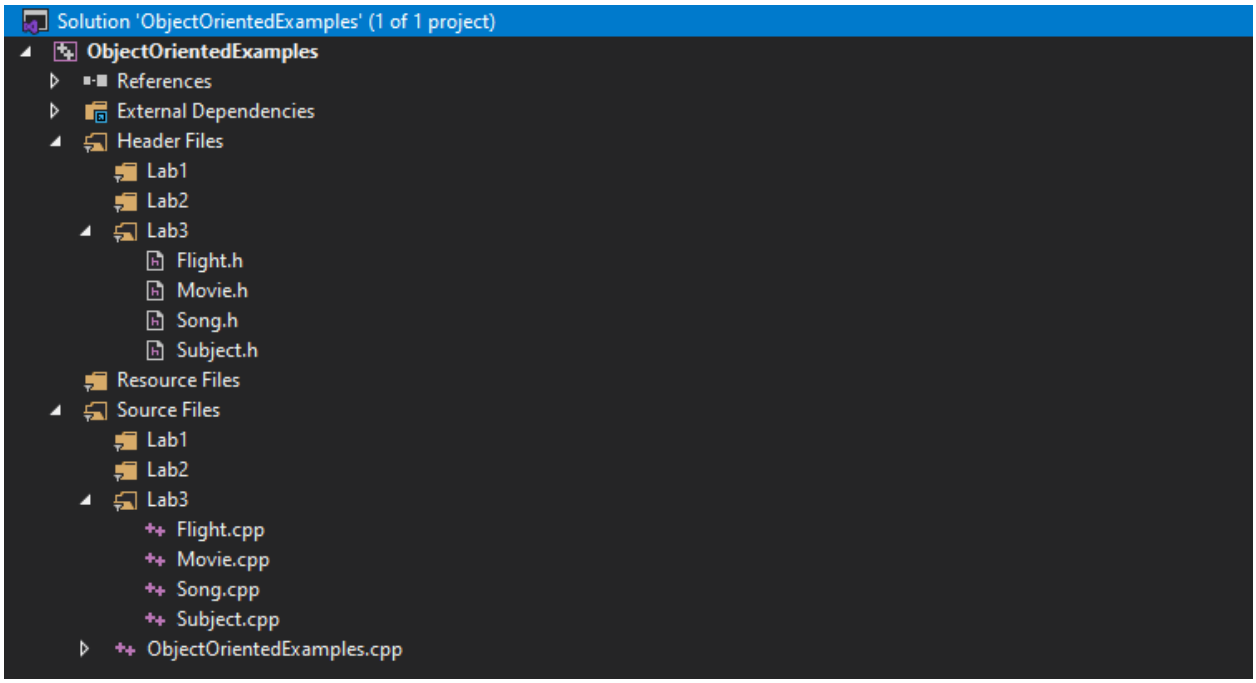
Еще одно требование проекта на отдельные файлы – написание в начале каждого файла директивы #pragma once. Данная директива должна быть указана до любых директив #include как в заголовочных файлах, так и файлах исходного кода:

```
#pragma once
#include "Movie.h"
```

Данная директива является защитой от повторного подключения одного и того же файла в разных местах программы. Так, любое размещение директивы include приводит к подключению указанного файла. Если файл подключить дважды, то компилятор сообщит об ошибке – несколько функций/структур имеющих одинаковые названия. Директива pragma once указывает компилятору, что данных файл может быть

подключен только один. Таким образом, если во время компиляции компилятор повторно найдет директиву на подключение уже подключенного файла, он пропустит данную директиву.

Отдельно стоит сказать об представлении проектов в Visual Studio. Среда разработки Visual Studio самостоятельно предлагает группировать заголовочные файлы и файлы исходного кода по отдельным папкам Header Files и Source Files соответственно. Опыт работы с исходным кодом показывает, что у такой организации есть свои преимущества, однако для разработчика это накладывает ограничение дублирования структуры подпапок проекта в двух папках проекта. Так, если вы создадите подпапку lab3 в папке Header Files, то такую же подпапку lab3 необходимо создать и в папке Source Files. Это требование не влияет на компиляцию программы, только на удобство работы разработчика с исходным кодом. Например:



Пример структуры проекта.
Проект разделен на две подпапки заголовочных файлов и файлов исходного кода,
в каждой подпапке одинаковая структура подпапок для лабораторных работ.
Файлы именованы по имени структур, которые в них хранятся

Также стоит помнить, что представление файлов в проекте может отличаться от их реального размещения на физическом диске. Если файлы проекта в среде разработки сгруппированы по подпапкам, то на физическом диске подпапки автоматически не создаются. Это должен контролировать сам разработчик. Также на физическом диске не будет подпапок Header Files и Source Files, это лишь представление внутри самой среды разработки. Помните об этом при указании относительного пути до файла в директиве include.

Структуры с полями-массивами

В качестве полей структуры могут быть не только простые типы данных int, double, bool, но и указатели, массивы, другие структуры.

Массив может храниться с фиксированным размером, а может по указателю с дополнительным полем – размером массива. Например, для описания треугольника необходимо хранить координаты трёх точек. Примером такой структуры может быть следующий код:

```
struct Triangle
{
   double X1;
   double X2;
   double X3;
   double Y1;
   double Y2;
   double Y3;
}
```

Однако, такая структура будет неудобна в работе, так как обращение к каждой координате осуществляется по прямому обращению к полю, и нельзя обработать все координаты в едином цикле. Поэтому, координаты могут быть сгруппированы в массивы:

```
struct Triangle
{
   double X[3];
   double Y[3];
}
```

Теперь обращение к точке осуществляется через индекс, и, следовательно, координаты могут обрабатываться в цикле:

```
 Triangle triangle;
 ...
 // Перемещение треугольника на 5 пунктов
 triangle.X1 += 5;
 triangle.X2 += 5;
 triangle.X3 += 5;
 triangle.Y1 += 5;
 triangle.Y2 += 5;
 triangle.Y3 += 5;
```

```
Tringle triangle;
...
// Перемешение треугольника на 5 пунктов
for (int i = 0; i < 3; i++)
{
   triangle.X[i] += 5;
   triangle.Y[i] += 5;
}
```

Возможность обработки данных в цикле даёт разработчику бóльшую свободу в разработке алгоритмов.

Треугольник всегда описывается тремя точками, поэтому размер массива фиксирован и равен 3. Однако, количество элементов массива может быть не известно заранее. Например, многоугольник может быть как из трёх точек, так из десяти и более. Для хранения массивов с неизвестным заранее количеством элементов можно воспользоваться одним из двух приёмов. Первый заключается в хранении в качестве поля заведомо большого массива, а в дополнительном поле хранить реально используемое количество элементов. Например, для многоугольника такая структура могла бы выглядеть следующим образом:

```
struct Polygon
{
   double X[100];
   double Y[100];
   int PointsCount;
}
```

При создании переменной такой структуры автоматически выделяется память для массивов из ста элементов. Однако реальное количество используемых точек будет храниться в переменной PointsCount:

```
Polygon polygon;
cout << "Введите количество точек многоугольника: ";
cin >> polygon.PointsCount;
for (int i = 0; i < polygon.PointsCount; i++)
{
   cout << "Введите X[" << i << "]: ";
   cin >> polygon.X[i];
   cout << "Введите Y[" << i << "]: ";
   cin >> polygon.Y[i];
}
```

У такого подхода есть недостаток – для каждой переменной структуры всегда выделяется память для 100 точек, даже если они не будут использоваться. Это приводит к неэффективному использованию памяти.

Рассмотрим второй вариант – хранение массива по указателю:

```
struct Polygon
{
   double* X;
   double* Y;
   int PointsCount;
}
```

Данный вариант предполагает хранение только указателя на массив, в то время как сам массив будет создаваться во вне:

```
Polygon polygon;
cout << "Введите количество точек многоугольника: ";
cin >> polygon.PointsCount;

// Выделяем память под нужное количество элементов
polygon.X = new double[polygon.PointsCount];
polygon.Y = new double[polygon.PointsCount];

for (int i = 0; i < polygon.PointsCount; i++)
{
   cout << "Введите X[" << i << "]: ";
   cin >> polygon.X[i];
   cout << "Введите Y[" << i << "]: ";
   cin >> polygon.Y[i];
}
...
delete[] polygon.X;
delete[] polygon.Y;
```

Вариант с хранением массивов по указателю позволяет хранить только то количество точек, которое нужно для данного объекта. Но, устраняя недостатки первого решения, вариант добавляет собственные:

1) в большинстве случаев массивы для полей надо создавать динамически;
2) нужно помнить об освобождении памяти после работы с объектом.

Несмотря на недостатки, в большинстве случаев второй способ хранения массивов в полях структуры является более предпочтительным. Выбор способа во многом определяется структурой, которую вы хотите создать.

Стоит отметить, что оба варианта имеют существенный недостаток – если забыть обновить значение поля PointsCount при добавлении/удалении точек, или случайно присвоить неправильное значение, то в последствии это приведет к обращению за пределы массива и ошибке при выполнении.

Примечание: если вам нужно хранить в объекте неопределенное количество элементов, то для этих целей лучше использовать список вместо массива. Список обладает рядом преимуществ, однако работа с таким полем будет отличаться от работы с массивом. Подробнее можно почитать здесь [1, 2, 3].

Агрегирование

Как было отмечено ранее, структуры могут хранить в качестве собственных полей объекты других структур. Такая взаимосвязь структур в коде позволяет описать взаимосвязи реальных объектов из предметной области. Например, в следующем примере описаны две структуры: одна структура описывает данные о преподавателе, а вторая структура описывает дисциплину:

```
struct Teacher
{
    string Surname;     // Фамилия
    string Initials;    // Инициалы
    string Position;    // Должность
};

struct Subject
{
    string Title;       // Название
    int Hours;          // Кол-во часов
    Teacher Lecturer;   // Лектор
    int Mark;           // Оценка
};
```

В данном примере объект дисциплины Subject хранит в качестве поля объект преподавателя Teacher. Такая организация структур предпочтительнее варианта, при котором дисциплина хранила бы три дополнительных поля LecturerSurname, LecturerInitials и LecturerPosition – она интуитивно понятна; имеет преимущества с точки зрения модификации программы в случаях, когда необходимо добавить дополнительные поля о преподавателе или скопировать данные о преподавателе в объект другой дисциплины.

Работа с объектами таких структур выглядит следующим образом:

```
Subject subject;
subject.Title = "Объектно-ориентированное программирование";
subject.Hours = 160;
subject.Mark = 5;

subject.Lecturer.Surname = "Смирнов";
subject.Lecturer.Initials = "И.В.";
subject.Lecturer.Position = "Профессор";
```

Для создания переменной структуры используется также её название и уникальное имя переменной. В следующих строках мы можем обратиться через оператор `.` к полям и присвоить данные. Обращение к полю преподавателя идет по имени самого поля `Lecturer`, а не названия структуры `Teacher` (аналогично, например, полю Hours, когда обращение идет к имени поля, а не его типу int).

Однако мы помним, что у поля Lecturer есть собственные поля, и обращение к ним осуществляется также через точку. Таким образом появляется запись последовательного обращения к полям, разделенных точками:

```
subject.Lecturer.Surname = "Смирнов";
```

Такое взаимодействие типов данных, при котором один тип данных является полем другого типа данных, называется ***агрегированием***. В нашем примере, структура Subject ***агрегирует*** структуру Teacher, т.е. хранит её в качестве своего поля. Subject – ***агрегирующая*** структура, Teacher – ***агрегируемая*** структура. В данном определении не зря упоминаются *типы данных*, а не просто структуры. Фактически, когда перечисление является полем структуры – такая связь также называется агрегированием, или, другими словами, структура агрегирует перечисление. Кроме того, далее мы познакомимся с понятием классов, которые также могут агрегировать структуры, перечисления и другие классы.

В приведенном примере стоит также обратить внимание на то, что при создании переменной subject сразу же создается объект Teacher, который будет храниться в поле Lecturer. Дополнительно объявлять переменную для преподавателя не надо. Если мы создадим еще одну переменную типа Subject, то для неё также будет создан собственный объект преподавателя.

Агрегирование Teacher структурой Subject не подразумевает, что мы не можем использовать структуру преподавателя самостоятельно. Переменные типа Teacher могут создаваться отдельно от Subject и использоваться независимо.

