Все последующие техники и механизмы относятся к объектно-ориентированному программированию

Зачем нужно объектно-ориентированное программирование?

Для пользователя?

Процессору?

Производительность?

Объектно-ориентированное программирование позволяет сделать **более строгую структуру** исходного кода

Приложения глазами разработчика

Без ООП

ООП

- меньше дублирования кода
- лучшая защищенность
  от неправильного использования
- выше читаемость кода

Структуры

```
void Function()
{
   string phoneModel = "Galaxy S20";
   string phoneCompany = "Samsung";
   int phoneMemoryGb = 32;
   double phoneScreenSize = 7.9;
   double phoneCost = 79990;
}
```

```
void Function()
{
   string phoneModel = "Galaxy S20";
   string phoneCompany = "Samsung";
   int phoneMemoryGb = 32;
   double phoneScreenSize = 7.9;
   double phoneCost = 79990;

   string phoneModel2 = "Galaxy S21";
   string phoneCompany2 = "Samsung";
   int phoneMemoryGb2 = 64;
   double phoneScreenSize2 = 7.9;
   double phoneCost2 = 99990;
}
```

```
void Function()
{
   string* phoneModels = new string[200];
   string* phoneCompanies = new string[200];
   int* phoneMemoriesGb = new int[200];
   double* phoneScreenSizes = new double[200];
   double* phoneCosts = new double[200];
}
```

```
for (int i = 0; i < 200; i++)
   for (int j = 0; j < 200; j++)
   {
      if (phoneCosts[j] < phoneCosts[i])
      {
         string tempModel = phoneModels[j];
         phoneModels[j] = phoneModels[i];
         phoneModels[i] = tempModel;
         ...
         string tempMemory = phoneMemoriesGb[j];
         phoneMemoriesGb[j] = phoneMemoriesGb[i];
         phoneMemoriesGb[i] = tempMemory;
         ...
         string tempCost = phoneCosts[j];
         phoneCosts[j] = phoneCosts[i];
         phoneCosts[i] = tempCost;

      }
   }
```

```
void SortPhones(string* phoneModels,
                string* phoneCompanies,
                int* phoneMemoriesGb,
                double* phoneScreenSizes,
                double* phoneCosts,
                int phonesCount)
{
   ...
}
```

Приходится писать очень много кода

Структура –
пользовательский составной тип данных, переменные которого хранят несколько именованных переменных

Простыми словами:
Структуры позволяют работать с группами переменных,
уменьшая количество кода

Объявление структуры

```
struct Phone
{
}
```

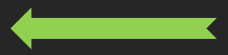
```
int Function() {...}

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

Объявление структуры

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



int Function() {...}

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

Объявление структуры

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



Эти переменные называются **поля** структуры

```
int Function() {...}

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

Создание переменной структуры

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

int Function()
{
    Phone phone;
}
```

Принцип создания переменных структуры такой же,
как и для обычных типов данных –
сначала пишем тип данных, затем имя переменной:

```
int a;
double value;
Phone myPhone;
```

Инициализация полей структуры

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

int Function()
{
    Phone phone;
    phone.Model = "Galaxy S20";
    phone.Company = "Samsung";
    phone.MemoryGb = 32;
    phone.Cost = 79990;
}
```

Оператор "." используется для обращения к полям структуры

Относитесь к нему как к "\" в названиях файлов

Массивы структур

```
int Function()
{
   Phone* phones = new Phone[200];

   for (int i = 0; i < 200; i++)
   {
      cin >> phone[i].Model;
      cin >> phone[i].Company;
      cin >> phone[i].MemoryGb;
      cin >> phone[i].ScreenSize;
      cin >> phone[i].Cost;
   }
   ...
   delete[] phones;
}
```

Сортировка массива структур

```
int Function()
{
   Phone* phones = new Phone[200];

   for (int i = 0; i < 200; i++)
      for (int j = 0; j < 200; j++)
         if (phones[j].Cost < phones[i].Cost)
         {
            Phone tempPhone = phones[j];
            phones[j] = phones[i];
            phones[i] = tempPhone;
         }
}
```

