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

Лекции по информатике и программированию

Лекция 4

Абстрактные типы данных (часть 2)

Содержание

4. Работа с динамической памятью в C++

4.1. Операции `new` и `delete`

Языке Си сам по себе не включает операций работы с динамической памятью. В Си средства работы с динамической памятью вынесены в стандартную библиотеку в виде стандартных функций: `malloc`, `realloc` и `free`. Эти функции непригодны для создания и удаления объектов в динамической памяти, поскольку они не могут вызывать конструкторы и деструкторы.

Более того, в языке Си++ конструкторами и деструкторами оперирует компилятор, поэтому выносить работу с динамической памятью из языка в отдельную библиотеку проблематично. Из-за этого в язык Си++ внесены **операции `new` и `delete`** для создания и удаления объектов в динамической памяти.

Для создания в динамической памяти одиночного объекта используется **скалярная форма** операция `new`. Синтаксис:

```
myClass *p;
p = new myClass;
```

Здесь операция `new`:
- выделяет динамическую память для объекта типа `myClass`;
- вызывает для объекта конструктор по умолчанию;
- возвращает указатель (типа `myClass *`) на созданный объект.

Конструктору объекта можно передать параметры. Например, для описанного класса комплексных чисел:

```
Complex *p;
p = new Complex(2.7, 3.8);
```

или

```
Complex *p = new Complex(2.7, 3.8);
```

Операцию `new` можно использовать и для простых типов данных:

```
int *p;
p = new int;
```

или с инициализацией:

```
int *p = new int(123);
```

Для удаления одиночного объекта из динамической памяти используется скалярная форма операции `delete`:

```
delete p;
```

Операция `delete`:
- вызывает деструктор для объекта, на который указывает `p`;
- освобождает занимаемую объектом динамическую память.

Для создания и удаления динамических массивов используются **векторные формы** операций `new []` и `delete []`. Например:

```
int *p = new int[100];   /* Создание массива из 100 чисел типа int. */
 ...
delete [] p;            /* Удаление всего массива целиком. */
```

При этом конструкторы и деструкторы вызываются **для каждого элемента массива**.

В векторной форме операции `new []` невозможно передать параметры конструкторам элементов массива, поэтому **класс должен обязательно содержать конструктор по умолчанию**.

Объекты, созданные с помощью векторной формы `new []`, необходимо удалять **только** с помощью векторной формы `delete []`.
Объекты, созданные с помощью скалярной формы `new`, необходимо удалять **только** с помощью скалярной формы `delete`.

Также не следует удалять с помощью `delete` объекты, созданные функцией `malloc`. И не следует удалять с помощью `free` объекты, созданные функцией `new`.

4.2. Конструктор копирования

Рассмотрим следующий пример: пусть в конструкторе некоторого класса создается динамический массив, а в деструкторе он уничтожается:

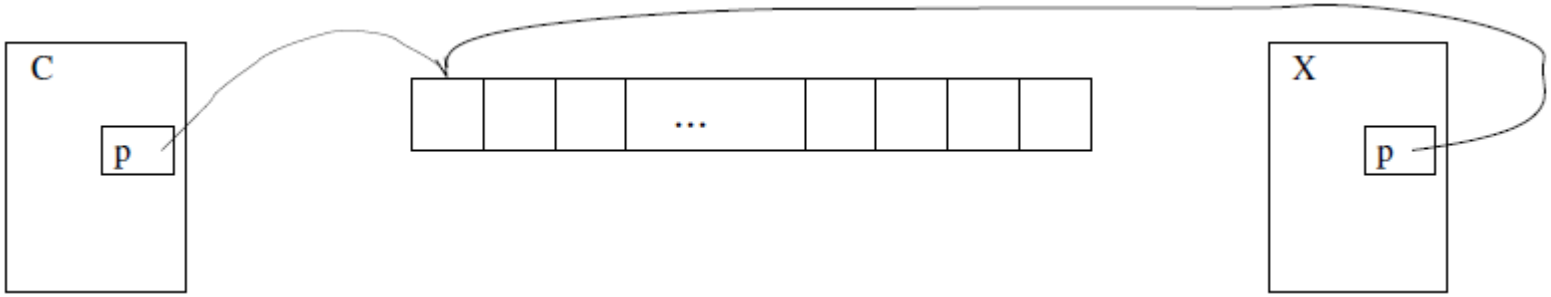
```
class myClass {
    int *p;
public:
    myClass()  { p = new int[20]; }
    ~myClass()  { delete [] p; }
     ...
};
```

