Объектно-ориентированное программирование в C++

Классы и объекты

Инкапсуляция и сокрытие

Конструкторы

Деструкторы

Функции-члены

ООП

- ООП – концептуальный подход к проектированию сложных программных систем (*парадигма программирования*).
- Программная система декомпозируется на элементы - взаимодействующие объекты
- Объекты (классы) *инкапсулируют* в себе данные и функции
- Три кита ООП:
  - *Инкапсуляция*: все элементы, связанные с некоторой абстракцией, определяются внутри одного типа – класса; некоторые элементы класса скрыты от внешнего использования
  - *Наследование*: классы могут определяться на основе существующих базовых классов
  - *Полиморфизм*: возможность работы унифицированным способом с разными типами объектов, реализующими разное поведение
- ООП упрощает (*и поощряет*) повторное использование кода, модификацию и расширение существующей функциональности, «управление зависимостями»

Классы и объекты

- Классы в C++ (как и структуры) предназначены для введения новых пользовательских типов.
- Тип определяет: сколько памяти нужно объекту; как интерпретировать биты памяти; какие операции, методы можно использовать с переменной типа.
- В ООП **класс** определяет какими характеристиками (свойствами), функциональностью обладает объект.
- В ООП **объекты** – это экземпляры определенного класса (переменные определенного типа), конкретные представители класса с определенным состоянием (значения свойств)

```
// Работа с встроенными типами
int n = 23;
n = 17;
n++;
n = sqrt(n);
```

```
Person p("Jimmy Page", 1944);
string name = p.getName();
// обращаемся к методу Person
p.Show();
```

Классы

- Внутри класса: данные и функции
- Функции-члены часто называют *методами*
- Специальные функции класса:
  - Конструкторы (используются для создания экземпляра класса);
  - Деструкторы (используются для удаления объекта и освобождения памяти);
  - Перегрузки операторов;
- Компилятор автоматически создает специальные функции (*по умолчанию*):
  - Конструктор без параметров;
  - Деструктор;
  - Оператор копирования (x = y);

```
class Person {
    public:
        // Конструктор для создания объекта
        Person();
        // Деструктор для удаления
        ~Person();
        // Методы
        void Show() const;
        // Методы для доступа к полям
        const string getName() const;
        void setBirthYear(int);
    private:
        // Поля (данные)
        string name;
        int birthYear;
};
```

Управление доступом и инкапсуляция

- Доступ к членам класса определяется метками (спецификаторы доступа).
- В C++ используются метки: **public, private, protected**
- Действие метки сохраняется до следующей метки (в отличие от C#, Java)
- Элементы, помеченные **public**, - открыты, доступны всем внешним пользователям (составляют *интерфейсную часть* типа)
- Элементы, помеченные **private**, доступны только внутри класса (в функциях-членах и *дружественных функциях*)

Инкапсуляция

- Инкапсуляция подразумевает:
  - Выделение данных и функций работы с этими данными в одной сущности – объекте (capsula);
  - Сокрытие данных и части функциональности объекта
- В ООП рекомендуется все данные-члены класса объявлять как закрытые, тем самым запрещая напрямую обращаться к данным объекта.
- Внешние пользователи получают опосредованный доступ к элементам объекта через открытые методы (*интерфейсную часть*). Методы доступа к конкретным элементам также называют как getter, setter
- Инкапсуляция как сокрытие используется:
  - Для устранения из области видимости внутренней функциональности, не представляющей интерес для внешнего пользователя
  - Для предотвращения некорректного определения объекта
  - Для возможности изменений в «закрытой части» без необходимости изменения кода, который работает с объектом через *интерфейсную часть*

Объявление класса и определение

- Объявление класса должно содержать объявление функций-членов и функций данных. Размещают в header-файле (н-р, **person.h**)
- Определение функций-членов размещают в файле кода (**person.cpp**).

```
void Person::Show() const
{
    cout << "Name: " << name << ", Age: " << age << endl;
}
```

- В объявлении класса можно привести и определение элементов. Как правило, выполняют для «быстрых» открытых функций. Такие функции автоматически являются *подставляемыми* (inline-функции).

```
class Person {
    public:
        const string getName() const { return name; };
        ..
};
```

inline-функции

- Inline-функции – компилятор вместо вызова функции (переход по определенному адресу функции) подставляет тело функции в точку вызова

```
// Внешняя подставляемая функция
int inline maxValue(int a, int b) { return a > b ? a : b; }
```

```
// В коде есть вызов функции
cout << maxValue(23, 17) << endl;
```

```
// Компилятор подставляет тело функции
cout << (23 > 17 ? 23 : 17) << endl;
```

- Ключевое слово inline является рекомендацией компилятору
- Подстановка приводит к более эффективному выполнению «коротких» функций: не тратится время на переход по адресу, нет дополнительной работы со стэковой памятью, не происходит копирования аргументов

Конструктор

- Специальная функция-член (метод) класса, которая вызывается при создании объекта.
- Конструкторы определяют «правила» создания объектов (*каким образом можно создать объект*)
- Можно ввести несколько конструкторов (перегрузки)

```
class Person
{
    private:
        string name;
        int age;
    public:
        Person()  { name = “No-name”, age = 10; }
        ..
};
```

```
// Создание объекта с помощью конструктора
Person man;
```