Сортировка массива структур

```
for (int i = 0; i < 200; i++)
   for (int j = 0; j < 200; j++)
      if (phones[j].Cost < phones[i].Cost)
      {
         Phone tempPhone = phones[j];
         phones[j] = phones[i];
         phones[i] = tempPhone;
      }
}
```

```
for (int i = 0; i < 200; i++)
   for (int j = 0; j < 200; j++)
   {
      if (phoneCosts[j] < phoneCosts[i])
      {
         string tempModel = phoneModels[j];
         phoneModels[j] = phoneModels[i];
         phoneModels[i] = tempModel;
         ...
         string tempMemory = phoneMemoriesGb[j];
         phoneMemoriesGb[j] = phoneMemoriesGb[i];
         phoneMemoriesGb[i] = tempMemory;
         ...
         string tempCost = phoneCosts[j];
         phoneCosts[j] = phoneCosts[i];
         phoneCosts[i] = tempCost;

      }
   }
```

Программирование – это всё про обработку данных:
сортировка, поиск, фильтрация, категоризация,
перерасчет, ввод и вывод

Чем проще писать обработку данных
(без потери производительности и надежности),
тем быстрее разработчики смогут разрабатывать новые программы

Передача структур в функцию

```
void ReadPhone(Phone& phone)
{
   cin >> phone.Model;
   cin >> phone.Company;
   cin >> phone.Cost;
   ...
}
```

Сравнение с обычными типами данных

```
void ReadPhone(Phone& phone)
{
   ...
}
```

```
void ReadInt(int& value)
{
   ...
}
```

Передача массива структур в функцию

```
void SortPhones(Phone* phones, int count)
{
   ...
}
```

Нельзя просто так передавать структуры в функцию по значению. Только по ссылке или по указателю. Позже будет объяснено почему

Работа с указателями

```
int main()
{
   // Указатели для обычных переменных
   int a = 5;
   int* pointer = &a;
   *pointer = 7;
   cout << *pointer;

   //Указатели для структур
   Phone phone;
   Phone* pointer2 = &phone;
   (*pointer2).Model = "Galaxy S20";
   cout << (*pointer2).Model;
}
```

Использовать скобки и разыменование для обращения к полям указателя неудобно. Удобнее использовать оператор ->

```
(*pointer2).Model = "Galaxy S20";
pointer2->Model = "Galaxy S20";
```

Еще один пример преимущества структур

```
void SortPhones(string* phoneModels, string* phoneCompanies, int* phoneMemoriesGb,
                double* phoneScreenSizes, double* phoneCosts, int phonesCount)
{ ... }


int main()
{
   string* phoneModels = new string[200];
   string* phoneCompanies = new string[200];
   double* phoneCosts = new double[200];
   ...

   SortPhones(phoneModels, phoneCompanies, phoneMemoriesGb,
      phoneScreenSizes, phoneCosts, 200);
   ...
   SortPhones(phoneModels, phoneCompanies, phoneMemoriesGb,
      phoneScreenSizes, phoneCosts, 200);
   ...
}
```

Функция сортировки (и другие функции, работающие с данными о телефонах) может вызываться два, три, пятьдесят раз (!) в разных местах программы

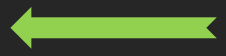
Заказчик попросил добавить в программу информацию о цвете телефонов...

```
void SortPhones(string* phoneModels, string* phoneCompanies, int* phoneMemoriesGb,
                double* phoneScreenSizes, double* phoneCosts, string* phoneColors,
                int phonesCount)
{ ... }


int main()
{
   string* phoneModels = new string[200];
   string* phoneCompanies = new string[200];
   double* phoneCosts = new double[200];
   string* phoneColors = new string[200];
   ...

   SortPhones(phoneModels, phoneCompanies, phoneMemoriesGb,
      phoneScreenSizes, phoneCosts, phoneColors, 200);
   ...
   SortPhones(phoneModels, phoneCompanies, phoneMemoriesGb,
      phoneScreenSizes, phoneCosts, phoneColors, 200);
   ...
}
```

Без структур разработчику придется исправлять
не только функции, но и каждый вызов этих функций

А что со структурами?

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

void SortPhones(Phone* phones, int phonesCount) { ... }

int main()
{
    Phone* phones = new Phone[200];
    ...
    SortPhones(phones, 200);
    ...
    SortPhones(phones, 200);
    ...
}
```



Это все изменения, которые надо сделать

Структуры:

- позволяют работать с группами переменных как единым целым;
- уменьшают количество кода;
- упрощают добавление новой функциональности в программу

Функции для работы со структурами

Функции, которые нужно создать для удобной работы с любой структурой:

- функции-сеттеры
- функции-конструкторы
- функции копирования

Функция-сеттер -
функция, позволяющая присваивать в поля только корректные значения

Проблема присвоения значений в поля

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

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

Проблема

компилятор не будет проверять логичность присваиваемых данных.
Только проверка типов данных.
Как следствие, в полях структур могут храниться некорректные данные

Функция-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);
}
```