Создадим копию объекта класса `myClass`, передав его в функцию в качестве параметра по значению:

```
void func(myClass x) {
    ...
}
int main(void) {
    ...
    myClass c;
    func(c);
    ...
}
```

Тогда в функции `func` локальная переменная `x` (формальный параметр) является копией объекта `c` (фактический параметр, аргумент), созданной путем обычного побитового копирования.

Значит при копировании объекта класса `myClass` оказался скопирован указатель на динамический массив, созданный конструктором объекта `c`. Следовательно, оба объекта (оригинал `c` и его локальная копия `x`) используют один и тот же динамический массив:



Теперь функция `func` получила доступ к изменению состояния объекта `c`, что противоречит концепции ООП.

Кроме того, при выходе из функции `func` отработает деструктор для локального объекта `x`, который уничтожит динамический массив, объект `c` окажется в ошибочном состоянии. В результате чего любое действие с объектом `c` приведет к ошибке. Даже если никаких действий с объектом `c` не предпринимать, ошибка возникнет при его уничтожении (деструктор попытается уничтожить уже несуществующий динамический массив).

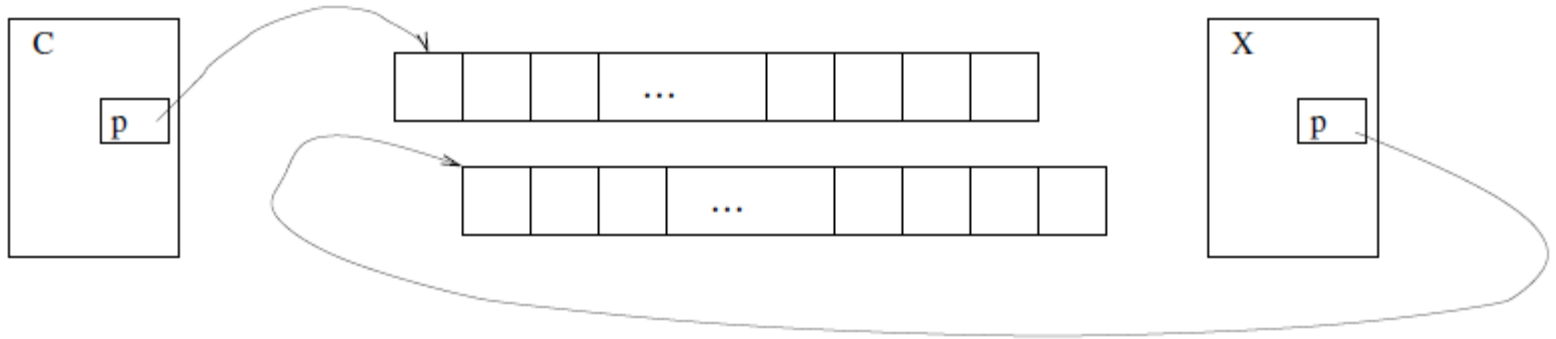
Таким образом для объектов класса `myClass` побитовое копирование непригодно.

Чтобы сообщить компилятору как корректно создать копию имеющегося объекта, в языке Си++ предусмотрен специальный ***конструктор копирования***. Он имеет ровно один параметр, который имеет тип «**ссылка на объект описываемого класса**». Часто эту ссылку делают константной, чтобы показать, что исходный объект не изменится.