Конструктор

- Явная форма вызова конструктора

```
Person other = Person("Bob", 23);
```

- Неявная форма вызова конструктора:

```
Person man("John", 17);
```

- Вызов конструктора при динамическом создании объекта:

```
Person* lady = new Person(“Ann", 17);
```

- *Конструктор по умолчанию* вводится компилятором в случае отсутствия явных конструкторов для класса:

```
Person somebody;
```

- Можно создать свой конструктор без параметров, либо используя значения по умолчанию в конструкторе с параметрами

```
// "Списковая инициализация"
Person p1 { "Peter", 11};
Person p2 = { "Nick", 12 };
Person p3 {};
```

```
// явный вызов конструктора без параметров
Person goodOne = Person();
// объявление функции
Person badOne();
```

Объявление конструкторов

- Конструктор может использовать *выражения инициализации* или вызов другого конструктора

```
class Person
{
    public:
      Person() : name("Anonyme"), age(0) {  ..   }
       // Выражение инциализации
       Person(string name, int age) : name(name), age(age) { }
       // Вызываем другой конструктор
       Person(string name) : Person(name, 0) { }
  ..
};
```

Приведение при инициализации

- Неявный вызов конструктора с одним параметров

```
// Опасная форма
Person pDanger = string("DangerMan");
Person StrangeMan = 22;
```

```
class Person
{
    public:
        Person(int age);
        Person(string name);
     ..
};
```

- Можно запретить такой вызов, если пометить конструктор с `explicit`

```
explicit Person(int age);
```

Деструкторы

- Деструкторы – специальные методы, предназначены для освобождения ресурсов (памяти).
- Если при создании объекта выделяется динамическая память (н-р, с помощью `new`), тогда в деструкторе необходимо освобождать память с помощью `delete`.
- Деструктор по умолчанию вводится компилятором, если не создается явный деструктор.

```
// Определение деструктора
Person :: ~Person()
{
    cout << "[ Destructor ] ";
    cout << name << ", " << age;
    cout << " <" << this << ">" << endl;
}
```

Вызов деструкторов

- Как правило, деструкторы не вызываются напрямую. Вызов выполняется компилятором.
- Для локальных объектов, размещенных в стэковой памяти, деструкторы автоматически вызываются при выходе из области видимости объекта (метода, функции, структурного блока, ограниченного фигурными скобочками)

```
{
    Person one = Person();
}
```

- Для объектов, размещаемых в динамической памяти, деструктор автоматически вызывается после вызова `delete`.

```
Person* p = new Person("Mr. Pointer", 32);
delete p;
```

Присвоение\Инициализация

- Присвоение предполагает копирование содержимого
- Использование в правой части конструктора предполагает создание временного объекта

```
// создание объекта x
Person x;
// создание временного объекта и копирование
x = Person("Mr. X", 23);
```

```
[Constructor] Empty object  <0x61fc90>
[Constructor] Mr. X, 23     <0x61fdb0>
[Destructor  ] Mr. X, 23    <0x61fdb0>
Program finish
[Destructor  ] Mr. X, 23    <0x61fc90>
```

- Инициализация предполагает создание объекта без временного

```
Person x = Person(“Mr. X”, 23);
```

Const-методы

- Методы, не изменяющие объект, рекомендуется помечать как константные

```
void Show() const
{
    cout << "Name: " << name << ", Age: " << age << endl;
}
```

- При работе с константным объектом можно обращаться только к константным методам:

```
const Person pConst { "Constantin", 12};
pConst.Show();
```

Указатель this

- В методах класса можно оперировать указателем на текущий объект

```
void Show() const
{
	cout << "Name: " << this->name << ", ";
	cout << "Age: " << age << endl;
}
```

- Указатель this можно использовать для получения ссылки на текущий объект

```
const Person& Person::getOlder(const Person& other) const
{
	return age > other.age ? *this : other;
}
```

Структурирование кода: *.h vs *.cpp

- Классы, которые используются в нескольких файлах (проектах) рекомендуется объявлять в заголовочном файле *.h (с таким же именем).
- Нетривиальные методы класса рекомендуется переносить в *.cpp файл.
- Заголовочные файлы представляют собой открытый интерфейс для внешних пользователей.
- Некоторые фрагменты C++ кода должны компилироваться совместно с кодом «пользователей», поэтому определяются в h-файлах (шаблонные функции)
- Изменение h-файла требует перекомпиляции всех «пользователей»

- Вне программ с открытым исходным кодом (где предоставляются оба файла: .h и .cpp), большинство сторонних библиотек предоставляют только заголовочные файлы вместе с предварительно скомпилированным файлом библиотеки. На это есть несколько причин:
- На этапе линкинга быстрее будет подключить предварительно скомпилированную библиотеку, чем выполнять перекомпиляцию каждый раз, когда она нужна.
- Защита интеллектуальной собственности (создатели не хотят, чтобы другие просто «воровали» их код).