Введение в объектно-ориентированное программирование (на примере языка программирования C++)

Содержание

1 Обзор методов программирования **2**
1.1 Неструктурированное программирование . . . 2
1.2 Процедурное программирование . . . 3
1.3 Модульное программирование . . . 3
1.4 Использование структур данных (пример) . . . 4
1.4.1 Обработка однонаправленного связанного списка . . . 4
1.4.2 Обработка нескольких списков . . . 5
1.5 Недостатки модульного программирования . . . 6
1.5.1 Явное создание и удаление . . . 6
1.5.2 Отделение данных от операций . . . 7
1.5.3 Отсутствие контроля типов . . . 9
1.6 Объектно-ориентированное программирование . . . 9

2 Абстрактные типы данных **10**
2.1 Подход к решению задач . . . 10
2.2 Свойства абстрактных типов данных . . . 11
2.3 Родовые абстрактные типы данных . . . 13
2.4 Один из способов описания АТД . . . 14
2.5 Абстрактные типы данных и объектный подход . . . 15

3 Концепции объектно-ориентированного подхода **15**
3.1 Реализация абстрактных типов данных . . . 15
3.2 Класс . . . 16
3.3 Объект . . . 17
3.4 Сообщение . . . 17
3.5 Заключение . . . 18
3.6 C++: Основные расширения . . . 19
3.6.1 Типы данных . . . 19
3.6.2 Функции . . . 20
3.7 C++: Первое объектно-ориентированное расширение . . . 21
3.7.1 Классы и объекты . . . 21
3.7.2 Конструкторы . . . 24
3.7.3 Деструкторы . . . 25
3.8 C++: Перегрузка операторов . . . 26
3.9 Отношения . . . 28
3.10 Наследование . . . 30
3.11 Множественное наследование . . . 32
3.12 Абстрактные классы . . . 34

3.13 C++: Наследование . . . 35
3.13.1 Виды наследования . . . 35
3.13.2 Создание объектов . . . 36
3.13.3 Удаление объектов . . . 37
3.13.4 Множественное наследование . . . 37
3.14 C++: Абстрактные классы . . . 38
3.15 C++: Друзья . . . 38
3.16 Статическое и динамическое связывание . . . 38
3.17 Полиморфизм . . . 40
3.18 C++: Полиморфизм . . . 43
3.19 Родовые типы данных . . . 44

1 Обзор методов программирования

Основные подходы к программированию:

- Неструктурированное программирование
- Процедурное программирование
- Модульное программирование
- Объектно-ориентированное программирование

Начиная с раздела 1.1 и до раздела 1.3 кратко описываются первые три подхода к программированию. Далее приводится простой пример модуля, реализующего однонаправленный связанный список (раздел 1.4). На этом примере будет показан ряд проблем, возникающих при использовании модульного программирования (раздел 1.5). В разделе 1.6 даётся понятие объектно-ориентированного подхода.

1.1 Неструктурированное программирование

При использовании данного подхода "основная программа" состоит из последовательности команд или *операторов*, которые модифицируют данные, которые, в свою очередь, являются *общими* для всей программы. Это проиллюстрировано на Рис. 1.

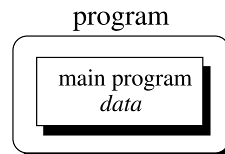

Рис. 1: Неструктурированное программирование. Основная программа непосредственно оперирует общими данными.

Этот подход имеет недостатки при разработке достаточно больших по объему программ. Например, если одну и ту же последовательность операторов требуется выполнить в различных местах программы, то эта последовательность операторов должна копироваться в соответствующее место этой программы. Это приводит к необходимости *выделять* такие последовательности операторов, присваивать им имена и предоставлять методы вызова и возврата из получившихся таким образом *процедур*.

1.2 Процедурное программирование

При использовании процедурного подхода некоторая законченная последовательность операторов оформляется в виде отдельной *процедуры*. *Вызов процедуры* используется для передачи управления процедуре. После того, как процедура будет выполнена, управление передается на следующий за оператором вызова оператор основной программы (Рис. 2).

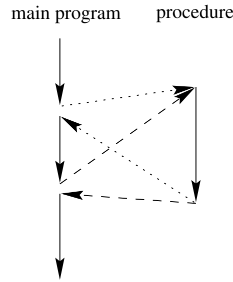

Рис. 2: Выполнение процедур.