Для рассматриваемого примера описания класса `myClass`:

```
class myClass {
    int *p;
public:
    myClass() {
        p = new int[20];
    }
    myClass(const myClass &a) {
        p = new int[20];
        for (int i=0; i<20; i++)
            p[i] = a.p[i];
    }
    ~myClass()  { delete [] p; }
     ...
};
```

Теперь у каждого объекта будет свой динамический массив:



5. Временные и анонимные объекты

5.1. Анонимные объекты

Анонимные объекты не имеют имени и обычно применяются для однократного использования, чтобы код оказался более лаконичным и наглядным. Для этого указывают имя конструктора объекта и список его фактических параметров (возможно, пустой).

Пример – умножение на мнимую единицу для комплексных чисел:

```
Complex t, z;
...
Complex im1(0, 1);       /* Создание именованного объекта im1. */
t = z * im1;             /* Использование именованного объекта im1. */
t = z * Complex(0, 1);   /* Использование анонимного объекта. */
```

Анонимные объекты вводятся **программистом** в явном виде.

5.2. Временные объекты

Временные объекты – это объекты, которые не имеют имени и порождаются **компилятором** без их явного указания.

Простой пример – передача параметра в функцию с помощью конструктора преобразования. Рассмотренный ранее пример – 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 объект типа `Complex`, который передается в функцию `func` в качестве фактического параметра.

Другой пример появления временного объекта:

```
Complex t, a, b, c;
...
t = a + b + c;       /* Использование временного объекта для (a+b). */
```

Здесь сложение выполняется слева направо, т.е. сначала выполнится первая операция (`a+b`), результат которой сохраняется во временном объекте типа `Complex`, а затем выполняется вторая операция сложения этого результата с объектом `c`. Временный объект для хранения результата (`a+b`) создается компилятором.

Компилятор вынужден порождать временные объекты при использовании в выражении **вызова функции, которая возвращает значение типа объект** (причем возврат производится по значению, а не другим способом). В приведенном примере такой функцией является `operator+`.

Анонимные объекты можно считать **частным случаем** временных объектов: анонимные объекты – это временные объекты, которые задаются в коде явно.

5.3. Правила использования

Время жизни временного или анонимного объекта: объект существует до момента окончания вычисления содержащего их выражения, затем уничтожаются с помощью деструктора (если он имеется). Например, в операторе вычисления выражения:

```
t = a + b + c;
```

временный объект существует «до точки с запятой».

Из этого правила имеется одно исключение: если временный или анонимный объект указан в качестве инициализатора ссылки, то он существует до тех пор, пока существует эта ссылка.

На временный или анонимный объект **нельзя ссылаться неконстантной ссылкой**. По этому правилу временный или анонимный объект можно передавать в функцию либо по значению, либо по константной ссылке.

По неконстантной ссылке функция обычно получает только **выходной** параметр, то есть через него функция **возвращает** некоторую информацию, помещать которую во временный или анонимный объект бессмысленно.

Поэтому квалификатор `const` следует использовать для всех ссылок, для которых это возможно. Неконстантные ссылки следует использовать тогда и только тогда, когда функция должна изменять значение переменной, передаваемой в нее в качестве параметра.

6. Значения по умолчанию и конструкторы

6.1. Значения параметров по умолчанию

При объявлении функции (там, где впервые встречается прототип функции) язык Си++ позволяет для всех (или некоторых) ее параметров задать ***значения по умолчанию***. Тогда при вызове функции заданные по умолчанию параметры можно не указывать – компилятор подставит их сам.

Например, для функции с прототипом:

```
void f(int a = 3, const *char b = "string", int c = 5);
```

возможны следующие вызовы:

```
f(5, "name", 10);
f(5, "name");     // эквивалентно f(5, "name", 5);
f(5);             // эквивалентно f(5, "string", 5);
f();              // эквивалентно f(3, "string", 5);
```

По умолчанию может быть задано любое количество параметров функции, но при этом **в списке параметров функции все параметры, следующие за заданным по умолчанию, также должны быть заданы по умолчанию**. Например, корректные описания:

```
void f(int a = 3, const *char b = "string", int c = 5);
void f(int a, const *char b = "string", int c = 5);
void f(int a, const *char b, int c = 5);
```

а следующие описания некорректны:

```
void f(int a = 3, const *char b = "string", int c); // ошибка!
void f(int a = 3, const *char b, int c = 5);        // ошибка!
void f(int a = 3, const *char b, int c);            // ошибка!
void f(int a, const *char b = "string", int c);     // ошибка!
```

Это ограничение упрощает компилятору поиск подходящей функции.