Решение

значения в поля должны присваиваться через специальные функции – сеттеры.
Сеттер выполняет проверку значения.
Если значение ок – помещает его в поле.
Если значение не ок – выбрасывает исключение

Сеттеры пишутся для каждого поля каждой структуры, даже если для поля на данный момент нет ограничений

```
void SetMemoryGb(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,
но надежность работы программы повышается в разы
Проще искать ошибки в исходном коде

Функция-конструктор -
функция, создающая объект структуры,
и присваивающая значения в его поля

Проблема создания переменных структур

```
struct Movie
{
   string Title;   // название фильма
   string Genre;   // жанр фильма
   int Year;       // год выпуска
   double Rate;    // рейтинг фильма от 0 до 10
};
```

```
void DemoMovie()
{
   Movie* movie1 = new Movie();
   movie1->Title = "Крепкий орешек";
   movie1->Genre = "Боевик";
   movie1->Year = 1988;
   movie1->Rate = 8.0;

   Movie* movie2 = new Movie();
   movie2->Title = "Побег из Шоушенка";
   movie2->Genre = "Драма";
   movie2->Year = 1994;
   movie2->Rate = 9.1;

   Movie* movie3 = new Movie();
   movie3->Title = "1+1";
   movie3->Genre = "Комедия, драма";
   movie3->Year = 2012;
   movie3->Rate = 8.8;
   ...
}
```

Проблема

Присвоение значений в поля всегда занимает много строк кода

Решение

создать специальную функцию (функцию-конструктор)
Функция будет создавать объект и присваивать значения в поля

Функция-конструктор

```
struct Movie
{
   string Title;   // название фильма
   string Genre;   // жанр фильма
   int Year;       // год выпуска
   double Rate;    // рейтинг фильма от 0 до 10
};

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;
}
```

Использование функции-конструктора

```
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);
   ...
}

// Теперь создание любых переменных структуры занимает одну строчку кода
```

Лайфхак

Если именовать все функции-конструкторы со слова Make, то Visual Studio всегда с через автодополнение подскажет, как правильно вызвать функцию

Еще лайфхак

Для присвоения полей в объект структуры внутри конструктора вызывайте сеттеры

```
Movie* MakeMovie(string& title, string& genre,
                 int year, double rate)
{
   Movie* movie = new Movie();
   SetTitle(*movie, title);
   SetGenre(*movie, genre);
   SetYear(*movie, year);
   SetRate(*movie, rate);
   return movie;
}
```

Функция копирования -
функция, создающая объект структуры,
копируя значения всех полей из другого объекта

Проблема создания копий переменных структур

```
void DemoMovie()
{
   Movie* movie1 = MakeMovie("Крепкий орешек", "Боевик", 1988, 8.0);
   Movie* movie2 = MakeMovie("Побег из Шоушенка", "Драма", 1994, 9.1);

   // Создаём первую копию
   Movie* copiedMovie1 = new Movie();
   copiedMovie1->Title = movie1->Title;
   copiedMovie1->Genre = movie1->Genre;
   copiedMovie1->Year = movie1->Year;
   copiedMovie1->Rate = movie1->Rate;

   // Создаём вторую копию
   Movie* copiedMovie2 = new Movie();
   copiedMovie2->Title = movie2->Title;
   copiedMovie2->Genre = movie2->Genre;
   copiedMovie2->Year = movie2->Year;
   copiedMovie2->Rate = movie2->Rate;
 }
```

Проблема

Создание копии объекта занимает много строк кода

Решение

создать специальную функцию (функцию копирования)
Функция будет создавать объект,
копировать значения полей другого объекта

Создание функции копирования

```
Movie* CopyMovie(Movie& movie)
{
   Movie* copiedMovie = new Movie();
   copiedMovie->Title = movie.Title;
   copiedMovie->Genre = movie.Genre;
   copiedMovie->Year = movie.Year;
   copiedMovie->Rate = movie.Rate;
   return copiedMovie;
}
```

Использование функции копирования

```
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);

   // Создаём первую копию
   Movie* copiedMovie1 = CopyMovie(*movie1);
   Movie* copiedMovie2 = CopyMovie(*movie2);
   Movie* copiedMovie3 = CopyMovie(*movie3);
}
```

Лайфхак

Для создания копии объекта вызывайте функцию-конструктор внутри функции копирования

```
Movie* CopyMovie(Movie& movie)
{
   Movie* copiedMovie = MakeMovie(movie.Title, movie.Genre,
                                  movie.Year, movie.Rate);
   return copiedMovie;
}
```