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

Лекции по информатике и программированию

Лекция 3

Абстрактные типы данных (часть 1)

Содержание

Обозначения:

ООП – объектно-ориентированное программирование;

АТД – абстрактные типы данных.

Защита, конструкторы и деструкторы используются как для создания объектов в терминах ООП, так и для разработки АТД.

Различие ООП и АТД:

	ООП (класс, объект)	**АТД (тип, структура)**
Внутреннее устройство	не доступно извне	
Внешняя открытость	интерфейсная часть	
Внутреннее состояние	имеется, но вне объекта не должно использоваться	отсутствует
Обмен сообщениями	доступен извне	отсутствует
Поля (члены)	защищены, доступны только из методов	
Методы (функции-члены)	обязательны для обмена сообщениями	необязательны, один из возможных способов описания операций

Примеры средств АТД, которые не имеют отношения к парадигме ООП:

- возможность ввести свои версии операций (например, арифметических);
- контроль объектов за собственным созданием, копированием, присваиванием и ликвидацией;
- концепция типа-ссылки (позволяющее ввести *леводопустимое выражение* для типа).

Программист может создать **сколь угодно сложную абстракцию** и работать с ней так же, как с встроенными типами данных, **полностью игнорируя детали реализации**.

1. Перегрузка функций и операций

1.1. Перегрузка имен функций

Перегрузка имен функций (перегрузка функций, function overloading) – техническое решение языка Си++ (не относящееся к ООП, АТД и вообще парадигмам программирования), которое позволяет в одной области видимости ввести **несколько** различных функций, имеющих **одно и то же имя**, но разное количество и/или тип параметров. Такие функции называются ***перегруженными***.

Например:

```
void print(int n)  { printf("%d\n", n); }
void print(double x)  { printf("%lf\n", x); }
void print(const char *s)  { printf("%s\n", s); }
void print()  { printf("Hello!\n"); }
```

При обработке вызова компилятор определяет, какую из трех функций вызвать, учитывая количество и тип фактических параметров:

```
print(50);            // вызов версии print(int)
print(2.5);           // вызов версии print(double)
print("some text");   // вызов версии print(const char*)
print();              // вызов версии print() без параметров
```

Введение перегруженных функций может привести к **ошибкам компиляции**. Если в рассмотренном примере не описывать функцию `print(int)`, то вызов

```
print(1);           // вызов версии print(double)
```

откомпилируется успешно, поскольку целочисленная константа `1` неявно преобразовывается к типу `double`. Но вызов

```
print(0);              // ошибка компиляции!
```

приведет к ошибке компиляции из-за **неоднозначности выбора типа параметра**: константа `0` может быть с одинаковым успехом расценена как константа типа `double` или как адресная константа «нулевой указатель». Однако, если бы присутствовала только одна из описанных функций – `print(double)` или `print(const char*)`, то вызов `print(0)` был бы корректен.

1.2. Понятие полиморфизма

В языке Си++ перегрузка функций – пример проявления ***статического полиморфизма***, вводимого пользователем.

> ***Полиморфизм*** **– способность одинаковых конструкций языка программирования обозначать различные действия в зависимости от типов задействованных переменных и значений.**

Полиморфизм в общем случае может не относиться ни к ООП, ни к АТД, как, например, перегрузка имен функций в Си++.

В языке Си тоже проявляется полиморфизм, но ***встроенный в язык***. Например, операция, обозначаемая знаком «`/`», если оба операнда имеют тип `int`, выполняет целочисленное деление (результат имеет тип `int`, а остаток отбрасывается). Но если хотя бы один операнд имеет тип `double`, то деление выполняется над числами с плавающей точкой, а результат также имеет тип `double`.

Более того в языке Си операция сложения `a+b` может обозначать целочисленное сложение или сложение чисел с плавающей точкой (на уровне инструкций процессора эти два вида сложения – совсем разные операции).

1.3. Перегрузка операций

Еще одним проявлением статического полиморфизма в языке Си++ является возможность переопределить символы стандартных операций (***перегрузить операции***).