Агрегирование указателя на объект

Если подумать, текущая версия программы имеет недостаток – для каждой дисциплины создается отдельный экземпляр преподавателя. Хотя, как мы знаем из реального опыта, один преподаватель может вести несколько дисциплин. Мы могли бы копировать данные преподавателя между дисциплинами следующим образом:

```
Subject subject;
subject.Title = "Объектно-ориентированное программирование";
subject.Hours = 160;
subject.Mark = 5;

subject.Lecturer.Surname = "Смирнов";
subject.Lecturer.Initials = "И.В.";
subject.Lecturer.Position = "Профессор";

Subject subject2;
subject2.Title = "Технологии разработки ПО";
subject2.Hours = 108;
subject2.Mark = 4;

subject2.Lecturer.Surname = subject.Lecturer.Surname;
subject2.Lecturer.Initials = subject.Lecturer.Initials;
subject2.Lecturer.Position = subject.Lecturer.Position;
```

Но данный подход имеет свои недостатки. Во-первых, если данные о преподавателе поменяются (например, должность), то потребуется перебирать все существующие дисциплины, искать нужного преподавателя и обновлять данные. Это дополнительный код, и может быть затратно по времени выполнения, если программа будет хранить данные о всех дисциплинах университета. Во-вторых, если потребуется добавить дополнительное поле в структуру преподавателя (например, ученую степень), то опять же придется модифицировать код практически во всех местах, где идет обращение к полю Lecturer.

Решением данной проблемы может стать хранение объекта в поле по указателю – ***агрегирование указателя на объект***. Объявление структуры Subject примет следующий вид:

```
struct Subject
{
    string Title;       // Название
    int Hours;          // Кол-во часов
    Teacher* Lecturer;  // Лектор
    int Mark;           // Оценка
```

```
};
```

Теперь поле Lecturer является указателем на тип Teacher. Добавление одного знака `*` кардинально меняет логику работы с данной структурой. Во-первых, при создании переменной Subject в качестве поля Lecturer будет создаваться указатель на Teacher, а не сам объект Teacher. Объект Teacher придется создавать отдельно и помещать его адрес в поле. Однако, как мы помним из работы с указателями, указатели могут ссылаться на один и тот же объект. Это значит, что, создав экземпляр преподавателя Teacher, мы можем поместить адрес одного преподавателя в любое количество дисциплин. Код работы со структурой Subject будет выглядеть следующим образом:

```
Teacher lecturer;
lecturer.Surname = "Смирнов";
lecturer.Initials = "И.В.";
lecturer.Position = "Профессор";

Subject subject;
subject.Title = "Объектно-ориентированное программирование";
subject.Hours = 160;
subject.Mark = 5;
subject.Lecturer = &lecturer;

Subject subject2;
subject2.Title = "Технологии разработки ПО";
subject2.Hours = 108;
subject2.Mark = 4;
subject2.Lecturer = &lecturer;
```

В примере сначала создается объект преподавателя и инициализируются все его данные. Далее создаются переменные дисциплин. Но вместо инициализации преподавателя по отдельным внутренним полям, в поля Lecturer помещается адрес переменной lecturer. Инициализация объекта дисциплины стала более лаконичной и занимает меньше кода. Но главное преимущество заключается в том, что при изменении данных в переменной lecturer они автоматически изменятся во всех дисциплинах с этим преподавателем, так как это один и тот же объект. В этом можно убедиться, добавив следующие строчки кода:

```
cout << "Лектор дисциплины #1: " << subject.Lecturer->Surname << endl;
cout << "Лектор дисциплины #2: " << subject2.Lecturer->Surname << endl;

lecturer.Surname = "Быков";

cout << "Лектор дисциплины #1: " << subject.Lecturer->Surname << endl;
cout << "Лектор дисциплины #2: " << subject2.Lecturer->Surname << endl;
```

Результатом работы программы станет следующий текст:

```
Лектор дисциплины #1: Смирнов
Лектор дисциплины #2: Смирнов
Лектор дисциплины #1: Быков
Лектор дисциплины #2: Быков
```

Объект преподавателя в примере выше также может быть создан в динамической памяти, или даже без создания временной переменной lecturer:

```
Subject subject;
subject.Title = "Объектно-ориентированное программирование";
subject.Hours = 160;
subject.Mark = 5;

// создаем новый объект динамически и сразу помещаем в поле
```

```
subject.Lecturer = new Teacher();

// инициализируем поля преподавателя через переменную дисциплины
subject.Lecturer->Surname = "Смирнов";
subject.Lecturer->Initials = "И.В.";
subject.Lecturer->Position = "Профессор";

...

subject2.Lecturer = subject.Lecturer;
```

При этом нужно помнить, что данные о преподавателе можно изменить через любую переменную дисциплины, которая ссылается на объект преподавателя. С одной стороны, это удобно при модификации данных. С другой стороны, при невнимательности можно потерять данные о преподавателе.

Вы могли обратить внимание, что теперь в записи присвоения фамилии через переменную дисциплины или выводе фамилии на экран, для обращения к внутренним полям преподавателя используется оператор `->` вместо `.`:

```
cout << "Лектор дисциплины: " << subject.Lecturer->Surname << endl;
subject.Lecturer->Position = "Профессор";
```

Как мы помним, при работе с объектами структуры через указатель, обращение к полям выполняется через оператор `->`. Это правило действует и для агрегируемых структур. Так как поле Lecturer является указателем на структуру, то обращение к полям структуры через указатель также будет выполняться через оператор `->`. При этом, если переменная subject была объявлена как указатель, тогда обращение к её внутренним полям также осуществлялось бы через оператор `->`, что привело бы к последовательности операторов `->` при обращении к внутренним полям.

```
Subject* subject = new Subject();
subject->Mark = 5;
subject->Lecturer = new Teacher();
subject->Lecturer->Surname = "Смирнов";
```

Разница использования `->` и `.` при обращении к агрегируемым структурам сначала может сбивать вас с толку. Однако несколько неудачных компиляций научат вас быть внимательным.

Передача агрегирующих структур в функции никак не отличается от передачи в функцию обычных структур. Так, для структуры Subject можно создать функции-конструкторы, функции копирования и сеттеры:

```
void SetTitle(Subject& subject, string title);
void SetHours(Subject& subject, int hours);
void SetLecturer(Subject& subject, Teacher* lecturer);
void SetMark(Subject& subject, int mark);

Subject* MakeSubject(string title, int hours, int mark, Teacher* lecturer);
Subject* CopySubject(Subject& subject);
```

Сеттер для лектора в данном случае будет принимать указатель на уже созданный объект. Возможен вариант с передачей в сеттер SetLecturer отдельных значений:

```
void SetLecturer(Subject& subject, string surname, string initials, string position);
```

Однако такой вариант имеет ряд недостатков: приводит к созданию новых объектов лектора при каждом вызове сеттера – нельзя присвоить одного и того же лектора двум дисциплинам; требует сложной поддержки при добавлении новых полей в структуру преподавателя; навязывает способ выделения памяти под объекты

преподавателя – если внутри сеттера память под объект преподавателя будет выделяться динамически, то поместить в поле статический объект будет уже невозможно. По этим и другим причинам предпочтительнее реализовывать сеттер с передачей указателя на готовый объект. Аналогично в функцию-конструктор предпочтительнее передавать указатель на готовый объект lecturer вместо передачи отдельных данных.

Агрегирование "один ко многим"

Агрегирующая структура может агрегировать не один объект другой структуры, а сразу несколько. Расширяя пример с дисциплиной, мы можем добавить дополнительное поле типа Teacher для хранения информации о преподавателе-практике:

```
struct Subject
{
    string Title;
    int Hours;
    Teacher* Lecturer;
    Teacher* Assistant;
    int Mark;
};
```

В данном примере дисциплина хранит информацию о двух преподавателях. Стоит отметить, что в указателях может храниться адрес одного и того же объекта, что будет означать, что лекции и практики преподает один человек. Или значение nullptr (нулевой указатель – константа, фактически указывающая, что в указателе не хранится объект) может указывать, что преподаватели не назначены.

Мы можем создать в структуре три, четыре, любое количество агрегируемых полей. При большом количестве агрегируемых объектов удобно использовать массивы. Например, опишем структуры студента и группы, хранящей массив студентов:

```
struct Student
{
    string Surname;   // Фамилия
    string Name;      // Имя
    int Id;           // Номер зачетной книжки
};

struct Group
{
    string Number;         // Номер группы
    int EntranceYear;      // Год поступления
    Student Students[30]; // Массив из 30 студентов
};
```

Как правило, в вузах есть ограничение на максимальное количество студентов в одной группе. Если это количество составляет, например, 30 студентов, то мы можем отразить данное ограничение при объявлении поля структуры. Создав поле `Student Students[30]` мы гарантируем, что в группе нельзя будет хранить больше положенного количества студентов.

Возможно, вы обратили внимание на то, что номер группы представлен типом данных string. Это сделано намеренно, так как в вузах может встречаться применение букв в номерах групп ("з" – заочное, "м" – магистры и т.д.).

Работа с массивом в качестве поля не будет отличаться от работы с обычным массивом. Далее показан пример кода, инициализирующий группу с вводом информации о студентах с клавиатуры:

```
Group group;
group.Number = "517";
group.EntranceYear = 2007;
```

```
for (int i = 0; i < 30; i++)
{
    cin >> group.Students[i].Surname;
    cin >> group.Students[i].Name;
    cin >> group.Students[i].Id;
}
```

В следующем примере показан поиск студента с заданным номером зачетной книжки в группе и выводом его на экран:

```
int findedId;
int findedStudent = -1;
cout << "Введите номер зачетной книжки для поиска студента: ";
cin >> findedId;
for (int i = 0; i < 30; i++)
{
    if (group.Students[i].Id == findedId)
    {
        findedStudent = i;
        break;
    }
}

if (findedStudent != -1)
{
    cout << "Найден студент:" << endl;
    cout << group.Students[findedStudent].Surname << " ";
    cout << group.Students[findedStudent].Name << " ";
    cout << group.Students[findedStudent].Id << endl;
}
```

Данная реализация имеет недостаток: в группе может быть меньше 30 человек. Этот недостаток легко исправить, добавив в структуру Group поле int StudentsCount, хранящее реальное количество студентов в группе. Читателям предлагается добавить данное поле самостоятельно. Конечно же, при работе с такой структурой разработчик должен самостоятельно контролировать правильность числа StudentsCount в случаях добавления и удаления студентов из массива.

Реализация структуры Group с полем-массивом предполагает автоматическое выделение памяти под 30 студентов, даже в тех случаях, когда реальное количество студентов в группе меньше. В большинстве случаев, при агрегировании структур нельзя заранее сказать, какое именно количество объектов нужно хранить: количество студентов в группе, количество товаров в магазине, количество графических примитивов в 3D-модели, количество пользователей в веб-приложении. Поэтому создание полей-массивов с фиксированным количеством является неэффективным решением. Более грамотным решением будет хранение массива объектов по указателю:

```
struct Group
{
    string Number;         // Номер группы
    int EntranceYear;      // Год поступления
    Student* Students;     // Указатель на массив студентов
    int StudentsCount;     // Количество студентов в массиве
};
```

Данная реализация даёт больше свободы при разработке, однако требует и большей внимательности. Замена поля-массива на указатель означает, что при создании экземпляра группы сам массив создаваться не будет – его нужно будет создать отдельно и присвоить его адрес в указатель:

```
void GroupExample()
{
```

```
    Group group;
    group.Number = "517";
    group.EntranceYear = 2007;

    group.StudentsCount = 17;
    group.Students = new Student[group.StudentsCount];

    //...

    delete[] group.Students;
}
```

В примере выше в поле-указатель помещается адрес динамически созданного массива. Размер массива может быть произвольным. Например, браться из константы с ограничением в 30 студентов, или считываться с клавиатуры – в зависимости от решаемой задачи.

Разумеется, при создании динамических массивов в качестве полей структуры следует помнить об освобождении динамической памяти. В противном случае, в программе появятся утечки памяти, что приведет к неправильной работе компьютера.

Примечание: как было сказано ранее, в большинстве случаев разработчик не может заранее предугадать точное количество объектов, которое нужно хранить в поле. Однако на практике для данной цели используют не массивы и указатели, а хранение объектов в списках и других так называемых контейнерах. Вы можете ознакомиться с данной темой самостоятельно и применять их в своих программах.

Таким образом, разработчик может организовать агрегирование "один к одному", "один к фиксированному количеству", "один ко многим".

Агрегация и композиция

В программировании принято разделять агрегирование на два понятия – ***агрегация*** и ***композиция***, чтобы точнее описывать взаимодействие двух объектов. Агрегация и композиция являются подтипами агрегирования.

Композиция – связь между двумя объектами "часть-целое", при котором время жизни объекта-"части" совпадает со временем жизни объекта-"целого". ***Агрегация*** – связь между двумя объектами "часть-целое", при котором время жизни объекта-"части" отличается от времени жизни объекта-"целого".