С введением *параметров* и процедур (*подпрограмм*) программа становится более структурированной и свободной от ошибок. Например, если процедура является корректной, то каждый раз, когда она используется, процедура возвращает корректный результат. Следовательно, в случае наличия ошибок достаточно просто локализовать место, в котором программа работает некорректно.

При использовании процедурного подхода программа рассматривается как последовательность вызовов процедур. Основная программа отвечает за передачу необходимых данных для процедур, данные обрабатываются процедурами и, после обработки, основная программа выводит результаты. Таким образом, *поток данных* может быть представлен в виде *дерева* (Рис. 3).

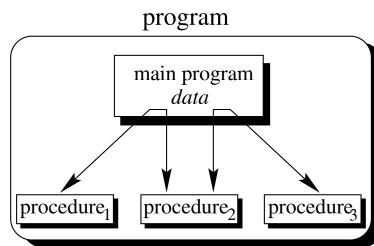

Рис. 3: Процедурное программирование. Основная программа координирует вызовы процедур и передаёт необходимые данные в виде параметров.

Итак, при процедурном подходе единая программа делится на небольшие части, называемые процедурами. Для того, чтобы обеспечить использование основных процедур или групп процедур также и в других программах, эти процедуры должны быть доступны раздельно. Программирование с использованием модульного подхода позволяет группировать процедуры в модулях.

1.3 Модульное программирование

При использовании модульного программирования процедуры, обеспечивающие некоторую общую функциональность, группируются вместе в отдельных *модулях* (Рис. 4).

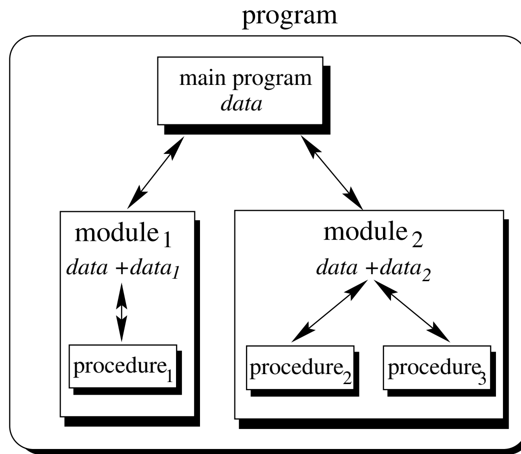

Рис. 4: Модульное программирование. Основная программа координирует вызовы процедур из отдельных модулей и передаёт необходимые данные в виде параметров.

Каждый модуль может обладать своими собственными данными. Это позволяет каждому модулю управлять внутренним *состоянием*, которое модифицируется вызовами процедур этого модуля. Причем, в каждый момент времени каждый модуль находится в одном определённом состоянии и присутствует в одном единственном экземпляре по отношению к определённой программе.

1.4 Использование структур данных (пример)

Для хранения данных программы используют структуры данных. Существуют различные структуры данных, например: списки, деревья, массивы, множества и т.п. Каждая структура данных (тип данных) характеризуется собственно *структурой* и *методами доступа*. Другими словами, тип данных характеризуется множеством возможных значений D, которые могут принимать переменные этого типа, и множеством операций F, определённых для данного множества значений.

1.4.1 Обработка однонаправленного связанного списка

Однонаправленный связанный список представлен на Рис. 5.

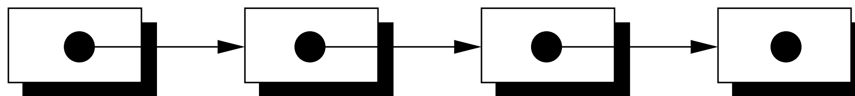

Рис. 5: Структура однонаправленного связанного списка.

Список представляет собой общую структуру данных и, следовательно, будет реализован в виде *модуля*. Это требует создания двух файлов: в одном будет содержаться *описание интерфейса*, а в другом собственно *описание реализации*.

Для описания используется простейший псевдо-код.