Возвращаясь к примеру комплексных чисел, создадим функции-члены, описывающие действия операций сложения, вычитания, умножения и деления комплексных чисел (с использованием стандартных символов этих операций).

```
class Complex {
    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); }
    double argument()  { return atan2(im, re); }
    double get_re()  { return re; }
    double get_im()  { return im; }
    Complex operator+(Complex op2) {
        Complex res(re + op2.re, im + op2.im);
        return res;
    }
    Complex operator-(Complex op2) {
        Complex res(re - op2.re, im - op2.im);
        return res;
    }
    Complex operator*(Complex op2) {
        Complex res(re*op2.re – im*op2.im, re*op2.re + im*op2.im);
        return res;
```

```
    }
    Complex operator/(Complex op2) {
        double dvs = op2.re*op2.re + op2.im*op2.im;
        Complex res((re*op2.re + im*op2.im)/dvs,
                    (re*op2.re - im*op2.im)/dvs);
        return res;
    }
};
```

Ключевое слово `operator` и знак операции «`+`» образуют **имя функции**, которую для объектов класса `Complex` можно вызвать двумя способами:

```
a = b.operator+(c);
a = b + c;
```

Последнюю форму записи можно использовать в выражениях:

```
Complex c1(2.7, 3.8);
Complex c2(1.15, -7.1);
double m = (c1+c2).modulo();
```

Аналогичным образом в Си++ можно переопределить символы любых операций (которые называются ***перегружаемыми***), кроме двух:

1) тернарная операция условия «`a ? b : c`»,
2) операция обращения к полю структуры или класса «`a.f`» (точка).

Стандартную операцию можно перегрузить как обычную функцию вне класса.

Если разные операции имеют одно и то же обозначение, то компилятор выбирает из нескольких перегруженных функцию ту, которую следует выполнить, на основании **типов аргументов** (операндов).

Перегрузка операций в Си++ является существенной поддержкой концепции АТД.

1.4. Конструктор умолчания

Поскольку конструктор класса (или структуры) является функцией-членом класса, имя которой совпадает с именем класса, то для конструкторов тоже имеется возможность перегрузки имен функций. При этом перегружаемые конструкторы различаются количеством и/или типом параметров.

Например, для класса `Complex` создадим конструктор, который позволит проводить инициализацию объектов по умолчанию:

```
class Complex {
    double re, im;
public:
    Complex(double re_val, double im_val)
        { re = re_val;  im = im_val; }
    Complex()
        { re = 0.0;  im = 0.0; }
    ...
```

Конструктор с пустым списком параметров называется ***конструктором умолчания***. Он делает возможным привычное описание объектов без указания значений для инициализации:

```
Complex c;       // используется конструктор умолчания
Complex m[10];   // конструктор умолчания вызывается 10 раз
```

1.5. Конструктор преобразования

В языке Си при выполнении операций с операндами разных типов вначале проводится **неявное преобразование типов**.

Например, при выполнении сложения:

```
double x=5.8, y;
int n=37;
y = x + n;
```

значение второго операнда (переменной `n` типа `int`) компилятор сначала неявно преобразует к типу `double`, а затем выполнит сложение для двух операндов типа `double`.

Также если в программе описана функция с прототипом:

```
void func(double);
```

ее вызов можно выполнить с фактическим параметром типа `int`:

```
func(n);
func(25);
```

При этом компилятор сначала неявно преобразует значение фактического параметра к типу `double`, а затем передаст его в функцию.

То есть в языке Си имеется правило неявного преобразования значений типа `int` в значения типа `double`. В языке Си++ это тоже поддерживается, но также имеются средства указывать такие правила преобразования для **типов данных, введенных пользователем**.

В СИ++ одним из таких средств является ***конструктор преобразования***.

По аналогии с приведенным примером, преобразовать значение типа «A» в значение типа «B» – это значит задать инструкцию по созданию объекта типа «B» по имеющемуся значению типа «A». Создание объектов Си++ выполняет конструктор, поэтому в классе «B» необходимо описать конструктор, получающий параметры типа «A» – он и будет выполнять неявное преобразование типов.

Например, для класса Complex такой конструктор преобразовывает вещественное число (типа double) в комплексное:

```
class Complex {
    double re, im;
public:
    Complex(double re_val, double im_val)
        { re = re_val;  im = im_val; }
    Complex(double val)
        { re = val;  im = 0.0; }
    Complex()
        { re = 0.0;  im = 0.0; }
    ...
```

Конструктором преобразования называется конструктор, который получает ровно один параметр, имеющий тип отличный от описываемого класса. Он используется компилятором не только для явного создания объекта:

```
Complex c(4.7);
```

но и для неявного преобразования типов. Например, если в программе имеется функция:

```
void func(Complex a);
```

то ее можно вызвать для вещественного параметра:

```
func(9.5);
```

При этом с помощью конструктора преобразования создается **временный анонимный** объект типа Complex, который передается в функцию func в качестве фактического параметра.