Что это означает? Связь композиция предполагает более жесткую, строгую связь между агрегируемой и агрегирующей структурами, при которой создание агрегирующего объекта предполагает создание агрегируемого объекта, а уничтожение агрегирующего объекта приводит к уничтожению агрегируемого объекта. В реальной жизни примером такой связи может быть дом и его стены. Стены являются обязательной частью дома. Для создания дома мы должны создать и его стены, а для разрушения дома мы должны разрушить стены. Другой пример – человек и его мозг. Мозг неотделим от человека, является его обязательной частью. В данном случае можно сказать, что человек ***композирует*** мозг.

Связь агрегации менее строгая. Она подразумевает, что при создании агрегирующего объекта необязательно создавать агрегируемый объект. Он может быть создан и помещен в поле агрегирующего объекта значительно позже и не является обязательным для работы агрегирующего объекта. Также при уничтожении агрегирующего объекта необязательно уничтожение агрегируемого объекта и наоборот. В реальной жизни примером агрегации может служить комната и мебель в ней. Комната может хранить в себе мебель, но мебель не является обязательной частью комнаты. При создании комнаты необязательно сразу же заносить в неё мебель, в любое время мебель может быть вынесена из одной комнаты в другую комнату, при уничтожении комнаты необязательно уничтожать мебель, а при уничтожении мебели необязательно уничтожать комнату. В данном случае можно сказать, что комната ***агрегирует*** мебель.

Еще один пример: пользователь соцсети, его личные данные и его подписчики. При удалении пользователя, его личные данные (фотографии, посты, переписка) должны быть удалены, но его подписчики (другие пользователи) вряд ли обрадуются, если вместе с пользователем удалятся и их аккаунты. Поэтому пользователь композирует личные данные и агрегирует список подписчиков.

Понятия агрегации и композиции не привязаны к конкретным синтаксическим элементам языка Си++. То есть, если в программе один объект является полем другого объекта, мы не можем точно определить тип агрегирования. Для этого необходимо посмотреть, как именно создаются объекты обоих структур и как они уничтожаются. Поэтому говоря об агрегации или композиции, в первую очередь следует отталкиваться от того, как описанные объекты взаимодействуют в реальном мире с точки зрения решаемой задачи.

В разных программах одни и те же объекты могут быть связаны либо агрегацией, либо композицией. Например, в программе для учета товаров магазина мобильных телефонов, мобильный телефон будет строго композировать один процессор. Ведь покупатель не захочет покупать телефон без процессора. Однако в программе для учета комплектующих сервисного центра по ремонту мобильных телефонов, мобильный телефон может агрегировать процессор – так как процессор отдельная комплектующая, он может быть установлен из одного телефона в другой. Таким образом, в разных задачах связь между объектами может быть различна.

Более того, в разных контекстах связь между объектами можно толковать по-разному. Например, человек владеет квартирой или квартира имеет владельца? Связь агрегирования можно истолковать в любом направлении. Вопрос софистический, но на практике ответ определяется удобством дальнейшей обработки данных. Например, в информационной системе налоговой службы определяющим объектом для обработки является человек, который хранит информацию о своём имуществе. В такой системе правильнее реализовать агрегацию имущества человеком. А для коммунальных служб, занимающихся обслуживанием квартир, удобнее хранить первичную информацию по квартирам, в которых есть ссылка на данные о её владельце, т.е. квартира агрегирует человека. Помните, что правильно определенная архитектура напрямую влияет как на удобство работы с данными, так и на скорость вычислительных алгоритмов.

Архитектура приложения и введение в UML-диаграммы классов

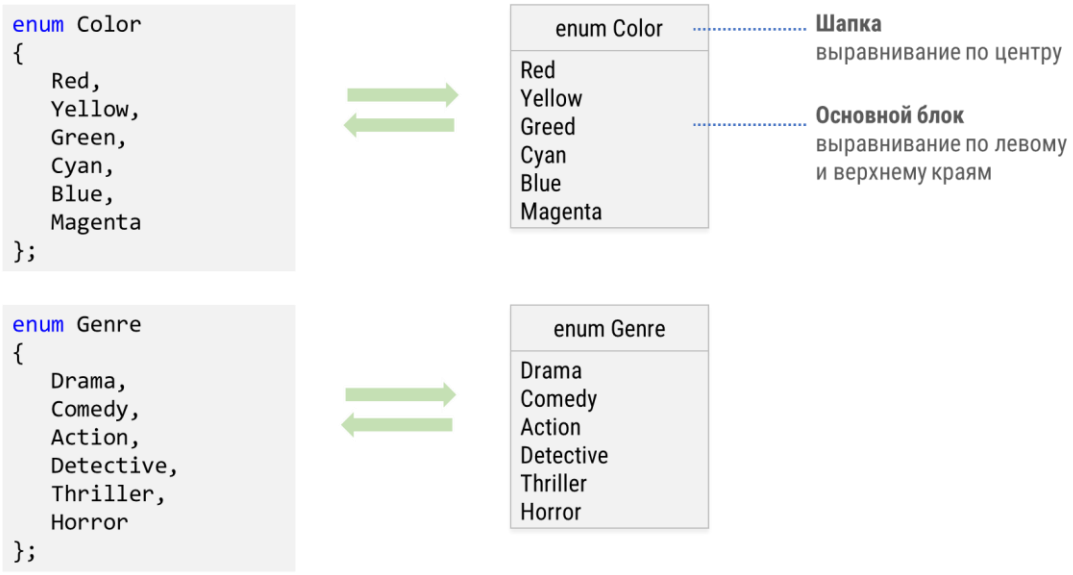
Ранее мы рассмотрели варианты агрегирования дисциплины и преподавателя, группы и студентов. Но агрегирующая структура может быть агрегируемой по отношению к другой структуре. Например, структура Group агрегирует массив структур Student, а структура Faculty агрегирует массив структур Group. Появляется иерархия взаимодействующих структур и функций, работающих с ними. В данном случае можно говорить об архитектуре программы.

При разработке программы очень важно создать правильную архитектуру. С одной стороны, архитектура должна описывать связи реальных объектов предметной области. В противном случае разработчику будет сложно разобраться в программе. Например, если у вас в программе будут такие абсурдные связи, когда кошка композирует машину, а в квадрате храниться массив кругов – чтение такого кода будет затруднительно.

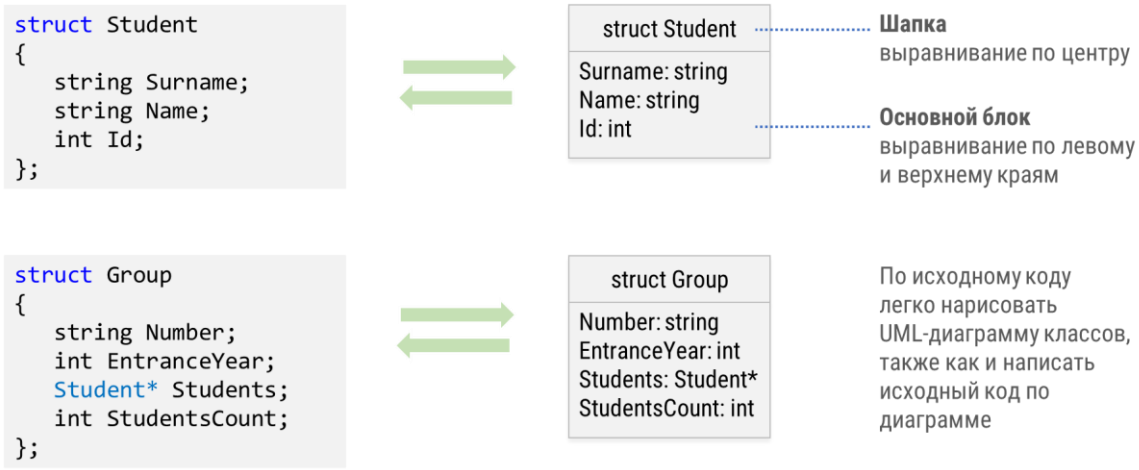
С другой стороны, архитектура должна быть достаточно гибкой для возможности её расширения новыми функциями и структурами. В данном курсе не будут рассматриваться вопросы грамотного проектирования приложений, но будут рассмотрены некоторые инструменты для описания архитектуры, её анализа и проектирования.

Исходный код программы не способен выразительно представить архитектуру, поэтому разработчики используют специальные диаграммы для визуализации архитектуры. Наиболее распространенным видом диаграмм является диаграммы классов нотации UML. UML (unified modeling language) – это графический язык моделирования, позволяющий описывать бизнес-процессы, объектно-ориентированные приложения и организационные структуры. Всего UML содержит 12 видов диаграмм, одним из которых являются диаграммы классов. UML-диаграммы классов описывают такие пользовательские типы данных, как классы, структуры и перечисления, а также связи между ними. Фактически, диаграммы классов представляют архитектуру программы в лаконичной графической форме, удобной для анализа и проектирования. В ходе последующих разделов будут изучены все ключевые элементы диаграмм классов. В этом же разделе будут рассмотрены отображение на диаграммах структур, перечислений и связей агрегации и композиции.

Перечисление на диаграммах классов обозначается в виде прямоугольника, разделенного на две части, где в верхней части – шапке – пишется ключевое слово `enum` и название перечисления, а в нижней части – основной – последовательно перечисляются все допустимые значения перечисления, каждое значение на новой строке:



Структура на диаграммах классов обозначается в виде прямоугольника, разделенного на две части, где в шапке пишется ключевое слово `struct` и название структуры, а в основной части перечисляются все поля структуры в формате `Name: Type`, где `Name` – это имя поля, а `Type` это тип данных:

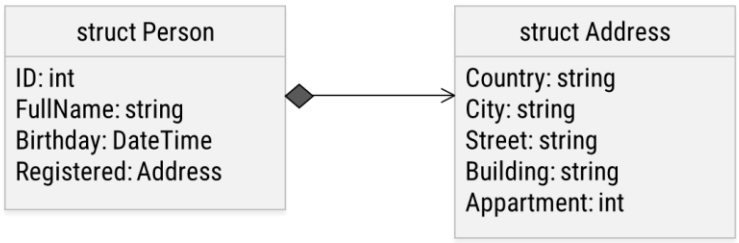


По графическому представлению можно однозначно восстановить исходный код структуры или перечисления. Более того, графическое представление UML опускает синтаксические особенности языков программирования, а потому является универсальным инструментом для описания приложений среди разработчиков разных языков программирования.

После отрисовки всех структур и перечислений, существующих в программе, необходимо отобразить существующие связи.

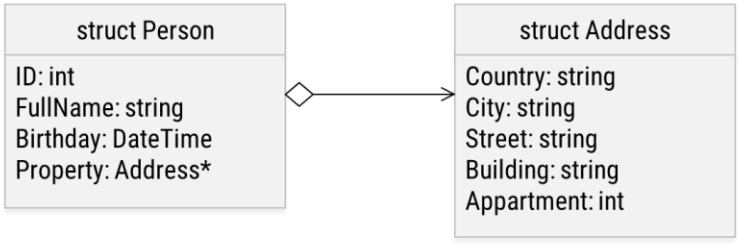
Связь композиции рисуется в виде направленной незакрытой стрелки с закрашенным ромбом в её начале. Связь агрегации рисуется в виде направленной незакрытой стрелки с незакрашенным ромбом в её начале. Ромб рисуется со стороны агрегирующего объекта, стрелка направлена в сторону агрегируемого объекта, т.е. стрелка идет от объекта-"целого" к объекту-"части":

Человек композирует адрес регистрации (Registered)
(Объект адреса является полем человека)



Композиция между человеком и адресом показывает, что в программе адрес регистрации это обязательные данные человека

Человек агрегирует имущество (Property)
(Объект адреса является полем человека)

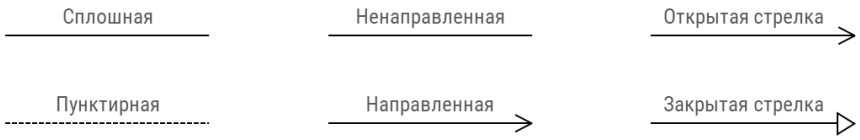


Агрегация между человеком и адресом показывает, что в программе имущество это необязательные данные человека

Хранение объекта по указателю не является обязательным признаком агрегации.
При композиции поля также могут представлять указатели

Обратите внимание, что стрелка должна быть незакрытой, так как вид стрелки (закрытая/незакрытая), вид линии (сплошная/пунктирная), тип фигуры (закрашенный/не закрашенный ромб) – всё это имеет значение. Неправильное обозначение может привести к неправильному пониманию архитектуры другими разработчиками.