Описание интерфейса может выглядеть следующим образом:

```
MODULE Singly-Linked-List-1
```

```
BOOL list_initialize();
/* инициализация локальных переменных */
BOOL list_append(ANY data);
/* добавление элемента в список */
BOOL list_delete();
/* удаление элемента из списка */
     list_end();
     /* удаление списка */

ANY list_getFirst();
/* получение первого элемента списка */
ANY list_getNext();
/* получение следующего элемента списка */
BOOL list_isEmpty();
/* проверка списка на пустоту */

END Singly-Linked-List-1
```

Определения, представленные в интерфейсной части, описывают *что* доступно и не описывают *как* это реализуется. Таким образом осуществляется *скрытие* реализации.**Скрытие реализации** является фундаментальным принципом разработки программного обеспечения и позволяет скрыть детали реализации той или иной процедуры, что, в свою очередь, позволяет изменять реализацию не изменяя других модулей программы, поскольку формат вызова процедуры остается неизменным.

При таком определении, просмотр и обработка каждого элемента списка представляется в виде следующего цикла:

```
ANY data;

data <- list_getFirst();
WHILE data IS VALID DO
    doSomething(data);
    data <- list_getNext();
END
```

1.4.2 Обработка нескольких списков

Для обработки нескольких списков одновременно необходимо изменить описание интерфейса и, соответственно, реализацию однонаправленного связанного списка. В этом случае в описание интерфейса включается определение *управляющего идентификатора списка*. Этот идентификатор используется в каждой процедуре для уникальной идентификации конкретного списка. В этом случае определение интерфейса выглядит следующим образом:

```
MODULE Singly-Linked-List-2

DECLARE TYPE list_handle_t;

list_handle_t list_create();
              list_destroy(list_handle_t this);
BOOL          list_append(list_handle_t this, ANY data);
ANY           list_getFirst(list_handle_t this);
ANY           list_getNext(list_handle_t this);
```

```
BOOL          list_isEmpty(list_handle_t this);

END Singly-Linked-List-2;
```

Оператор *DECLARE TYPE* вводит новый тип *list_ handle_ t*, который представляет собой управляющий идентификатор списка. Его представление и реализация здесь не приводятся.

При таком представлении можно создавать *объекты типа список*. Каждый такой объект уникально идентифицируется управляющим идентификатором и, соответственно, для него выполняются только те *методы*, которые действительны именно для этого идентификатора.

1.5 Недостатки модульного программирования

1.5.1 Явное создание и удаление

В приведенном примере для каждого списка необходимо явно объявлять его управляющий идентификатор и осуществлять вызов метода (процедуры) *list_ create()* для того, чтобы получить значение этого идентификатора. После обработки списка необходимо явно вызывать метод (процедуру) *list_ destroy()* и передавать ему в качестве параметра соответствующий управляющий идентификатор.

Пример использования списка в процедуре:

```
PROCEDURE f() BEGIN
    list_handle_t myList;
    myList <- list_create();

    /* Обработка списка myList */
    ...

    list_destroy(myList);
END
```

Использование обычной переменной не требует явных вызовов процедур создания и удаления для этой переменной. То есть такая переменная создается и уничтожается автоматически.

В связи с этим хотелось бы обеспечить подобное поведение и для списка. Например:

```
PROCEDURE f() BEGIN
    list_handle_t myList;
    /* Создание и инициализация списка */

    /* Обработка списка */
    ...
 END /* список уничтожен */
```

Преимущество состоит в том, что теперь компилятор заботится о вызове процедур инициализации и уничтожения для соответствующего объекта. Например, в этом случае можно быть увереным, что список корректно уничтожается, возвращая при этом ресурсы памяти программе.

1.5.2 Отделение данных от операций

Отделение данных от операций обычно базируется на алгоритмическом подходе, при котором структурируются именно операции, а не данные, а именно: модули объединяют общие операции (такие как *list_ ...()*. Затем в эти операции явно передаются данные, требующие обработки. Соответственно, структура модуля ориентирована скорее на операции, чем на данные. Другими словами, операции определяют используемые ими данные.

Пример описания и использования модуля(см. также рисунок 6).

```
MODULE Singly-Linked-List-2

    DECLARE TYPE list_handle_t;

    list_handle_t list_create();
                  list_destroy(list_handle_t this);
    BOOL          list_append(list_handle_t this, ANY data);
    ANY           list_getFirst(list_handle_t this);
    ANY           list_getNext(list_handle_t this);
    BOOL          list_isEmpty(list_handle_t this);

END Singly-Linked-List-2;


PROGRAM My-List

    list_handle_t list01, list02;

    list01 <- list_create();
    list02 <- list_create();

    list_append(list01, data01);
    list_append(list02, data02);
    ...
    list_destroy(list01);
    list_destroy(list02);
END My-List
```

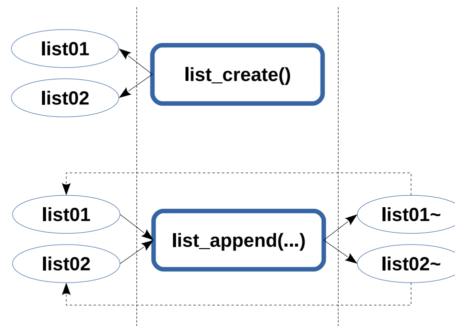

Рис. 6: Отделение данных от операций.

