НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»
Кафедра информатики и процессов управления (№17)

Лекции по информатике и программированию

Лекция 2

Основы объектно-ориентированного программирования.

Содержание

1. **Объекты и классы**
   - 1.1. **Функции-члены (методы)**
   - 1.2. **Неявный указатель на объект (`this`)**
   - 1.3. **Защита, `private` и `public`**
   - 1.4. **Конструктор объекта**
   - 1.5. **Инициализация и присваивание**
   - 1.6. **Единица защиты**
   - 1.7. **Зачем нужна защита**
   - 1.8. **Избыточность и целостность**
   - 1.9. **Инкапсуляция**
   - 1.10. **Классы**
   - 1.11. **Деструкторы**

1. Объекты и классы

1.1. Функции-члены (методы)

Рассмотрим пример создания структуры для работы с комплексными числами в декартовых координатах:

$$z = x + i \cdot y, \qquad x = \operatorname{Re} z, \qquad y = \operatorname{Im} z$$

```
struct complex {
    double re, im;
};
```

Функция на Си для вычисления модуля комплексного числа:

```
double modulo(struct complex *c) {
    return sqrt(c->re*c->re + c->im*c->im);
};
```

На языке Си++ функцию можно внести внутрь структуры:

```
struct complex {
    double re, im;
    double modulo() { return sqrt(re*re + im*im); }
};
```

Функция `modulo` называется ***функцией-членом*** структуры (или ***методом*** структуры).

Вызов такой функции:

```
complex z;
double mod;
z.re = 2.7;
z.im = 3.8;
mod = z.modulo();
```

Функция-член (метод) вызывается **для конкретной** структуры (объекта `z`).

Вызов метода – *отправка сообщения* объекту.
Возвращаемое значение – *полученный* от объекта *ответ*.

1.2. Неявный указатель на объект (`this`)

На уровне машинного кода: вызов функции-члена (метода) – то же самое, что и вызов обычной функции, первым параметром которой является адрес объекта. Функциям-членам при вызове их для объекта передается ***неявный параметр*** – адрес объекта, для которого функция вызывается.

К этому параметру можно обратиться через ключевое слово `this`:

```
struct complex {
    double re, im;
    double modulo() {
        return sqrt(this->re*this->re + this->im*this->im);
    }
};
```

Можно воспринимать `this` как локальную константу, имеющую тип указателя на описываемый объект (в примере – тип `complex *`).

В приведенном примере это не имеет особого смысла. Но использование `this` оказывается необходимым, когда внутри объекта нужно вызвать какую-либо внешнюю функцию, аргументом которой должен стать сам данный объект.

1.3. Защита, `private` и `public`

В языке Си++ имеется возможность ***защиты***, которая позволяет запретить доступ к некоторым частям структуры (полям или функциям-членам) из любых мест программы, кроме тел функций-членов (всех функций-членов данной структуры).

Ключевое слово `private` помечает поля и функции-члены, доступные только из тел функций-членов данной структуры. Ключевое слово `public` помечает поля и функции-члены, доступные извне структуры. Например:

```
struct complex {
private:
    double re, im;
public:
    double modulo() { return sqrt(re*re + im*im); }
};
```

Но теперь пользоваться такой структурой стало невозможно – нет средств для задания значений полей `re` и `im`. Попытка прямого присваивания значений полям приведет к ошибке компиляции:

```
complex z;
z.re = 2.7;     // ошибка компиляции!
z.im = 3.8;     // ошибка компиляции!
```

Поэтому добавим функцию-член для задания значений полей:

```
struct complex {
private:
    double re, im;
public:
    void set(double re_val, double im_val)
        { re = re_val;  im = im_val; }
    double modulo()
        { return sqrt(re*re + im*im); }
};
```

Тогда пройдет компиляцию такое использование объекта `z`:

```
complex z;          // 1 – объявление z
double mod;
z.set(2.7, 3.8);    // 2 – задание значений полей z
mod = z.modulo();
```

Это решение имеет серьезный недостаток: с момента объявления объекта `z` (**1**) до вызова функции-члена `set` (**2**) сама переменная `z` находится в **неопределенном состоянии** (попытки ее использовать будут заведомо ошибочны). Это противоречит принципу ООП:

Вне объекта не следует делать предположений о его внутреннем состоянии.