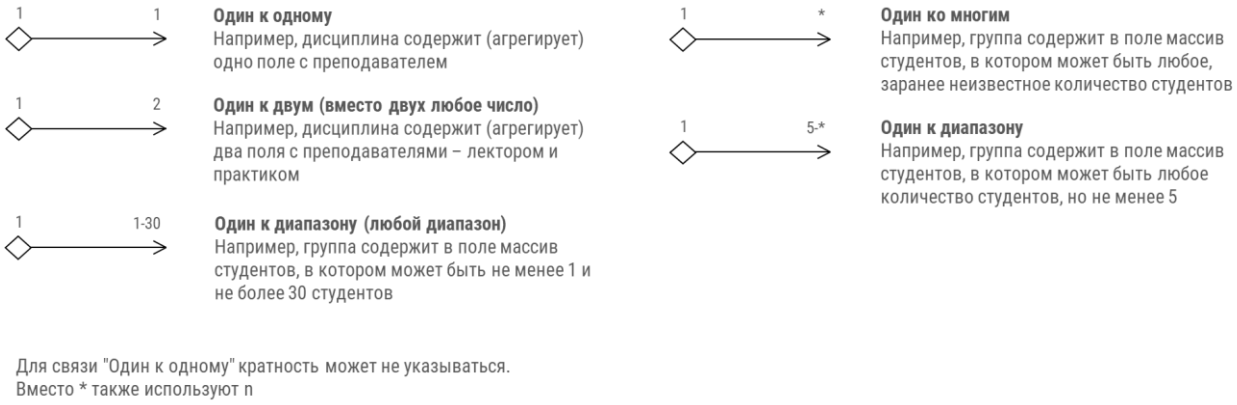
Примеры стилей линий в UML



Показаны не все стили.
Представленные стили могут комбинироваться.
Каждая комбинация имеет своё значение

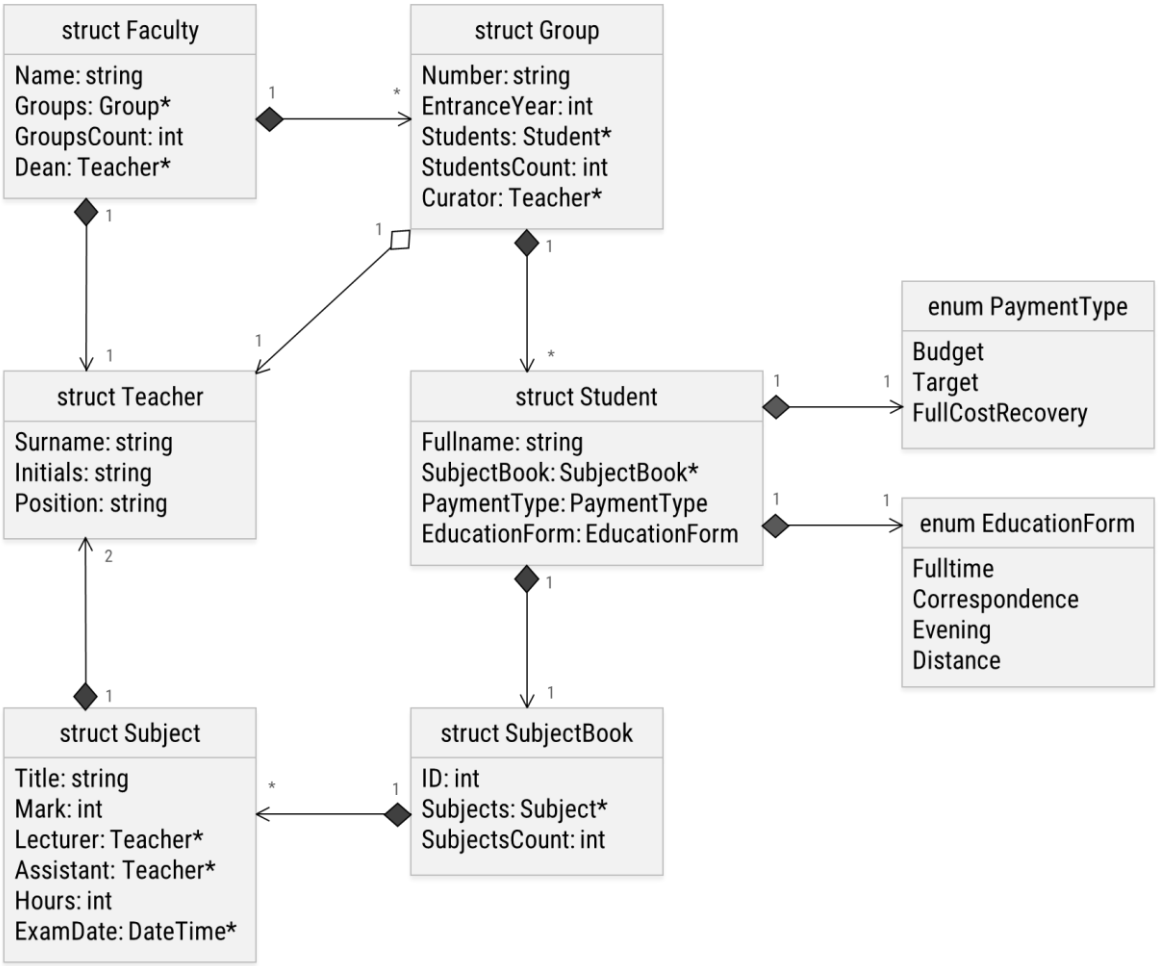
При необходимости стрелки могут изгибаться, пересекаться или иметь точки перелома – в тех случаях, когда нельзя напрямую отрисовать связи на диаграмме.

Обязательным элементом связей агрегации и композиции является кратность. Кратность (в некоторых источниках также называется мощностью или кардинальностью) показывает количественное соотношение объектов между собой, например, "один к одному", "один ко многим", "один к двум" и т.д. На диаграмме кратность обозначается в виде двух чисел, указанных на концах стрелки:



Символ `*` обозначает неопределенное количество. Таким образом, указывается связь "один ко многим". В качестве значений также могут указываться диапазоны. Например, если мы знаем, что агрегируемых объектов может быть не менее 2, но не более 10, можно явно указать диапазон `2 - 10` у конца стрелки.

Теперь попробуем описать с помощью UML-диаграммы классов архитектуру небольшой информационной системы вуза. Ранее мы уже создали структуры дисциплины, группы, студента, преподавателя. Давайте добавим структуры факультета и зачетной книжки, а также несколько перечислений, описывающих форму обучения студента и форму оплаты:



В отличие от исходного кода, где структуры располагаются последовательно или в разных файлах, диаграмма наглядно визуализирует все типы данных на одной схеме. Наглядность достигается за счет взаимного расположения структур и обозначения связей, чего нельзя достичь с помощью исходного кода. Стоит прокомментировать, что большинство связей на представленной диаграмме могут быть реализованы и как агрегации, за исключением связей композиции между студентом и перечислениями формы оплаты и формы обучения – в подавляющем большинстве случаев структуры композируют перечисления.

Важную роль играет расположение структур на диаграмме. Чтобы диаграмма была комфортной для понимания, нужно:

1) Располагать типы данных таким образом, чтобы уменьшить количество пересечений и изломов стрелок-связей. Чем ближе располагаются связанные типы данных, тем проще воспринимать диаграмму.
2) Располагать типы данных приблизительно на одинаковом расстоянии друг от друга. Расположение структур должно чем-то напоминать таблицу, где в каждой ячейке своя структура, а между всеми соседними структурами одинаковое расстояние, достаточное для отображения связи между ними.
3) Располагать типы данных по принципу "сверху вниз" или "слева направо", где сверху находятся более высокоуровневые структуры (агрегирующие), а внизу низкоуровневые (агрегируемые). Вертикальная иерархия привычна для человека и сразу показывает значимость структуры по отношению к остальным типам данных.
4) Выравнивать типы данных по размеру и по границам (выравнивание по верхней границе, выравнивание по правой границе и т.д.). Если вы составляете диаграмму в специальном редакторе, а не рисуете от руки, позаботьтесь о выравнивании элементов друг относительно друга. Выровненные элементы воспринимаются человеком проще, не выровненные элементы, наоборот, создают ощущение небрежности, ненадежности архитектуры и мешают восприятию диаграммы.
5) Располагать на одной диаграмме не более 15 структур и перечислений. Реальные приложения могут насчитывать сотни и даже тысячи структур и перечислений. Не надо стараться отобразить их все на одной диаграмме. Если архитектура достаточно сложная, разделите её на несколько диаграмм. Главная цель – это отобразить архитектуру для её комфортного анализа и проектирования, а более 15 элементов на одной диаграмме воспринимаются тяжело. Если разделение диаграммы на несколько схем сделает понимание архитектуры проще, то стоит это сделать.

Соблюдая эти правила, у вас получится создавать наглядные и понятные диаграммы классов.

Функции-сеттеры

В прошлой лабораторной работе для работы со структурами уже рассматривались специальные функции – функции-конструкторы и функции копирования. В данном разделе рассмотрим еще один вид функций – функции-сеттеры. Функция-сеттер (от англ. set – "установить") – функция, позволяющая присваивать в поля только корректные значения.

Рассмотрим следующий код:

```
struct Phone
{
   string Model;
   string Company;
   int MemoryGb;
   double ScreenSize;
   double Cost;
}

int main()
{
   Phone phone;
   phone.MemoryGb = 32;
   ...
   phone.MemoryGb = -64;
}
```

В коде объявляется структура Phone, хранящая данные о телефонах для интернет-магазина смартфонов. Структура хранит данные о названии модели, компании-производителе, памяти телефона, размере экрана и стоимости. Например, для хранения размера доступной памяти используется тип данных int (поле MemoryGb), указывающее количество Гб. Предметная область предполагает, что в смартфонах может быть только положительное количество Гб памяти. В телефоне не может быть отрицательного объема памяти. Однако, создав переменную типа Phone, разработчик может присвоить в поле MemoryGb отрицательное значение.

Неправильное значение может попасть в поле по разным причинам: невнимательность разработчика, ввод пользователя, ошибка при чтении значения из файла или базы данных. Вне зависимости от причины, компилятор присвоит некорректное значение в поле. Компилятор не может проверить, допустимо ли значение с точки зрения предметной области, проверяется только совпадение типов данных слева и справа от знака присваивания. Некорректные данные в системе могут привести к неправильной работе программы в дальнейшем.

Для решения данной проблемы создадим функцию-сеттер:

```
struct Phone
{
   string Model;
   string Company;
   int MemoryGb;
   double ScreenSize;
   double Cost;
}

void SetMemoryGb(Phone& phone, int memoryGb)
{
   if (memoryGb < 1)
   {
      throw exception(
       "Память не может быть отрицательной");
   }
   phone.MemoryGb = memoryGb;
}

int main()
{
   Phone phone;
   SetMemoryGb(phone, 32);
   SetMemoryGb(phone, -64);
}
```

Функция-сеттер SetMemoryGb() принимает на вход ссылку на объект телефона и значение, которое нужно поместить в поле телефона. Внутри функции происходит проверка правильности значения (например, чтобы оно было не меньше 1). Если значение не проходит проверку, функция генерирует исключение, что при отсутствии обработки приведет к завершению программы. Если же значение корректное, оно будет присвоено в поле объекта.

Таким образом, присвоение значений в поля должно осуществляться через функции-сеттеры (см. функцию main в примере выше). Такая форма записи менее удобна, чем использование оператора присваивания. Однако применение функций-сеттеров обеспечивает проверку правильности значений и прерывание программы в случае некорректных данных. Прерывание программы во время выполнения позволит быстрее обнаружить ошибку в коде и сделать программу безопаснее для пользователя.

Сеттеры пишутся для каждого поля каждой структуры, даже если для поля на данный момент нет ограничений. Для структуры Phone разработчику понадобится написать сеттеры:

```
void SetMemoryGb(Phone& phone, int memoryGb) {...}
void SetScreenSize(Phone& phone, int memoryGb) {...}
void SetCost(Phone& phone, double cost) {...}
void SetModel(Phone& phone, string model) {...}
void SetCompany(Phone& phone, string company) {...}
```

Приходится писать много маленьких функций, однако это повышает надежность программы.

Несмотря на то, что название модели телефона в предметной области ничем не ограничивается и может быть любым, разработчику всё равно следует написать функцию-сеттер. Сеттер без проверки будет выглядеть следующим образом:

```
void SetModel(Phone& phone, string model)
{
   phone.Model = model;
}
```

Это позволит в будущем быстро добавлять ограничения на название модели, если они появятся в системе. В противном случае, сеттер будет написан позже, и вам как разработчику придется также исправлять все участки кода, где название модели присваивается напрямую в поле.

Классы

Структуры – мощный инструмент, позволяющий создавать более читаемый и поддерживаемый код. Обязательными функциями, которые следует создавать совместно со структурами, это функции-сеттеры и функции-конструкторы. Однако, даже при правильном создании структур и сопутствующих функций, при их использовании очень легко допустить ошибки.