2. Ссылки

2.1. Понятие ссылки

Язык Си++ имеет важное понятие ***ссылки***, которого нет в языке Си и других языках программирования.

Ссылка – особый вид данных, хранящий **адрес переменной**, но (в отличие от указателя) ссылка **семантически эквивалентна переменной**, на которую она ссылается.

Любые операции над ссылкой проводятся над той переменной, адрес которой содержится в ссылке. **Саму ссылку изменить невозможно**, ее значение задается в момент создания ссылки и остается неизменным все время существования ссылки.

Пример описание ссылки:

```
int i;
int *p = &i;    // p - указатель на переменную i (& - взятие адреса).
int &r = i;     // r - ссылка на i (& - символ обозначения ссылки).
...
          // Увеличить значение i на 1:
i++;      //   инкрементированием самой переменной i,
(*p)++;   //   через указатель p на переменную i,
r++;      //   по ссылке r на переменную i.
```

Ссылка с именем `r` – синоним имени переменной `i`.

Ссылки часто называют **переменными ссылочного типа**, хотя это не вполне корректно, поскольку изменить значение самой ссылки нельзя (все действия происходят над переменной, на которую ссылка ссылается). Присвоить значение самой ссылке также невозможно, поэтому ссылку **нельзя описать без инициализации** (здесь также проявляется, что инициализация и присваивание – совершенно разные вещи).

2.2. Передача параметров по ссылке

Чаще всего в языке Си++ ссылки используются для передачи параметров в функции и возврата функциями значений. В языке Си способ передачи параметров по ссылке отсутствует.

Рассмотрим пример функции, которая находит минимальный и максимальный элементы массива. В языке Си приходится использовать передачу параметров по адресу:

```
void minmax(double *arr, int k, double *min, double *max) {
    int i;
    *min = arr[0];
    *max = arr[0];
    for (i=1; i<k; i++) {
        if (*min > arr[i])
            *min = arr[i];
        if (*max < arr[i])
            *max = arr[i];
    }
}
```

Вызов такой функции:

```
double m[100];
double min, max;
...
minmax(m, 100, &min, &max);
```

В языке Си++ можно использовать передачу параметров по ссылке:

```
void minmax(double *arr, int k, double &min, double &max) {
    int i;
    min = arr[0];
    max = arr[0];
    for (i=1; i<k; i++) {
        if (min > arr[i])
            min = arr[i];
        if (max < arr[i])
            max = arr[i];
    }
}
```

Вызов этой функции:

```
double m[100];
double min, max;
...
minmax(m, 100, min, max);
```

Ссылки часто называют **переменными ссылочного типа**, хотя это не вполне корректно, поскольку изменить значение самой ссылки нельзя (все действия происходят над переменной, на которую ссылка ссылается). Присвоить значение самой ссылке также невозможно, поэтому ссылку **нельзя описать без инициализации** (здесь также проявляется, что инициализация и присваивание – совершенно разные вещи).

2.3. Ссылки для возвращаемого значения

В Си++ возможно использование ссылочного типа как типа значения, возвращаемого самой функцией

Например, необходимо сначала найти целочисленную переменную в составе сложной структуры данных (переменная может являться полем структуры, которая, в свою очередь, является элементом массива и т.д.), а затем нужно изменить значение найденной переменной. Тогда поиск можно выделить в отдельную функцию `find`, которая возвращает ссылку на искомую целочисленную переменную:

```
int &find(/*...*/);
```

Тогда будут допустимы следующие действия:

```
int a = find(/*...*/) + 5;
find(/*...*/) = 3;
find(/*...*/) += 10;
find(/*...*/)++;
int b = ++find(/*...*/);
```

3. Запрет изменения с помощью `const`

3.1. Ключевое слово `const`

Ключевое слово `const` в языках Си и Си++ является квалификатором, который запрещает изменение значения.