Если при вызове функции указано ***n*** фактических параметров (аргументов), то компилятор сопоставит их с ***n*** **первыми** формальными параметрами.

Выражения, задающие значения по умолчанию, могут стать причиной ошибки. Поэтому в качестве таких выражений следует использовать **только константные выражения** (которые будут вычислены во время компиляции программы, до ее запуска).

6.2. Конструкторы специальных видов

При объявлении функции (там, где впервые встречается прототип функции) язык Си++ позволяет для всех (или некоторых) ее параметров задать ***значения по умолчанию***. Тогда при вызове функции заданные по умолчанию параметры можно не указывать – компилятор подставит их сам.

Ранее введены конструкторы специальных видов:

Конструктор …	**в восприятии компилятора – это конструктор …**
умолчания	без параметров
преобразования	с одним параметром, имеющим тип, отличный от описываемого
копирования	с одним параметром типа «ссылка на объект описываемого класса»

Использование параметров по умолчанию заставляет компилятор воспринимать как конструктор специального вида такой конструктор, который **допускает вызов** с соответствующими параметрами.

Например, в классе `Complex` достаточно описать всего один конструктор, который будет служить одновременно и конструктором умолчания, и конструктором преобразования, и обычным конструктором от двух аргументов:

```
class Complex {
    double re, im;
public:
    Complex(double re_val = 0, double im_val = 0)
        { re = re_val;  im = im_val; }
    double modulo() ...
    ...
};
```

Такой конструктор может быть вызван как:

```
Complex c;       // конструктор умолчания, эквивалентно Complex c(0, 0);
Complex c(4.7); // конструктор преобразования, эквивалентно Complex c(4.7, 0);
Complex c(2.7, 3.8); // обычный конструктор.
```

7. Класс как область видимости

7.1. Описание функций-членов (методов) вне класса

Тела функций-членов (методов) класса могут содержать значительный объем кода. В этом случае описание такого класса затрудняется для восприятия программистом: поиск заголовков функций-членов (методов) сопровождается долгим перелистыванием с необходимостью запоминать найденные заголовки. Из-за этого изучение класса становится утомительным занятием.

Проблема решается вынесением тел функций-членов (методов) за пределы описания класса. В описании класса остаются только прототипы функций-членов (методов). Такое описание класса называется ***заголовком класса***.

Прототип функции-члена (метода) содержит:

- тип возвращаемого значения;
- имя метода;
- список формальных параметров;
- ключевое слово `const` (если функция-член (метод) является константной),
- символ «точка с запятой» (`;`) в конце (как у обычного прототипа).

При этом тело функции-члена (метода) описывается в другом месте (за пределами заголовка класса) с помощью лексемы ***раскрытия области видимости*** – «двойного двоеточия» (`::`), которую иногда называют «четвероточием». Пример:

```
class MyCl { // Заголовок класса MyCl.
    ...
public:
    MyCl();                          // Прототип конструктора.
    void f(int a, int b);            // Прототип метода f.
    int g(const char *str) const;    // Прототип константного метода g.
};

...

MyCl::MyCl() {
    ...  // Тело конструктора.
}

void MyCl::f(int a, int b) {
    ...  // Тело метода f.
}

int MyCl::g(const char *str) const {
    ...  // Тело константного метода g.
}
```

Класс (или структура) представляет собой отдельную ***область видимости***. Раскрытие области видимости (`::`) позволяет сделать доступными вне класса **имена членов класса**, которые не закрыты защитой. Если член класса `MyCl` внутри класса доступен по имени `f`, то снаружи он доступен по имени `MyCl::f`.