Пояснение на примере языка программирования С(см. также рисунок 7). Предположим, что в системе программирования С отсутствует операция

++ для целых чисел. Тогда, используя модульный подход, такая операция могла бы быть реализована с помощью следующей функции:

```
void plusplus(int * par){
    *par = *par+1;
    return;
}
```

Использование такой функции выглядело бы следующим образом:

```
int main(){
    int a, b:

    a = 33;
    plusplus(&a);

    b = 55;
    plusplus(&b);
}
```

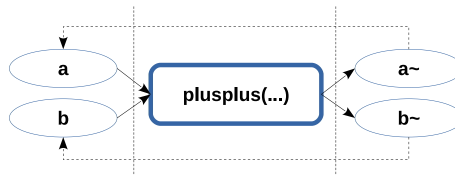

Рис. 7: Отделение данных от операций на примере языка программирования С.

При использовании объектного подхода структурирование основывается на данных. То есть выбирается такое представление данных, которое наилучшим образом удовлетворяет поставленной задаче. В этом случае программы структурируются по данным, а не по операциям, то есть данные определяют набор необходимых операций, которые неотрывны от самих данных (см. рисунки 8, 9).

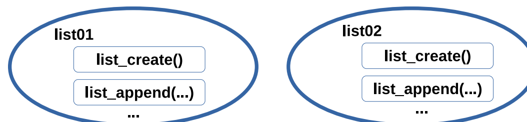

Рис. 8: Данные и операции неотрывны друг от друга.

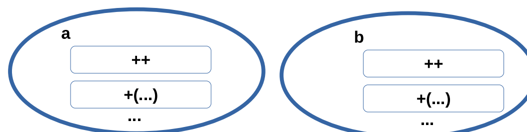

Рис. 9: Данные и операции неотрывны друг от друга (на примере языка программирования С).

1.5.3 Отсутствие контроля типов

В приведённом примере списка использовался специальный тип данных ANY, который позволяет обрабатывать списки с элементами любых типов. В этом случае компилятор не гарантирует обеспечение контроля типов. Например:

```
PROCEDURE f() BEGIN
    SomeDataType data1;
    SomeOtherType data2;
    list_handle_t myList;

    myList <- list_create();
    list_append(myList, data1);
    list_append(myList, data2); /* Ошибка */

    ...

    list_destroy(myList);
END
```

В этом случае ответственность за правильное использование типов полностью лежит на программисте.

В связи с этим было бы желательно иметь механизм, который позволял бы специфицировать тип данных, для которого данный список создается. Причём, все функции обработки списка остались бы те же самые, чтобы в списке не хранилось: числа, строки символов или ещё что-то, и, в этом случае объявление списка могло бы выглядеть, например, следующим образом:

```
list_handle_t<String> list1;
/* список строк */
list_handle_t<Number> list2;
/* список чисел */
```

Соответствующие процедуры обработки списка автоматически возвращали бы правильный тип данных. Компилятор также мог бы проверить корректность использования типов данных.

1.6 Объектно-ориентированное программирование

Объектно-ориентированное программирование решает некоторые из упомянутых выше проблем. В отличие от других подходов, объектно-ориентированный подход предполагает наличие сети взаимодействующих *объектов*, причём каждый из них обладает своим собственным состоянием (Рис. 10).

В отличие от вышеприведенного примера, в котором использовался модульный подход к программированию, при использовании объектно-ориентированного подхода возможно существование стольких объектов типа список, сколько необходимо. Вместо вызова процедуры, которая обеспечивает корректную обработку списка, объекту посылается соответствующее сообщение, в ответ на которое объект изменяет своё состояние. Другими словами, каждый объект реализует свой собственный модуль, позволяя сосуществовать многим объектам одновременно.

Каждый объект сам отвечает за корректную инициализацию и уничтожение себя самого. То есть исчезает необходимость явного вызова процедур создания или уничтожения соответствующего объекта.