Рассмотрим следующую ситуацию. Вы участник команды разработчиков, и вам поручили разработку ряда структур и сопроводительных функций. Эти структуры в дальнейшем будут использоваться другим разработчиком в вашей команде. Например, вам поручено разработать ранее описанную структуру Phone:

```
struct Phone
{
   string Model;
   string Company;
   int MemoryGb;
   double ScreenSize;
   double Cost;
}

void SetMemoryGb(Phone& phone, int memoryGb)
{
   if (memoryGb < 1)
   {
      throw exception(
       "Память не может быть отрицательной");
   }
   phone.MemoryGb = memoryGb;
}
```

Будучи ответственным разработчиком, вы создаете структуры, функции-сеттеры для всех полей структуры, функции-конструкторы. Для каждого поля и каждой функции вы оставляете комментарии в документации, поясняющие ограничения на вводимые значения и ошибки в случае ввода некорректных данных. По завершению разработки вы передаете ваш код (или скомпилированные dll) коллеге-разработчику, который будет разрабатывать более высокоуровневую функциональность. Однако, несмотря на проделанную вами работу, ваш коллега начинает использовать структуры в обход контроля допустимых значений. Фактически, он присваивает значения в поля структуры напрямую вместо использования функций-сеттеров:

```
Phone* ReadPhone()
{
   Phone* phone = new Phone();

   cin >> phone->Model;
   cin >> phone->Company;
   cin >> phone->MemoryGb;
```

```
    return phone;
}
```

Причины этой ситуации могут быть разные: неопытность вашего коллеги, ожидание, что пользователь будет внимательным при вводе и не допустит ошибки, или банальная лень. Однако, проблема заключается в том, что даже при создании всех необходимых функций-сеттеров и функций-конструкторов, язык программирования позволяет их обойти. Для решения этой (и многих других) проблемы были придуманы классы.

Класс – это пользовательский составной тип данных, объединяющий в едином описании несколько именованных переменных (полей) и функции по обработке этих переменных. Функции класса называются ***методами***. Фактически, класс – это более расширенный вариант структур.

Перепишем представленный выше пример с использованием классов. Во-первых, при создании классов используется ключевое слово class вместо слова struct. Во-вторых в начале раскрывающихся скобок следует написать ключевое слово public с двоеточием (назначение этого слова будет показано чуть позже):

```
class Phone
{
public:
    string Model;
    string Company;
    int MemoryGb;
    double ScreenSize;
    double Cost;
}
```

Формат объявления полей остается прежним, как и в структурах. В таком варианте данный класс не отличается от первоначальной структуры Phone. Теперь поместим функцию-сеттер SetMemoryGb() внутрь класса:

```
class Phone
{
public:
    string Model;
    string Company;
    int MemoryGb;
    double ScreenSize;
    double Cost;

    void SetMemoryGb(int memoryGb)
    {
        if (memoryGb < 1)
        {
            throw exception(
             "Память не может быть меньше 1 Гб");
        }
        this->MemoryGb = memoryGb;
    }
}
```

Обратите внимание на изменение сигнатуры. Из передаваемых параметров исключена ссылка на Phone, передается только значение memoryGb, которые нужно присвоить в поле класса. А обращение к полю внутри функции происходит через ключевое слово this. this – это указатель на объект класса, от лица которого вызывается данная функция. Чтобы разобраться в этом, рассмотрим, как изменится вызов функции SetMemoryGb():

С использованием старой структуры Phone	**С использованием нового класса Phone**

```
int main()
{
   Phone phone1;
   SetMemoryGb(phone1, 32);

   Phone phone2;
   SetMemoryGb(phone2, 64);
}
```

```
int main()
{
   Phone phone1;
   phone1.SetMemoryGb(32);

   Phone phone2;
   phone2.SetMemoryGb(64);
}
```

Так как функция SetMemoryGb() находится внутри класса Phone, её вызов осуществляется от лица переменной класса, через точку или стрелку, как и полям. При этом вызывая функцию SetMemory() для переменной phone1, в качестве указателя this будет храниться адрес переменной phone1. А вызывая функцию SetMemory() для переменной phone2, в качестве указателя this внутри функции будет храниться адрес переменной phone2. Таким образом, функция-сеттер неразрывно связана с объектом. По аналогии, в класс Phone должны быть перенесены и остальные сеттеры. Указатель this будет доступен всем функциям, которые находятся внутри класса. Или, другими словами, ***указатель this будет доступен всем методам класса***.

Однако, данный пример не завершен. Следующий шаг – сокрытие полей. Для этого перед полями следует написать слово private, а написать слово public перед методами-сеттерами, как в примере ниже:

```
class Phone
{
private:
   string Model;
   string Company;
   int MemoryGb;
   double ScreenSize;
   double Cost;

public:
   void SetModel(string model) { ... };
   void SetCompany(string company) { ... };
   void SetMemoryGb(int memoryGb) { ... };
   void SetScreenSize(double screenSize) { ... };
   void SetCost(double cost) { ... };
}
```

Ключевые слова private и public называются ***модификаторами доступа***. Они определяют доступность членов класса (полей и методов). После ключевого слова private перечисляются те поля и методы, которые могут быть доступны только внутри класса. После ключевого слова public перечисляются те поля и методы, которые могут быть доступны вне класса. Так, поместив поле MemoryGb под модификатор private, клиентский код (код, где используется класс Phone), не сможет напрямую обращаться к полю MemoryGb – это приведет к ошибке компиляции:

```
int main()
{
   Phone phone;

   // Ошибка компиляции – нельзя обращаться к закрытым (private) полям вне класса
   phone.MemoryGb = 32;

   // Открытый (public) сеттер – единственный способ присвоить значение в поле MemoryGb
   phone.SetMemoryGb(32);
}
```

Модификаторы доступа позволяют создать внутри класса уникальную область видимости, ограничивающую доступ к членам класса. Таким образом, с помощью классов и модификаторов доступа можно обязать

клиентский код (и других разработчиков) использовать сеттер. Всё это повышает качество программ и уменьшает количество ошибок. В отличие от структур, ***классы позволяют уменьшить количество ошибок в клиентском коде***. То есть, написав правильный и надежный класс, вы уменьшаете количество ошибок в коде других разработчиков!

Однако, при закрытых полях класса, в клиентском коде нельзя не только присвоить значения в поля напрямую, но получить текущее значение полей. Например, следующий код также вызовет ошибку компиляции:

```
Phone phone;
cout << phone.MemoryGb; // Нельзя обращаться к закрытому полю напрямую
```

Для того, чтобы получать текущие значения полей класса, но при этом не нарушая их защищенность от неправильного ввода, в классы необходимы добавить методы-геттеры. Геттер (от англ. get – "получить") – метод, возвращающий значение поля класса. Геттеры – противоположность сеттерам, устанавливающим значения в поля. Реализация геттера для MemoryGb будет выглядеть следующим образом:

```
class Phone
{
private:
   string Model;
   string Company;
   int MemoryGb;
   double ScreenSize;
   double Cost;

public:
   int GetMemoryGb()
   {
      return this->MemoryGb;
   }
}
```

- Геттер не имеет входных аргументов;
- его возвращаемый тип данных совпадает с типом данных поля, для которого он реализован;
- его принято именовать словом Get и именем поля, для которого он реализован;
- в отличие от сеттеров, в нём не надо реализовывать проверки правильности, так как он возвращает заведомо корректное значение поля (при правильной реализации сеттера).

В большинстве случаев, реализация геттера состоит из одной строки с оператором return и указанием возвращаемого поля. Аналогично сеттерам, геттеры реализуются для всех полей, к значениям которых нужно предоставить доступ в клиентском коде.

Полный код класса Phone с реализацией всех сеттеров и геттеров будет выглядеть следующим образом:

```
class Phone
{
private:
    string Model;
    string Company;
    int MemoryGb;
    double ScreenSize;
    double Cost;

public:
    string GetModel()
    {
           return this->Model;
    }
```

```
    string GetCompany()
    {
        return this->Company;
    }

    int GetMemoryGb()
    {
        return this->MemoryGb;
    }

    double GetScreenSize()
    {
        return this->ScreenSize;
    }

    double GetCost()
    {
        return this->Cost;
    }

    void SetModel(string model)
    {
        this->Model = model;
    }

    void SetCompany(string company)
    {
        this->Company = company;
    }

    void SetMemoryGb(int memoryGb)
    {
        if (memoryGb < 1)
        {
            throw exception("Память должна быть не меньше 1 Гб");
        }
        this->MemoryGb = memoryGb;
    }

    void SetScreenSize(double screenSize)
    {
        if (screenSize < 0.0)
        {
            throw exception("Размер экрана должен быть положительным");
        }
        this->ScreenSize = screenSize;
    }

    void SetCost(double cost)
    {
        if (cost < 0.0)
        {
            throw exception("Стоимость должна быть положительной");
        }
        this->Cost = cost;
    }
}
```

Как можно заметить, написание классов требует написания большого количества методов – два метода (геттер и сеттер) на каждое поле. Всё это необходимая плата для повышения надежности программы и ускорения разработки. Ведь любые ошибки рано или поздно будут обнаружены – тестировщиками или самими пользователями – и их понадобится исправлять. По статистике исправление ошибок может стать в разы затратнее как по временным, так и по финансовым ресурсам, чем первичное написание защищенного кода. По этой причине, лучше потратить немного времени на написание защищенного класса, чем тратить время и нервы на постоянном исправлении ошибок в будущем.

Примечание: с точки зрения Си++, внутри структур также можно объявлять методы и использовать модификаторы доступа. Разница между структурами и классами в Си++ заключается в том, что внутри структуры по умолчанию используется модификатор доступа public, а в классах - private. Однако с точки зрения теории ООП принято различать структуры и классы – где структуры предназначены для хранения данных, а классы имеют как поля, так и методы.

В других языках программирования (например, C#) может быть иное толкование структур и классов, и отличия могут быть более существенными.

Конструкторы класса

По примеру геттеров и сеттеров, внутрь класса можно поместить и функции-конструкторы, навязав клиентскому коду способ создания объектов класса. Функции-конструкторы класса можно называть просто ***конструкторы***.

Конструкторы, являясь методами класса, имеют особый синтаксис. Например, в прошлом лабораторной задании требовалось создать структуру Movie и функцию-конструктор MakeMovie():

```
// Структура Фильм
struct Movie
{
   string Title;
   string Genre;
   int Year;
   double Rate;
};

// Функция-конструктор для структуры Фильм
Movie* MakeMovie(string& title, string& genre, int year, double rate)
{
   Movie* movie = new Movie();
   movie->Title = title;
   movie->Genre = genre;
   movie->Year = year;
   movie->Rate = rate;
   return movie;
}

// Функция, демонстрирующая вызов конструктора в клиентском коде.
void DemoMovie()
{
   Movie* movie1 = MakeMovie("Крепкий орешек", "Боевик", 1988, 8.0);
   Movie* movie2 = MakeMovie("Побег из Шоушенка", "Драма", 1994, 9.1);
   Movie* movie3 = MakeMovie("1+1", "Комедия, драма", 2012, 8.8);
   ...
}
```

Перепишем структуру Movie в класс:

```
class Movie
{
```

```
private:
   string _title;
   string _genre;
   int _year;
   double _rate;

public:
   string GetTitle() {...};
   string GetGenre() {...};
   int GetYear() {...};
   double GetRate() {...};

   void SetTitle(string& title) {...};
   void SetGenre(string& genre) {...};
   void SetYear(int year) {...};
   void SetRate(double rate) {...};
}
```

И перепишем функцию-конструктор с использованием сеттеров:

```
// Функция-конструктор для структуры Фильм
Movie* MakeMovie(string& title, string& genre, int year, double rate)
{
   Movie* movie = new Movie();
   movie->SetTitle(title);
   movie->SetGenre(genre);
   movie->SetYear(year);
   movie->SetRate(rate);
   return movie;
}
```

Вызов функции-конструктора при этом не поменялся. Теперь перенесем конструктор в класс:

```
class Movie
{
private:
   string _title;
   string _genre;
   int _year;
   double _rate;

public:
   string GetTitle() {...};
   string GetGenre() {...};
   int GetYear() {...};
   double GetRate() {...};

   void SetTitle(string& title) {...};
   void SetGenre(string& genre) {...};
   void SetYear(int year) {...};
   void SetRate(double rate) {...};

   Movie(string& title, string& genre, int year, double rate)
   {
      this->SetTitle(title);
      this->SetGenre(genre);
      this->SetYear(year);
      this->SetRate(rate);
   }
```

```
}
```

Обратите внимание на изменение синтаксиса:

1) конструкторы класса не имеют возвращаемого значения (даже void), а название конструктора должно обязательно совпадать с именем класса с точностью до регистра всех букв;
2) внутри самого конструктора не происходит выделения динамической памяти под объект класса – эта память выделяется автоматически при вызове конструктора и становится доступна через указатель this;
3) так как конструкторы классы не имеют возвращаемого значения, оператор return также не нужен в конструкторе.