Инициализацию объекта нужно провести в момент его создания. Необходимо **запретить** создание объекта без инициализации.

1.4. Конструктор объекта

Конструктор объекта – это функция-член, описывающая действия, которые необходимо выполнять каждый раз при создании объекта. **Имя** конструктора **совпадает** с именем описываемого типа.

Для описываемого типа структуры:

```
struct complex {
private:
    double re, im;
public:
    complex(double re_val, double im_val)
        { re = re_val;  im = im_val; }
    double modulo()
        { return sqrt(re*re + im*im); }
};
```

Конструктор используется **только для создания объекта**, сам конструктор в явном виде никогда не вызывается. Поэтому тип возвращаемого конструктором значения не указывается – конструктор ничего не возвращает, результатом его работы является созданный объект.

Создание структуры происходит посредством конструктора:

```
complex z(2.7, 3.8);     // создание z
double mod;
mod = z.modulo();
```

Можно вообще не описывать переменную `z`:

```
double mod = complex(2.7, 3.8).modulo();
```

здесь создается ***анонимная переменная*** типа `complex` и для этой переменной вызывается функция-член `modulo`.

После введения конструктора, имеющего параметры, предыдущая версия объявления переменной `z` перестанет компилироваться:

```
complex z;      // ошибка компиляции!
```

(Если это создает неудобства, то можно снова сделать такое объявление корректным с помощью *конструктора умолчания*, описанного ниже.)

Можно считать, что при создании объекта происходит вызов конструктора, но на самом деле конструктор используется для инициализации объекта.

1.5. Инициализация и присваивание

В языке Си++ любая переменная создается с помощью конструктора. Если в программе конструктор не описан, часто считается, что конструктор существует **неявно**.

Синтаксис создания переменной с помощью неявного конструктора допустим в языке Си++ даже для переменных встроенных типов. Например, такие описания эквивалентны:

```
int a(5);     // инициализация в Си++
int a = 5;    // инициализация в Си и Си++
```

Такое становится возможно потому, что конструктор выполняет инициализацию объекта и не имеет никакого отношения к присваиванию значений.

Инициализация отличается от ***присваивания***: при присваивании выполняется действие – изменяется значение переменной, которая уже существует, а при инициализации задается значение переменной, которой до этого момента не было.

Разница между присваиванием и инициализацией становится очевидна уже в языке Си, когда при инициализации массива можно задать все сразу все значения его элементов:

```
int m[6] = {10, 20, 30, 40, 50, 60};    // инициализация массива целиком
```

Тогда как присваивать массивы целиком невозможно – присваивание приходится выполнять отдельно для каждого элемента массива (например, в цикле).

В языке Си++ разница между инициализацией присваиванием еще более существенна. Конструкторы используются только для инициализации объектов. Для присваивания объектам значений используются другие средства.

1.6. Единица защиты

В Си++ приведенное выше описание структуры `complex` фактически является описанием **нового типа данных** с именем `complex`.

Создание каждого объекта (объявление каждой переменной) этого нового типа данных происходит только с помощью конструктора. Создание нескольких объектов:

```
complex z1(5.2, 4.73), z2(3.56, 9.1);
```

Создание массива объектов возможно только с инициализацией:

```
complex m[3] = {complex(5.2, 4.73), complex(3.56, 9.1), complex(4.0, 3.0)};
```

В языке Си++ ***единицей защиты*** является не объект, а **тип целиком**.