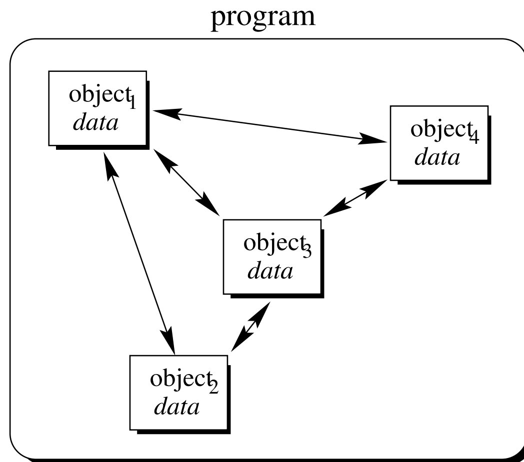

Рис. 10: Объектно-ориентированное программирование. Объекты программы взаимодействуют посылая сообщения друг другу.

2 Абстрактные типы данных

2.1 Подход к решению задач

Первое, с чем сталкиваются разработчики программ — *задача*. Обычно приходится сталкиваться с задачами из "реальной жизни". Однако такие задачи, как правило, являются нечёткими и первое, что необходимо сделать — это постараться понять постановку задачи и отделить несущественные элементы задачи от существенных. То есть необходимо получить абстрактное представление или *модель* задачи. Такой процесс моделирования называется *абстрагированием* (см. Рис. 11).

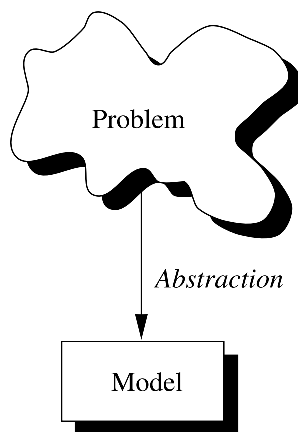

Рис. 11: Создание модели задачи с использованием абстрагирования.

Модель определяет абстрактное (формализованное) представление задачи. То есть модель фокусируется только на существенных сторонах задачи и определяет *свойства* задачи. Эти свойства включают в себя:

- *данные*, которые подлежат обработке, и
- *операции*, которые осуществляют обработку.

Предположим, что необходимо разработать программу для управления персоналом в какой-нибудь организации. Возникают следующие вопросы: какая информация о сотрудниках необходима и какие задачи должна решать разрабатываемая программа?

Сотрудник может быть охарактеризован множеством свойств, в частности:

- имя,
- дата рождения,
- пол,
- серия и номер паспорта,
- номер комнаты,
- цвет волос,
- хобби.

Не все из этих свойств необходимы для решения административных задач. Только некоторые из них являются *относящимися к задаче*. Следовательно необходимо создать модель сотрудника для решения этой задачи. Эта модель включает в себя свойства, которые необходимы для выполнения требований администрирования, например, имя, дата рождения и паспортные данные. Эти свойства называются *данными* модели (сотрудника). Таким образом создаётся описание реального человека с помощью абстракции сотрудника.

Также должны быть определены операции, с помощью которых происходит управление абстрактным сотрудником. Например, должны быть операции, которые позволяют создавать нового сотрудника, которого принимают на работу. То есть, должны быть определены операции, которые воздействуют на абстрактного сотрудника. Данные о сотруднике должны быть доступны только через использование определённого набора операций, что обеспечивает непротиворечивость информации. Например, необходимо проверять является ли введённая дата рождения реальной.

Итак, абстракция есть структурирование нечёткой задачи в хорошо определённые сущности посредством определения соответствующих этим сущностям данных и операций. В результате, эти сущности *объединяют* данные и операции, которые **не** отделены друг от друга.

2.2 Свойства абстрактных типов данных

Пример в предыдущем разделе показывает, что с помощью абстракции создаётся хорошо определённая сущность, которая может быть обработана должным образом. Такого рода сущности определяют *структуру данных* множества элементов. Например, каждый сотрудник описывается именем, датой рождения и паспортными данными.

Доступ к структуре данных может осуществляться посредством определённых *операций*. Такой набор операций называется *интерфейсом* и говорят, что он *экспортируется* сущностью. Сущность, обладающая названными свойствами, а именно, структурой данных и множеством соответствующих операций, называется *абстрактным типом данных* (АТД).