Добавив конструктор в класс, мы обязали клиентский код использовать только данный способ создания объектов класса Movie. Рассмотрим, как изменился клиентский код:

```
void DemoMovie()
{
   // Создание переменной в статической памяти
   Movie movie("Крепкий орешек", "Боевик", 1988, 8.0);

   // Создание переменной в динамической памяти
   Movie* movie2 = new Movie("Побег из Шоушенка", "Драма", 1994, 9.1);

   // Ошибка компиляции – создать объект по-старому не получится
   Movie* movie3 = new Movie();
}
```

Первое очевидное преимущество – в отличие от функций-конструкторов, конструкторы класса позволяют создавать переменные как в статической, так и в динамической памяти. Второе преимущество – невозможность создать объект с пустыми полями или полями с неправильными значениями (как мы помним, внутри конструктора вызывается сеттеры, выполняющие проверки значений). Третье преимущество – если в классе добавится новое поле, то, вероятно, изменится и конструктор. А это приведет к необходимости изменения всех точек вызова конструктора – то есть мы обязываем клиентский код исправлять вызов конструктора под новые требования. В противном случае, компилятор не сможет скомпилировать программу из-за недостатка входных аргументов конструктора.

Строка `Movie("Побег из Шоушенка", "Драма", 1994, 9.1)` и есть вызов нашего конструктора, который, в зависимости от контекста, будет создавать объект в статической или динамической памяти.

Конструкторов в классе может быть несколько. Главное требование – все конструкторы должны иметь разные наборы входных аргументов. Например, для класса Movie можно создать еще один конструктор, без указания рейтинга – так как рейтинг может быть проставлен позже:

```
class Movie
{
private:
   string _title;
   string _genre;
   int _year;
   double _rate;

public:
   string GetTitle() {...};
   string GetGenre() {...};
   int GetYear() {...};
   double GetRate() {...};

   void SetTitle(string& title) {...};
   void SetGenre(string& genre) {...};
   void SetYear(int year) {...};
```

```
   void SetRate(double rate) {...};

   Movie(string& title, string& genre, int year, double rate)
   {
      this->SetTitle(title);
      this->SetGenre(genre);
      this->SetYear(year);
      this->SetRate(rate);
   }

   Movie(string& title, string& genre, int year)
   {
      this->SetTitle(title);
      this->SetGenre(genre);
      this->SetYear(year);
      this->SetRate(0.0);
   }
}
```

Теперь мы можем создавать объекты Movie одним из двух способов:

```
void DemoMovie()
{
   // Вызов первого конструктора с рейтингом
   Movie movie("Крепкий орешек", "Боевик", 1988, 8.0);

   // Вызов второго конструктора без рейтинга
   Movie* movie2("Побег из Шоушенка", "Драма", 1994);
}
```

Разумеется, создавать все возможные варианты конструкторов нет необходимости. Лучше реализовывать только один конструктор, удовлетворяющий потребностям всех клиентских частей кода. Большое количество конструкторов требует и больших затрат на поддержку кода.

Возможно, вы обратили внимание, что реализация наших конструкторов дублируется. В последующих разделах будет показано, как избавиться от такого дублирования.

До того, как мы реализовали конструкторы в классе, создание объектов Movie выполнялось следующим образом:

```
Movie* movie3 = new Movie();
```

В данной строке `Movie()` – это тоже вызов конструктора. Однако после добавления конструкторов в класс, данный способ создания объектов перестает работать. Почему?

Дело в том, что язык Си++ требует, чтобы в классе (и структуре) обязательно был реализован хотя бы один конструктор. Самая простая реализация конструктора выглядит следующим образом:

```
Movie()
{
}
```

То есть, данный конструктор предназначен только для выделения памяти под объект, без инициализации его полей значениями. Аналогичный конструктор легко составить для любого класса.

В случаях, когда разработчик не создал ни одного конструктора в классе, компилятор во время компиляции автоматически добавляет такой конструктор в класс. Конструктор, который создается автоматически компилятором, называется ***конструктор по умолчанию***. Таким образом, в клиентском коде можно будет создавать объекты класса с помощью конструктора по умолчанию Movie(). После того, как разработчик

добавил в класс собственные конструкторы с параметрами, компилятор создать конструктор по умолчанию не будет. А следовательно, и в клиентском коде создавать объекты таким способом не получится.

Разработчик может создать ***конструктор без параметров***, проинициализировав поля некоторыми значениями, например:

```
Movie()
{
   this->Title = nullptr;
   this->Genre = nullptr;
   this->Year = 1900;
   this->Rate = 0.0;
}
```

Не путайте конструктор без параметров и конструктор по умолчанию: первый создается разработчиком самостоятельно, второй – создается компилятором при условии отсутствия других конструкторов в классе.

Подробнее о классах и конструкторах можно ознакомиться здесь [Лафоре, гл. 6]. В частности, следует ознакомиться с понятиями ***конструктора с параметрами по умолчанию***, ***конструктора копирования***, ***конструктора копирования по умолчанию***, ***перегрузки оператора присваивания***, ***деструктора класса***.

Методы класса

Помимо геттеров, сеттеров, конструкторов и деструкторов, внутри класса могут быть объявлены и иные методы, работающие с полями данного класса. Рассмотрим класс Circle и функцию GetArea(), рассчитывающую площадь объекта круга:

```
class Circle
{
private:
   double _x;
   double _y;
   double _radius;

public:
   double GetX() {...};
   double GetY() {...};
   double GetRadius() {...};

   void SetX(double x) {...};
   void SetY(double y) {...};
   void SetRadius(double radius) {...};

   Circle(double x, double y, double radius) {...};
}

// Функция расчета площади круга
double GetArea(Circle& circle)
{
   return 3.14 * circle.GetRadius() * circle.GetRadius();
}
```

Функция расчета площади круга тесно связана с классом круга, и предназначена только для работы с кругом. Все данные, необходимые для расчета площади круга, находятся в самом круге. В таких случаях удобно переместить функцию GetArea() в класс Circle:

```
class Circle
{
```

```
private:
   double _x;
   double _y;
   double _radius;

public:
   double GetX() {...};
   double GetY() {...};
   double GetRadius() {...};

   void SetX(double x) {...};
   void SetY(double y) {...};
   void SetRadius(double radius) {...};

   Circle(double x, double y, double radius) {...};

   double GetArea()
   {
      return 3.14 * GetRadius() * GetRadius();
   }
}
```

Так как методы классы могут обращаться к полям напрямую или вызывать внутри себя любые другие методы этого класса, сигнатура метода GetArea() упростится – передавать экземпляр класса Circle в качестве входного аргумента уже нет необходимости.

Далее представлен вызов функции GetArea() в клиентском коде:

```
Circle* circle = new Circle(0.0, 0.0, 5.0);
cout << circle->GetArea();
```

Чтобы не путать функцию GetArea() с геттерами класса, её можно переименовать в CalculateArea(), однако, это не обязательно.

С точки зрения разработки программ, реализация функциональности программы в виде методов классов предпочтительнее отдельных функций, как это принято в процедурном программировании. Удобство заключается в минимизации количества передаваемых аргументов в методы (меньше аргументов – меньше вероятность совершить ошибку при вызове функции).

Второе преимущество раскрывается при использовании автодополнения в среде разработки. Отдельно реализованные функции находятся в едином списке автодополнения. Если в вашей программе сотни или тысячи функций, поиск нужной функции в списке автодополнения может занять время. И это будет занимать какое-то время при ***каждом*** вызове ***любой*** функции. В случае же реализации функций в виде методов класса, обращаясь к объекту через точку или стрелку, мы видим перечень полей и методов, работающих непосредственно с этим классом и ***только с этим классом***. Поиск нужного метода в списке, состоящем из десятка названий, займет гораздо меньше времени.

Однако не следует бездумно перемещать все функции внутрь классов. Так, функции, предназначенные для работы с пользовательским интерфейсом, отладочные функции для проверки работы программы лучше оставлять вне классов, чтобы не засорять код классов и не ухудшать их читаемость.

Состояние и поведение, инкапсуляция и сокрытие реализации

Для дальнейшего изучения объектно-ориентированного программирования полезно ознакомиться с рядом определений. Они могут показаться излишне формализованы, поэтому для каждого определения будет дано упрощенное разъяснение. Возможно, упрощение делает термин не точным, но позволяет быстрее освоить терминологию. Сами же термины помогут в будущем проще объяснять сложные механики и принципы объектно-ориентированного программирования и проектирования.

Состояние – это перечень всех (как правило, статических) свойств данного объекта и текущих (как правило, динамических) значений каждого из этих свойств. Проще говоря, состояние – это все поля объекта и их текущее значение. У двух объектов состояние считается одинаковым только в том случае, если они являются объектами одного класса, а все поля этих объектов имеют одинаковые значения. Если объекты принадлежат разным классам, они отличаются набором полей, так как у каждого класса собственные поля. Если объекты одного класса отличаются значением хотя бы в одном поле, они обладают разными состояниями.

Поведение – это действия и реакции объекта, выраженные через изменения состояния объекта и передачу сообщений. Проще говоря, поведение – это все методы класса. Вызывая методы объекта, мы можем менять его состояние. Таким образом, состояние объекта представляет собой суммарный результат его поведения.

Интерфейс – внешний вид класса, скрывающий структуру и особенности его поведения. Другими словами, интерфейс – это все открытые члены класса (открытые поля и объявления методов). Клиентский код может взаимодействовать с объектом только через его интерфейс – те поля и методы, которые будут доступны извне класса.

Реализация – внутренний вид класса, скрывающий секретные особенности его поведения. Реализация состоит, в основном, из закрытых полей и методов, а также реализации всех методов. Стоит разделять понятия объявления метода и его реализации. Так, название метода, входные аргументы и возвращаемый тип данных относятся к объявлению метода – это его интерфейс, доступный клиентскому коду для использования. Блок кода, который выполняется внутри метода, обрабатывает входные аргументы и возвращает значения – это реализация. Реализация не видна клиентскому коду, но может быть вызвана через интерфейс.

Инкапсуляция – свойство системы, позволяющее объединить в едином описании состояние и поведение объекта с сокрытием его реализации. Фактически, механизм инкапсуляции позволяет создавать классы, помещая поля и методы в единую уникальную область видимости. Сложное слово "инкапсуляция" можно проще запомнить, зная словообразование от лат. in capsula – то есть, помещать что-то в капсулу. Подразумевается, что мы помещаем поля и методы для работы с ними в некоторый черный ящик. Со стороны клиентского кода мы не знаем, как работает этот черный ящик, но мы можем взаимодействовать с ним через его интерфейс. Знание интерфейса достаточно для работы с этим объектом.

Сокрытие реализации – принцип проектирования, заключающийся в разграничении доступа различных частей программы к внутренним компонентам друг друга. Применимо к ООП, это предоставление доступа к членам класса по средством модификаторов доступа (private, public и protected, который будет изучен позже). Сокрытие реализации позволяет с одной стороны защитить объект от некорректного использования, например, запрещая прямой доступ к полям, с другой стороны уменьшить когнитивную нагрузку при изучении сторонних классов/компонентов/библиотек, скрывая те методы и функции, которые нужны для внутренней реализации и не нужны клиентскому коду.

Часто можно встретить определение "инкапсуляция – это сокрытие реализации". Оно некорректно, так как инкапсуляция это более широкое понятие, которое использует сокрытие реализации только как инструмент. Иронично было бы сказать, что инкапсуляция инкапсулирует в себе понятие сокрытия реализации.

Требования к оформлению кода

Классы могут занимать сотни строк кода. При условии, что экран монитора может отобразить порядка пятидесяти строк, разработчик никогда не видит весь класс целиком. Изучая код конкретного метода, разработчик может одновременно видеть локальные переменные, поля и константы класса. Например, в следующем коде легко перепутать имя поля и имя входного аргумента:

```
class Phone
{
private:
   double cost;
   ...
public:
   void cost(double cost)
   {
```

```
      if (cost < 0.0)
         throw exception("Стоимость не может быть отрицательной");
      cost = cost;
   }
}
```

В данном примере поле, локальная переменная и метод названы одинаково – cost – что может создать путаницу при их использовании. Если вызов метода cost() можно будет отличить по круглым скобкам и передаче переменной, то реализация метода вызывает вопросы. Под условием if проверяется значение входной переменной или поля? В строке cost = cost происходит присвоение нового значения в поле, или значения поля в локальную переменную? Или присвоение поля в поле? Поведение метода становится неочевидным и требует времени, чтобы разобраться.