Для простых типов данных `const` можно указывать с любой стороны от названия типа:

```
const int a = 5;    /* Оба эти объявления */
int const a = 5;    /* эквивалентны. */
```

Ключевое слово `const` при объявлении массива запрещает изменять значения элементов массива (такой массив всегда объявляется с инициализацией):

```
const int m[] = {4, 5, 6, 7, 8};    /* Оба эти объявления */
int const m[] = {4, 5, 6, 7, 8};    /* эквивалентны. */
```

При объявлении структуры ключевое слово `const` можно применить ко всей структуре целиком, тогда значения всех ее полей изменять запрещено:

```
struct myst {
    int n;
    double x;
};
 ...
const struct myst a = {.n = 23, .x = 3.14};    /* Оба эти объявления */
struct const myst a = {.n = 23, .x = 3.14};    /* эквивалентны. */
```

Для запрета изменения отдельных полей структуры ключевое слово `const` необходимо указать при описании структуры:

```
struct myst {
    int n;
    const double x;
};
 ...
struct myst a = {.n = 23, .x = 3.14};
```

3.2. Указатели и `const`

Для адресных типов (указателей и ссылок) синтаксис использования `const` усложняется. Действие `const` зависит от места, в которое он помещается в языковой конструкции.

Правила языка Си **для указателей** (действуют также в Си++):

Объявление	Комментарий	Изменение…	
		значения `int`	Указателя
`int *p;`	`p` – указатель на значение типа `int`	`*p = 123;`	`p = NULL;`
`int const *p;`	`p` – указатель на неизменяемое значение типа `int`	`*p = 123;` ошибка!	`p = NULL;`
`int * const p;`	`p` – неизменяемый указатель на значение типа `int`	`*p = 123;`	`p = NULL;` ошибка!
`int const * const p;`	`p` – неизменяемый указатель на неизменяемое значение типа `int`	`*p = 123;` ошибка!	`p = NULL;` ошибка!

Для облегчения восприятия описание указателя следует читать **справа налево**.

Ключевое слово `const` следует указывать **всегда справа** от названия типа данных. В этом случае `const` накладывает запрет на изменение тех данных, тип которых указан **слева** от слова `const`.

Указатель на неизменяемое значение запрещает изменение области памяти по данному адресу и называется ***указатель на константу***. А неизменяемый указатель (`* const`) называется ***константный указатель***.

Чтобы не возникло путаницы при объявлении списка переменных, перечисленных через запятую, символ ***** всегда следует указывать **как можно правее** (присоединяя его к названиям переменных и/или слева к слову `const`):

Запутывающая запись	Понятная запись	Комментарий
`int* p;`	`int *p;`	`p` – указатель
`int* p, a;`	`int *p, a;`	`p` – указатель, `a` – переменная типа `int`
`int* p, *q;`	`int *p, *q;`	`p` и `q` – указатели
`int const* p;`	`int const *p;`	`p` – указатель на константу
`int* const p;`	`int *const p;`	`p` – константный указатель

Можно использовать ключевое слово `const` для указателей на указатели:

```
int a;          /* a - переменная. */
int *p = &a;    /* p - указатель на переменную a. */
 ...                                  /* Запрещено изменять:              */
int const **pp1 = &p;                 /* - значение a;                    */
int const *const *pp2 = &p;           /* - значение a и указатель p;      */
int const *const *const pp3 = &p;     /* - значение a, указатели p и pp.  */
```

3.3. Константные ссылки

В языке Си++ сама ссылка является неизменяемой после ее описания. Поэтому ключевое слово `const` при описании ссылки может быть использовано только, чтобы запретить изменение области памяти по этой ссылке. Такую ссылку на неизменяемое значение называют ***ссылкой на константу***, хотя она может ссылаться на обычную переменную:

```
int a;                  /* a - переменная. */
 ...                    /* r - ссылка на константу a. */
const int &r = a;       /*   Оба эти объявления */
int const &r = a;       /*   эквивалентны. */
 ...
a = 5;    /*  Корректная операция изменения значения переменной. */
r = 7;    /*  Ошибка! Значение по ссылке изменять запрещено. */
```

Зачастую в литературе ссылку на константу называют ***константной ссылкой*** (хотя это не совсем корректно по сравнению с термином **константный указатель**, но такое название утвердилось в языке Си++).