В описании структуры `complex` действие ключевых слов `private` и `public` распространяется на весь тип `complex` целиком. Это значит, что из тел функций-членов можно обращаться к закрытым полям не только текущего объекта (для которого вызывается эта функция-член), но и к закрытым полям любого объекта данного типа.

Чтобы это проиллюстрировать, дополним структуру `complex` функцией-членом `zero` в разделе `public`, которая обнуляет действительную и мнимую части комплексного числа:

```
void zero(complex *z)
    { z->re = 0.0;  z->im = 0.0; }
```

Тогда для описанных переменных `z1` и `z2`:

```
cout << z1.modulo() << " " << z2.modulo() << endl;
z2.zero(&z1);
cout << z1.modulo() << " " << z2.modulo() << endl;
```

Здесь через вызов функции-члена `zero` объекта `z2` обнулены закрытые поля объекта `z1`. В чем можно убедиться, вызвав функцию-член `modulo` для обоих объектов до и после обнуления.

1.7. Зачем нужна защита

Представим, что структуры `complex` используется в большой программе, активно работающей с комплексными числами. Дополним структуру функциями-членами для получения значений действительной (`get_re`) и мнимой (`get_im`) частей отдельно:

```
struct complex {
private:
    double re, im;
public:
    complex(double re_val, double im_val)
        { re = re_val;  im = im_val; }
    double get_re()  { return re; }
    double get_im()  { return im; }
    double modulo()  { return sqrt(re*re + im*im); }
    double argument()  { return atan2(im, re); }
};
```

Здесь также добавлена функция-член `argument`, которая для комплексного числа, представленного в полярных координатах

$$z = r \cdot (\cos\varphi + i \cdot \sin\varphi) = r \cdot e^{i\cdot\varphi}, \qquad r = |z|, \qquad \varphi = \arg(z),$$

вычисляет аргумент φ с помощью стандартной функции `atan2` из `<math.h>`.

К защищенным полям `re` и `im` такой структуры могут обращаться только функции-члены самой структуры, в других частях программы эти поля не доступны.

Допустим, что в программе модуль комплексного числа используется гораздо чаще, чем действительная и мнимая части. Тогда, чтобы сократить количество вычислений, комплексные числа разумнее хранить в полярных координатах, а не в декартовых.

Для этого заменим в структуре поля `re` и `im` на новые поля `mod` и `arg`. При этом переписывать всю программу не придется: поскольку поля структуры защищены, то достаточно внести изменения только в функции-члены структуры:

```
struct complex {
private:
    double mod, arg;
public:
    complex(double re, double im) {
        mod = sqrt(re*re + im*im);
        arg = atan2(im, re);
    }
    double get_re()  { return mod*cos(arg); }
    double get_im()  { return mod*sin(arg); }
    double modulo()  { return mod; }
    double argument()  { return arg; }
};
```

Можно даже сохранить обе реализации структуры `complex`, выбор между которыми будет осуществляться директивами условной компиляции, и использовать ту из них которая будет давать большее быстродействие.

1.8. Избыточность и целостность

Представим, что структуры `complex` Более того, можно хранить в структуре и декартово, и полярное представления комплексного числа одновременно:

```
struct complex {
private:
    double re, im, mod, arg;
public:
    complex(double re_val, double im_val) {
        re = re_val;  im = im_val;
        mod = sqrt(re*re + im*im);
        arg = atan2(im, re);
    }
    double get_re()  { return re; }
    double get_im()  { return im; }
    double modulo()  { return mod; }
    double argument()  { return arg; }
};
```

В этом случае вычисления модуля и аргумента выполняются только один раз – в конструкторе.

Теперь в объекте четыре поля, и значения двух пар полей (`re`, `im`) и (`mod`, `arg`), которые задают одно и то же комплексное число, должны всегда находиться в определенном соотношении друг с другом. В таких случаях говорят, что хранимая информация ***избыточна***.