Если у двойного двоеточия опустить имя области видимости (левую часть), то подразумевается ***глобальная область видимости***.

В рассмотренном примере можно создать обычную функцию с именем `f` (не относящуюся к классу `MyCl`), и это не вызовет конфликта имен:

```
class MyCl {
    ...  // Описание заголовка класса MyCl.
};
...
void f(double x) {
    ...  // Тело глобальной (обычной) функции f.
}
...
void MyCl::f(int a, int b) {
    ...  // Тело метода f класса MyCl.
}
int MyCl::g(const char *str) const {
    ...  // Тело метода g класса MyCl.
    f(7, 5);     // Вызов метода f класса MyCl.
    ::f(3.14);  // Вызов глобальной функции f (из метода g класса MyCl).
    ...
}
```

Если функция-член (метод) вызывается для объекта, то ее имя находится в области видимости класса, к которому принадлежит объект, а значит явно раскрывать область видимости (с помощью двойного двоеточия) не требуется:

```
MyCl my;  // Создание объекта my класса MyCl.
...
my.f(7, 5);  // Вызов метода f для объекта my класса MyCl.
```

Лексема раскрытия области видимости (`::`) **не является операцией**, поскольку слева и справа от двойного двоеточия указываются имена, но не выражения.

7.2. Подставляемые функции (`inline`)

Для программ, использующих ООП и АТД, при генерировании машинного кода компилятор некоторые функции-члены (методы) считает подставляемыми (`inline`), хотя ключевое слово `inline` может быть явно не указано. Для такой функции-члена компилятор вместо обычного вызова подставляет машинный код тела функции-члена. Поэтому на уровне машинного кода никаких вызовов функций-членов (методов) не происходит – код остается таким же эффективным, как если бы мы не использовали классы.

Такое поведение компилятора имеет смысл лишь для функций с очень коротким телом.

Компилятор Си++ пытается обрабатывать как **подставляемые** все функции-члены (методы), тела которых описаны непосредственно в заголовке класса.

С помощью ключевого слова `inline` можно предложить компилятору рассматривать любую функцию-член (метод) как подставляемую:

```
inline void MyCl::f(int a, int b) {
    ...  // Тело метода f класса MyCl.
}
```

Однако, ни описание функции-члена (метода) в заголовке класса, ни явное указание ключевого слова `inline` ни к чему компилятор не обязывает, это лишь рекомендация программиста. Способ обработки функций **определяется компилятором** для повышения эффективности машинного кода.

Но все-таки о подставляемых функциях стоит помнить, чтобы не бояться потерять эффективность из-за вызова функций с короткими телами.

7.3. Инициализация членов класса в конструкторе

Рассмотрим ситуацию, когда полем класса является другой класс, у которого отсутствует конструктор по умолчанию:

```
class MyA { // Заголовок класса MyA.
    ...
public:
    MyA(int a, int b) { /* Тело конструктора класса MyA. */ }
    ...
};
class MyB { // Заголовок класса MyB.
    MyA a;
    ...
public:
    MyB();  //* Прототип конструктора по умолчанию класса MyB.
    ...
};
```

Для конструктора по умолчанию класса `MyB` уже доступно поле `a`. Но для этого заранее должен был отработать конструктор класса `MyA`, который требует параметров.

Чтобы передать эти параметры в языке Си++ имеется возможность ***инициализации полей объекта***. При описании конструктора:

```
MyB::MyB() : a(5, 3)  { /* Тело конструктора класса MyB. */ }
```

При выполнении этого кода компилятор вызывает конструктор класса `MyA` с указанными параметрами в начале тела конструктора `MyB`.

Подобным образом можно провести инициализацию любых полей, а не только полей типа класс. 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)  {}
    ...
};
```

Инициализаторы полей в списке после двоеточия должны следовать **в том же порядке, в котором сами поля описаны в классе**.