На константу можно ссылаться только константной ссылкой:

```
const int a = 12;    /* a - константа. */
 ...
int &r = a;          /* Ошибка! Неконстантная ссылка на константу a. */
const int &r = a;    /* Корректно: r - ссылка на константу a. */
```

3.4. Константные ссылки при передаче в функцию

Константная ссылка позволяет передавать в функцию в качестве параметра адрес переменной вместо копирования ее значения, при этом изменять значение переменной запрещается (как при обычной передаче **по значению**).

Если передаваемый в функцию параметр представляет собой объект (или структуру), размер которой существенно превышает размер адреса, то ***передача по константной ссылке*** может увеличить скорость работы программы, поскольку не затрачивается время на копирование самого объекта (или структуры).

Для рассматриваемого примера класса комплексных чисел:

```
class Complex {
    double re, im;
public:
    ...
    Complex operator+(Complex op2) {
        return Complex(re + op2.re, im + op2.im);
    }
    ...
};
```

можно использовать константную ссылку для параметра функции-члена `operator+`:

```
    Complex operator+(const Complex &op2) {
        return Complex(re + op2.re, im + op2.im);
    }
```

Тогда вместо копирования двух полей типа `double` произойдет передача адреса существующего объекта и обращение по этому адресу (без возможности изменения значений полей передаваемого объекта).

3.5. Константные функции-члены (методы)

Пусть необходимо передать в функцию адрес объекта, запретив изменения самого объекта. Например, в некоторую функцию `func1` передадим указатель на константный объект типа `Complex`:

```
void func1(Complex const *p) {
    ...
}
```

Тогда из функции `func1` невозможно изменить объект, на который указывает `p`. Такой объект называется ***константным объектом***.

В соответствии с концепцией объекта все его поля скрыты, а взаимодействие с ним возможно только через функции-члены (методы), которые могут изменить внутреннее состояние объекта (то есть значения его скрытых полей). Но константный объект запрещено изменять, поэтому **вызов всех** его **методов запрещен** компилятором. Запрещено вызывать даже те методы, которые не изменяют состояние (скрытые поля) объекта.

Такая же ситуация возникает при передаче (например, в качестве параметра функции `func2`) константной ссылки на объект:

```
void func2(const Complex &r) {
    ...
}
```

Внутри функций `func1` и `func2` получается, что константный объект существует, но вызывать его функции-члены (методы) запрещено, а значит работать с объектом никак невозможно.

Для работы с константными объектами в языке Си++ предусмотрены ***константные функции-члены (методы)***. Они задаются добавлением ключевого слова `const` после заголовка функции-члена (метода), но перед ее телом (т.е. перед символом «`{`»).

В языке Си++ рекомендуется все функции-члены (методы), которые не должны изменять состояние объекта, помечать как **константные**.

Например, для класса комплексных чисел:

```
class Complex {
    double re, im;
public:
    Complex(double re_val, double im_val)
        { re = re_val;  im = im_val; }
    Complex(double re_val)
        { re = re_val;  im = 0; }
    Complex()  { re = 0;  im = 0; }
    double modulo() const  { return sqrt(re*re + im*im); }
    double argument() const  { return atan2(im, re); }
    double get_re() const  { return re; }
    double get_im() const  { return im; }
    Complex operator+(const Complex &op2) const
        { return Complex(re + op2.re, im + op2.im); }
    Complex operator-(Complex op2) const
        { return Complex(re - op2.re, im - op2.im); }
    Complex operator*(Complex op2) const
        { return Complex(re*op2.re - im*op2.im, re*op2.re + im*op2.im); }
    Complex operator/(Complex op2) const {
        double dvs = op2.re*op2.re + op2.im*op2.im;
        return Complex((re*op2.re + im*op2.im)/dvs,
                       (re*op2.re - im*op2.im)/dvs);
    }
};
```

Теперь для класса `Complex` можно вызывать константные функции-члены (методы) из функций:

```
void func1(Complex const *p) {
     ...
     p->modulo();
     ...
}
  ...
void func2(const Complex &r) {
     ...
     r.modulo();
     ...
}
```

Поскольку константной функции-члену (методу) запрещено изменять состояние объекта, то **в теле константной функции-члена (метода) допустим вызов только константных функций-членов (методов)** того же объекта. Это правило можно пояснить так:

> внутри константной функции-члена (метода) класса `myClass` указатель `this` имеет тип `const myClass *` (а не `myClass *`).

Вызов неконстантной функции-члена (метода) отменяет действие ключевого слова `const`, а это делать **запрещено** компилятором.