Чтобы уменьшить время на чтение кода, разработчики разрабатывают специальные требования по оформлению кода. В каждой команде могут быть собственные правила по именованию сущностей в исходном коде, но, как правило, большинство команд используют общепринятые для конкретного языка программирования требования к оформлению кода. Далее будут представлены требования по нотации RSDN для языка C#, так как, на взгляд авторов, эти правила обеспечивают достаточно комфортное чтение кода, в том числе и для Си++. Вот часть этих правил:

- Стиль именования Паскаль подразумевает такое именование, при котором все слова в имени начинаются с заглавной буквы, слова пишутся слитно без знаков подчеркивания или дефисов между собой. Например, PhoneNumber, но не phoneNumber, phonenumber, Phonenumber, Phone_Number.
- Стиль именовани Кэмел подразумевает такое именование, при котором все слова в имени начинаются с заглавной буквы, кроме первого слова; слова пишутся слитно без знаков подчеркивания или дефисов между собой. Например, phoneNumber, но не phonenumber, PhoneNumber, phone_Number. Стили Кэмел и Паскаль – базовые и используются для именования любых сущностей исходного кода.
- При именовании не используются сокращения, кроме общепринятых или тех, о которых разработчики договорились в рамках проекта. Например, GroupsCount, но не GC, GroupsC, GCount и т.д.
- При составлении названий не используются артикли the, a. При использовании родительного падежа предпочтительно использовать форму без предлога of. Например, правильным будет ArrivalTime, но не TheArrivalTime или TimeOfArrival.
- Классы и структуры именуются в стиле Паскаль от существительного в единственном числе того объекта предметной области, который класс описывает. Например, правильным будет Phone, но не Phones или phone.
- Закрытые поля класса именуются существительными в единственном числе в стиле Кэмел с нижним подчеркиванием в начале, без добавления имени класса или типа данных поля в название поля. Например, _memoryGb, но не MemoryGb, memoryGb, memories, _phoneMemoryGb, _memoryGbInt. Нижнее подчеркивание используется только у закрытых полей, что позволяет отличить закрытые поля от локальных переменных в любом контексте. Не надо добавлять название класса к полю, в противном случае при вызове будут получаться тавтологии типа phone.phoneMemoryGb.
- Открытые поля класса или структуры (все поля класса должны быть закрыты, однако в структурах открытые поля допустимы) именуются существительными в единственном числе в стиле Паскаль без подчеркиваний. Например, MemoryGb, но не _memoryGb, _MemoryGb.
- Поля-массивы или поля-указатели на массивы должны именоваться в множественном числе, но без добавления слов Array, List, Collection и т.д. Например, _students, но не _studentsArray или ArrayStudents.
- Поля, хранящие количество, например, количество элементов массива, должны называться по имени массива с добавлением в конец слова Count.
- Константные поля класса именуются заглавными буквами с нижним подчеркиванием для разделения слов. Например, MAX_STUDENTS_COUNT, но не _maxStudentsCount или MaxStudentsCount. Именование заглавными буквами позволяет быстро отличить константы от любых других переменных в исходном коде.
- Методы именуются от глагола в формулировке "Что сделать" в стиле Паскаль. Например, CalculateArea(), но не CalculationOfArea(), AreaCalculation(), _calculateArea(). Именование методов от глагола позволяет отличить методы от полей.
- Геттеры и сеттеры должны именоваться от глагола Get и Set соответственно.

- Методы, возвращающие булевы значения (кроме геттеров) должны именоваться от слова Is вместо глагола. Например IsExamPassed(), но не CheckExamPass(). Так как методы, возвращающие булево значение, часто используются внутри условий if, слово Is позволяет быстро отличить такие методы среди всех остальных, а также дает читаемую форму условия: "if (subject.IsExamPassed()) ..." читается близко к правильному английскому предложению "if the subject exam is passed, then ...". Чем больше исходный код похож на обычный язык, тем комфортнее его читать.
- Входные аргументы и локальные переменные методов именуются в стиле Кэмел без нижнего подчеркивания. Например memoryGb, но не MemoryGb, _memoryGb, memorygb. В итоге благодаря нижнему подчеркиванию или заглавным буквам, можно всегда отличить закрытое поле от открытого поля или локальной переменной, что удобно при чтении кода классов на сотни строк кода.
- Перечисления именуются в стиле Паскаль от существительного в единственном числе с добавлением слова Type или Mode в конец. Например, PaymentType, но не Payment, PaymentTypes, paymenttype. Слово Type или Mode позволяют быстро отличить перечисление от классов и структур. Исключением без слова Type или Mode могут быть названия перечислений, которые вероятно будут реализованы с помощью перечисления – Color, Weekday и т.п.
- Элементы перечисления имеются в стиле Паскаль, каждое новое значение на новой строке. Все значения сдвинуты на одну табуляцию относительно объявления перечисления.
- Фигурные скобки всегда ставятся на следующей строке после объявления класса, структуры, перечисления, операторов if или for, без сдвига табуляции (то есть равно под ключевыми словами class, struct, if, for и т.д.). Всё содержимое фигурных скобок сдвигается на одну табуляцию вправо.

Это только часть правил, которые должен помнить разработчик. Несмотря на их громоздкость, при практике программирования они достаточно быстро запоминаются. Другие правила будут упоминаться в следующих разделах по мере необходимости.

Кроме этого есть ряд правил, связанных с реализации класса на заголовочный файл и файл исходного кода (*.h и *.cpp). Так, в заголовочном файле должно храниться объявление класса, перечисление всех его полей и прототипы всех методов с указанием модификаторов доступа. В файл исходного кода выносятся реализации всех методов. Например, для класса Phone заголовочный файл может выглядеть следующим образом:

```
#pragma once

class Phone
{
private:
   string _model;
   string _company;
   int _memoryGb;
   double _screenSize;
   double _cost;

public:
     string GetModel();
     string GetCompany();
     int GetMemoryGb();
     double GetScreenSize();
     double GetCost();

     void SetModel(string model);
     void SetCompany(string company);
     void SetMemoryGb(int memoryGb);
     void SetScreenSize(double screenSize);
     void SetCost(double cost);

    Phone(string model, string company, int memoryGb, double sceenSize, double cost);
    ~Phone();
}
```

Обратите внимание на именование полей и методов по ранее описанным правилам. Также обратите внимание на использование пустых строк, визуально отделяющих блоки модификаторов доступа, а также группы методов: геттеры, сеттеры, конструкторы и деструкторы. Всё это позволяет комфортно воспринимать исходный код. Также в коде требуется оставлять комментарии для всех членов класса, но в данном примере комментарии для краткости опущены.

Cpp-файл будет выглядеть следующим образом:

```
#pragma once
#include "Phone.h"
#include <iostream>
#include <exception>

using namespace std;

string Phone::GetModel()
{
   return this->Model;
}

string Phone::GetCompany()
{
   return this->Company;
}

...

void Phone::SetModel(string model)
{
   this->Model = model;
}

...
```

В cpp-файле обратите внимание на следующие особенности:

- имя метода начинается с имени класса и двойного двоеточия `Phone::`, что является уточнением для компилятора, что описывается не просто новая функция, а реализация метода в ранее объявленном классе;
- методы уже не находятся внутри каких-либо общих фигурных скобок, а находятся в таком называемом глобальном пространстве видимости;
- несмотря на реализацию методов в отдельном файле вне фигурных скобок класса, все поля класса, а также неявный указатель this доступны внутри методов; но только тех методов, что объявлены в заголовочном файле.

Фактически, заголовочный файл содержит описание интерфейса класса, а детали реализации вынесены в cpp-файл. Таким образом, если разработчику необходимо изучить класс для использования в своём коде, ему достаточно ознакомиться с лаконичным h-файлом. Если же перед разработчиком стоит задача исправить ошибку в поведении класса, ему потребуется изучать cpp-файл. Всё это упрощает чтение кода, ускоряет разработку и уменьшает вероятность совершения ошибки.

Классы на UML-диаграммах классов

Рассмотрим графическое обозначение классов на диаграммах классов.

Обозначение класса рисуется в виде прямоугольника, разделенного на три части: шапка, область полей, область методов. В шапке без дополнительных ключевых слов пишется имя класса. В области полей

перечисляются поля класса, аналогично полям структуры. Отличие заключается в наличие знака `–` или `+` перед каждым полем. Этот знак обозначает модификатор доступа поля. Как правило, для полей это знак `-`.

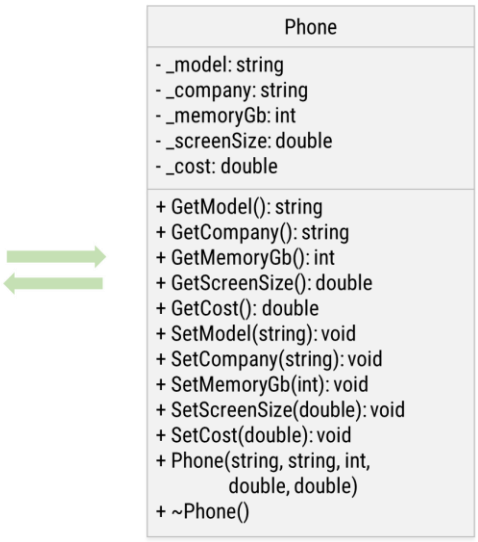
В области методов перечисляются методы класса в формате `Имя метода(аргументы): возвращаемый тип данных`. Перед каждым методом указывается его модификатор доступа.

```
class Phone
{
private:
   string _model;
   string _company;
   int _memoryGb;
   double _screenSize;
   double _cost;

public:
   string GetModel();
   string GetCompany();
   int GetMemoryGb();
   double GetScreenSize();
   double GetCost();

   void SetModel(string model);
   void SetCompany(string company);
   void SetMemoryGb(int memoryGb);
   void SetScreenSize(double screenSize);
   void SetCost(double cost);

   Phone(string model, string company,
         int memoryGb, double screenSize,
         double cost);
   ~Phone();
}
```



Обозначение класса занимает больше места, чем структура, поэтому при составлении реальных диаграмм разработчики могут опускать определенные детали по договоренности в коллективе. Например:

- не отображать имена входных аргументов или полностью скрыть входные аргументы
- не перечислять геттеры и сеттеры, предполагая, что они всегда подразумеваются
- не перечислять очевидные методы или конструкторы (конструкторы копирования, конструкторы по умолчанию).
- не отображать все закрытые члены класса, оставляя только открытые.

Диаграммы могут составляться для решения разных задач: проектирования новой программы, анализа проблем существующей программы, изучения чужой программы, составления документации к проекту. В зависимости от задачи, разработчик будет скрывать не значительные поля и методы, и оставлять наиболее важные. Также часто применяется подход, когда составляется несколько диаграмм: одна – концептуальная, вторая – детальная. Концептуальная скрывает все поля и методы классов, оставляя только блоки с названием классов и связи между ними. Такая диаграмма нужна для получения информации об архитектуре в целом. Вторая диаграмма – детальная – отображает все поля и методы классов для внимательного изучения отдельных частей архитектуры. Такие диаграммы могут составлять десятки страниц А1 и более.

В последующих разделах будут приводиться диаграммы классов. В случаях, если детали классов будут опускаться, об этом будет заранее указано в пояснительном тексте. Также, в последующих заданиях предполагается составление диаграмм классов обучающимися – обучающиеся должны рисовать полные uml-диаграммы классов, если в задании явно не указано составление упрощенных диаграмм.

Задания

Организация исходного кода в проекте

3.1.1 Организуйте структуру файлов проекта согласно требованиям, описанным в разделе "Организация исходного кода в проекте":

- Создайте подпапки для файлов каждой лабораторной работы.
- Для структур, созданных в третьей лабораторной работе, создайте заголовочные файлы и файлы исходного кода.
- Файлы должны именоваться согласно имени структуры в этом файле.
- Заголовочные файлы хранят описание структуры и прототипы функций, файлы исходного кода хранят реализацию функций для структуры.
- Убедитесь, что для новой структуры файлов программа компилируется и работает корректно.

Все последующие структуры, перечисления и классы, которые вы создадите в рамках лабораторных работ, должны также храниться в отдельных файлах с разделением на заголовочный файл и файл исходного кода.

Структуры с полями-массивами

3.2.1 Создайте структуру книги Book. В структуре должны быть поля "Название", "Год издания", "Количество страниц", массив строк "Авторы" (не более десяти), поле "Количество авторов".

3.2.2 Под структурой объявите функцию DemoBook(), внутри которой создайте переменную типа Book и присвойте в её поля значения. Обязательно присвойте несколько авторов и задайте правильное значение для поля "Количество авторов".