При избыточности должна обеспечиваться ***целостность*** информации – гарантия того, что в программе значения полей всегда будут находиться в зафиксированном для них соотношении. Для защищенных полей структуры такую гарантию обеспечить гораздо легче.

1.9. Инкапсуляция

Защита в языке Си++ предназначена не для того, чтобы защититься от злоумышленников, но исключительно для **защиты от собственных ошибок** программистов.

Обойти механизм защиты не составляет особых проблем. Защита работает только в том случае, если программисты не предпринимают целенаправленных действий по ее обходу. Попытки обойти защиту, скорее всего, приведут к внесению ошибок в программу.

В языке Си++ механизм защиты воплощает принцип проектирования, который в программировании называется ***сокрытие***:

> ***Сокрытие*** **— разграничение доступа различных частей программы к внутренним компонентам друг друга.**

Сокрытие деталей реализации некоторой части программы от всей остальной программы в языке Си++ тесно связано с понятием ***инкапсуляции***:

> ***Инкапсуляция*** **— разделение элементов абстракции, определяющих ее структуру (данные) и поведение (методы), и изоляция контрактных обязательств абстракции (протокол/интерфейс) от их реализации.**

В языке Си++ ***инкапсуляцией*** называется разделение объекта на данные (поля, члены) и методы (функции-члены) и отделение защищенной части объекта (`private`) от интерфейсной (`public`).

1.10. Классы

В языке Си++ введен составной тип переменных, называемый ***классом***. От структуры (`struct`) он отличается тем, что к **полям** (***членам***) класса доступ по умолчанию есть только из **методов** (***функций-членов***) самого этого класса.

Реализация комплексного числа в виде класса:

```
class Complex {
    double re, im;
public:
    Complex(double re_val, double im_val)
        { re = re_val;  im = im_val; }
    double get_re()  { return re; }
    double get_im()  { return im; }
    double modulo()  { return sqrt(re*re + im*im); }
    double argument()  { return atan2(im, re); }
};
```

Для класса все, что описано до ключевого слова `public` – защищенные детали реализации класса, доступные только из его функций-членов.

Защита, включаемая по умолчанию, – это единственное отличие классов (`class`) от структур (`struct`) с точки зрения компилятора Си++. Поэтому для открытых полей обычно используются структуры, а для скрытых – классы.

1.11. Деструкторы

Наряду со средствами **создания** объекта (конструкторами) в языке Си++ предусмотрены средства контроля над **уничтожением** объекта – ***деструкторы***.

Объект в ходе своей деятельности может захватить некоторый ресурс – например, открыть файл, выделить динамическую память и т.д. Если на захваченный ресурс ссылаются только закрытые поля объекта, то о захвате ресурса ни в какой другой части программы не известно. Поэтому перед прекращением своего существования объект обязан освободить захваченный ресурс.

Деструктор – функция-член класса, вызов которой автоматически вставляется компилятором в код в любой ситуации, когда объект прекращает существование.

Имя деструктора **совпадает** с именем класса (или структуры), к которому спереди добавлен знак «**~**» (тильда):

```
class File {
    int fd;  // Дескриптор файла (при значении  -1  файл не открыт).
public:
    File() { fd = -1; }  // В момент создания файл еще не открыт.
    bool OpenRO(const char *name) {  // Попытка открыть файл на чтение:
        fd = open(name, O_RDONLY);   //     true – успех,
        return (fd != -1);           //     false – неудача.
    }
    ...      // Другие функции-члены для работы с файлом.
    ...
    ~File { if (fd!=-1) close(fd); }  // Закрываем файл, если он открыт.
};
```

Список параметров деструктора всегда пуст (передать ему параметры невозможно). Как и конструктор, деструктор не возвращает никаких значений.

Конструктор и деструктор в языке Си++ в явном виде вызывать не принято.

В языке Си++ при *создании* объекта **ровно один раз** отрабатывает *конструктор*, при *уничтожении* объекта **ровно один раз** отрабатывает *деструктор*.