3.2.3 Создайте функцию ReadBookFromConsole(Book& book), которая выполняет чтение всех полей структуры книги с клавиатуры. Обратите внимание на ввод массива авторов с клавиатуры: вам сначала нужно запросить у пользователя количество авторов (от 1 до 10), затем организовать цикл с чтением нужного количества авторов. Пример работы функции:

```
Введите название книги: Далекая радуга
Введите год издания: 1963
Введите количество страниц: 224
Введите количество авторов: 2
Введите автора №1: Стругацкий А.
Введите автора №2: Стругацкий Б.
```

Примечание: обязательно реализуйте проверку правильности ввода – год издания должен быть положительным, но не более текущего года, количество страниц должно быть строго больше 0, количество авторов должно быть не менее 1 и не более 10. В случае, если пользователь ввел неправильное значение, программа должна сообщить об этом пользователю и запросить повторный ввод значения. Например:

```
Введите количество авторов: 15
Количество авторов должно быть в диапазоне 1-10. Повторите ввод
Введите количество авторов: 2
```

3.2.4 Создайте функцию WriteBookToConsole(Book& book), которая выводит данные книги на экран. Аналогично вводу, вам будет нужно организовать вывод правильного количества авторов на экран. Пример работы функции:

```
Стругацкий А., Стругацкий Б. Далекая Радуга. 1963. – 224с.
Хайнлайн Р. Звездный десант. 1959. – 320с.
Буч Г., Максимчук Р.А., Энгл М., Янг Б. Дж., Коналлен Д., Хьюстон К.А. Объектно-ориентированный анализ и проектирование с примерами приложений. 2008. – 720с.
```

Обратите внимание на оформление: между авторами должна стоять запятая, но не после последней фамилии. После названия должна стоять точка. Между годом и количеством строк обязательно разделитель в виде точки, знака пробела и дефиса. После количества страниц пишется "с."

В функции DemoBook() удалите код создания одной переменной, и добавьте код создания массива книг (не более пяти). С помощью цикла и функции ReadBookFromConsole() организуйте ввод всего массива книг с клавиатуры. Далее с помощью цикла и функции WriteBookToConsole() организуйте вывод всего массива книг на экран.

3.2.5 Создайте функцию int FindBookByAuthor(Book* books, int booksCount, string author), которая принимает на вход массив книг по указателю и среди книг находит книгу с указанным автором author. Функция должна вернуть индекс книги в массиве, среди авторов которой есть автор author (то есть для каждой книги надо перебрать всех авторов и сравнить с искомой фамилией). Если в массиве несколько книг с данным автором, функция возвращает индекс первой найденной книги с данным автором. Если в массиве нет книг с данным автором, функция возвращает -1.

В функции DemoBook() добавьте вызов функции FindBookByAuthor(). После получения индекса искомой книги в массиве, добавьте в функции DemoBook() вывод на экран найденной книги. Если индекс равен -1, выведите на экран "Нет книги с данным автором". Пример работы:

```
Введите автора для поиска книги: Хайнлайн Р.
Книга автора: Хайнлайн Р. Звездный десант. 1959. – 320с.
```

3.2.6 По аналогии со структурой Book, создайте структуру Route, описывающую маршрут общественного транспорта. В маршруте должны храниться следующие поля: номер маршрута, средняя продолжительность маршрута в минутах, частота следования маршрута в минутах (время между двумя автобусами одного маршрута), массив строк с названиями остановок, количество остановок. Для структуры Route также создайте функции ReadRouteFromConsole(), WriteRouteFromConsole(), FindRouteTo(), DemoRoute(). Функция ReadRouteFromConsole() должны Аналогично функции FindBookByAuthor(), функция FindRouteTo() должна принимать на вход массив маршруторов, их количество и название искомой остановки. Функция должна вернуть индекс первого найденного маршрута, среди остановок которого есть искомая. Функция возвращает -1, если такой остановки нет ни в одном маршруте. Работа функции должна быть продемонстрирована на массиве из трёх-пяти маршрутов.

Агрегирование

3.3.1 Создайте структуру точки Point в декартовых координатах (поля – вещественные координаты X и Y), функцию-конструктор и функции-сеттеры.

3.3.2 Создайте структуру прямоугольника Rectangle, хранящего поля длины и ширины, а также поле типа Point, хранящее центр прямоугольника в пространстве. Для структуры также создайте функцию-конструктор и функции-сеттеры с соответствующими проверками данных (длина и ширина не должны быть отрицательными).

3.3.3 Создайте функцию DemoRectangleWithPoint(), в которой объявите массив из 5 прямоугольников Rectangle с произвольными значениями координат, длин и ширин. Значения могут быть заданы из исходного кода программы (далее – программно), а не с помощью ввода пользователем с клавиатуры.

3.3.4 Допишите функцию DemoRectangleWithPoint(). Добавьте код, который выведет на экран данные о прямоугольниках массива. Пример вывода:

```
X = 5.0;   Y = 10.7;  Length = 25.4;  Width = 1.1
X = 12.0;  Y = -10.7; Length = 50.0;  Width = 7.3
X = -7.0;  Y = 16.2;  Length = 4.7;   Width = 43.6
X = 4.0;   Y = 4.0;   Length = 9.2;   Width = 23.9
X = -3.0;  Y = -2.1;  Length = 17.9;  Width = 38.0
```

Вывод данных массива должен осуществляться в цикле.

3.3.5 Допишите функцию DemoRectangleWithPoint(). Добавьте код, который рассчитает среднее значение всех центров прямоугольников и выводит его на экран. Пример вывода (значения рассчитаны для координат, представленных выше):

```
Xcenter = 2.2; Ycenter = 3.62
```

3.3.6 Составьте uml-диаграмму классов для структур Rectangle и Point. Обязательно укажите связь и кратность.

3.3.7 Создайте структуру времени Time, хранящую целочисленные поля Год, Месяц, День, Час, Минуты, функцию-конструктор и функции-сеттеры. Функции-сеттеры должны ввести реальные ограничения на месяц (не более 12), дня (для упрощения считать, что во всех месяцах 30 дней), часа (не более 24), минут (не более 60).

3.3.8 Создайте структуру авиарейса Flight, хранящего поля номера рейса, пункта отправления, пункта назначения, времени отправления и времени прибытия (композиция структуры Time). Для структуры также создайте функцию-конструктор и функции-сеттеры с соответствующими проверками данных (время прибытия не может быть раньше времени отправления).

3.3.9 Создайте функцию DemoFlightWithTime(), в которой объявите массив из 5 рейсов Flight с произвольными данными. Значения могут быть заданы программно.

3.3.10 Допишите функцию DemoFlightWithTime() для вывода на экран информации о рейсах. Пример вывода:

```
S015 Томск-Москва Вылет 02.15 18:40 Прибытие 02.15 23:25
```

Вывод данных массива должен осуществляться в цикле.

3.3.11 Напишите функцию int GetFlightTimeMinutes(Flight& flight), которая принимает на вход объект авиарейса и возвращает время в полете в минутах (или в новом объекте Time). Учтите, что нужно не просто вычесть значения полей Minutes в полях типа Time рейса, но учесть и разницу в часах, и то, что при вычитании из 20 минут 30 минут должно получиться не -10 минут, а 50 минут предыдущего часа.

Продемонстрируйте работу функции, вызвав её для каждого элемента массива в функции DemoFlightWithTime(). Убедитесь, что функция правильно считает время в полете. Пример вывода:

```
S015 Томск-Москва Время полета: 4ч 45мин
```

*На самом деле, создание структуры, описывающей время с учетом всех возможных комбинаций месяцев, дней, високосных лет, периодичности секунд, минут и часов – нетривиальная задача. В данной лабораторной работе вам предлагается создать функцию, правильно работающую только с часами и минутами.

3.3.12 Составьте uml-диаграмму классов для структур Flight и Time. Обязательно укажите связь и кратность.

Агрегирование "Один ко многим"

3.4.1 Создайте следующие типы данных:

- Перечисление жанра песни Genre
- Структуру песни Song с указанием названия песни, продолжительности, жанра
- Структуру альбома Album с указанием названия, года выпуска и массива песен
- Структуру музыкальной группы Band с указанием названия, текстового описания (история группы или её состав) и массива альбомов группы.

Для всех структур создайте функции-сеттеры, функции-конструкторы.

3.4.2 Создайте функцию Song* FindSong(string songTitle), которая находит среди всех песен группы всех альбомов песню с заданным именем и возвращает указатель на эту песню. Если песен с таким именем несколько, то вернуть первую найденную песню. Если песен с таким именем не найдено, вернуть nullptr.

3.4.3 Создайте функцию Album* FindAlbum(Song* song), которая находит среди всех песен группы альбом с заданной песней. Песня возвращает указатель на первый альбом, в котором найдена указанная песня. Если указанной песни нет ни в одном альбоме группы, функция возвращает nullptr.

3.4.4 Создайте функцию Song* GetAllSongs(Band* band, int& allSongsCount), принимающую на вход объект группы. Функция должна подсчитать количество всех песен во всех альбомах группы, затем создать динамический массив размерностью под общее количество всех песен и сохранить в этот новый массив все песни из всех альбомов. Функция должна поместить количество песен во входной аргумент allSongsCount, и вернуть указатель на сам массив из функции.

Для возврата динамического массива из функции необходимо возвращать два значения – это указатель на массив и размерность массива. Так как Си++ не позволяет возвращать две переменные из функции, данная функция должна вернуть указатель на сам массив с помощью оператора return, а размерность массива будет возвращена через входную переменную allSongsCount – по этой причине значение передается по ссылке.

В зависимости от выбранной вами реализации, функция может потребовать возврата не указателя на массив в виде Song*, а возврата указателя на массив указателей Song**. Возможны и другие варианты. При необходимости вы можете изменить сигнатуру функции. Главное – чтобы функция возвращала массив всех песен группы и количество песен.

3.4.5 Создайте функцию DemoBand(), в которой создайте объект структуры Band. Проинициализируйте группу программно, задав её три альбома по 4-5 песен в каждом альбоме. Продемонстрируйте работу функций FindSong(), FindAlbum() и GetAllSongs().

3.4.6 На основе функции GetAllSongs(), создайте функцию Song* GetAllGenreSongs(Band* band, Genre findingGenre, int& allSongsCount), возвращающую массив всех песен группы в заданном жанре findingGenre. Продемонстрируйте работу функции.

Классы

3.5.1 Перепишите все ранее созданные структуры на классы:

- Rectangle и Point
- Flight и Time
- Song, Album, Band

3.5.2 Перенесите все функции-сеттеры внутрь новых классов. Перемещение сеттеров внутрь классов приведет к необходимости исправления вызова сеттеров во всех ранее написанных функциях:

- DemoRectangleWithPoint()
- DemoFlightWithTime(), GetTimeFlightMinutes()
- FindSong(), FindAlbum(), GetAllSongs(), GetAllGenreSongs(), DemoBand()

Исправьте перечисленные функции и убедитесь, что они работают правильно.

3.5.3 Перенесите функции-конструкторы внутрь новых классов. Это также приведет к необходимости исправления всех функций, в которых создаются объекты данных классов.

3.5.4 Поместите метод GetFlightTimeMinutes() внутрь класса Flight. После переноса метода в класс ему уже не нужно будет принимать на вход объект авиарейса, так как он сможет обращаться к нему напрямую через указатель this. Исправьте функцию, и убедитесь, что она работает правильно.

3.5.5 Поместите методы FindSong(), FindAlbum(), GetAllSongs(), GetAllGenreSongs() внутрь класса Band. Исправьте функции таким образом, чтобы они работали с объектом Band через указатель this без передачи объекта в функцию в качестве входного аргумента.

3.5.6 Нарисуйте uml-диаграмму классов и перечислений Genre, Song, Album, Band.

Оформление кода

3.6.1 Убедитесь, что ваш проект имеет правильную структуру с разбиением на заголовочные файлы и файлы исходного кода, а также подпапки для лабораторных работ.

3.6.2 Проверьте правильность оформления кода классов – именование классов, полей, методов, локальных переменных и входных аргументов. Проверьте правильность расстановки табуляций и расположения фигурных скобок.

Список литературы

1) Последовательные контейнеры. Типы контейнеров / Metanit.com: сайт о программировании // URL: https://metanit.com/cpp/tutorial/7.1.php
2) Последовательные контейнеры. Вектор / Metanit.com: сайт о программировании // URL: https://metanit.com/cpp/tutorial/7.2.php
3) Последовательные контейнеры. Список / Metanit.com: сайт о программировании // URL: https://metanit.com/cpp/tutorial/7.6.php
4) Лафоре Р. Объектно-ориентиваронное программирование в C++. Глава 6 "Классы и объекты"
5) Лафоре Р. Объектно-ориентиваронное программирование в C++. Глава 7 "Массивы и строки"