На рисунке 12 показан АТД, который состоит из абстрактной структуры данных и операций. Только операции являются доступными из вне и они же определяют интерфейс.

Как только будет “создан” новый сотрудник, структура данных заполняется актуальными значениями, и, таким образом, создаётся *экземпляр* абстрактного сотрудника и количество таких экземпляров зависит от того, скольких сотрудников необходимо описать в программе управления персоналом.

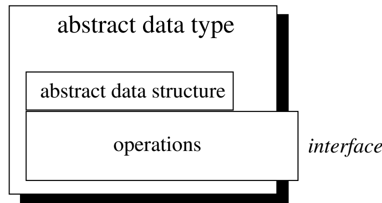

Рис. 12: Абстрактный тип данных (АТД).

Определение 2.2.1 (Абстрактный тип данных) Абстрактный тип данных *(АТД) характеризуется следующими свойствами:*

1. *Экспортирует* **тип**.
2. *Экспортирует* **множество операций**. *Это множество операций называется* **интерфейсом**.
3. *Операции интерфейса являются единственным механизмом доступа к структуре данных абстрактного типа.*
4. *Аксиомы и предусловия определяют область приложений абстрактного типа данных.*

Первое свойство позволяет создавать более чем один экземпляр АТД. Обратимся опять к примеру из раздела 1. В первой версии этого примера список был реализован в виде модуля и можно было использовать только один единственный список в каждый момент времени. Во второй версии вводится "управляющий идентификатор", который используется как ссылка на "объект список". Управляющий идентификатор совместно с определёнными в модуле списка операциями представляет собой АТД *Список*:

1. Использование управляющего идентификатор определяет соответствующую переменную типа *Список*.

2. Интерфейс экземпляра типа *Список* определяется в файле определения интерфейса.

3. Поскольку файл определения интерфейса не включает представления управляющего идентификатора, то этот модификатор не может быть модифицирован непосредственно из программы пользователя.

4. Область приложения определяется семантическим значением предоставляемых операций. Аксиомы и предусловия включают такие предложения, как

   - "Пустой список является списком."
   - "Если l=(d1, d2, d3, ..., dN) является списком, то результатом операции l.append(dM) является список l=(d1, d2, d3, ..., dN, dM)."
   - "Первый элемент списка может быть удалён только в том случае, если список не пуст."

Однако, все эти свойства действительны только благодаря нашему пониманию и дисциплине использования модуля, реализующего список. Ответственность за использование экземпляров *Списка* в соответствие с этими правилами лежит на программисте.

Важность инкапсуляции структуры данных

Принцип, при котором обеспечивается скрытие структуры данных и доступ к данным этой структуры обеспечивается только посредством хорошо определённого интерфейса (операций) называется *инкапсуляцией*.

Рассмотрим математический пример, в котором определяется АТД для обработки комплексных чисел. Как известно, комплексные числа состоят из двух частей: *действительной части* и *мнимой части*. Обе части представляются в виде вещественных чисел. Для комплексных чисел определён ряд операций, в частности: сложение, вычитание, умножение, деление. В качестве аксиом и предусловий используются математические определения комплексных чисел.

Для представления комплексного числа необходимо определить структуру данных, которая будет использоваться в АТД. Можно предложить по крайней мере две таких структуры данных, а именно:

- Обе части комплексного числа хранятся в виде двух-элементного массива c, причём первый элемент этого массива соответствует действительной части комплексного числа, а второй — мнимой части. Если обозначить действительную часть как x, а мнимую — y, то доступ к этим частям с использованием массива будет выглядеть следующим образом: x=c[0] и y=c[1].
- Обе части комплексного числа хранятся в виде структуры (записи), состоящей из двух полей. Если действительной части комплексного числа соответствует поле r этой структуры, а мнимой части — поле i, то значения x и y могут быть получены следующим образом: x=c.r и y=c.i.

Пункт 3 в определении АТД гласит, что для какого-либо обращения к структуре данных должна быть определена соответствующая операция. Приведённый выше пример доступа к структуре данных противоречит этому условию.

Например, сложение двух комплексных чисел требует выполнения операции сложения для каждой части комплексного числа. Следовательно, в зависимости от используемой версии представления структуры данных будет выбираться тот или иной способ доступа. Если же *инкапсулировать* (включить) операцию “add” в определение АТД как часть интерфейса, то в контексте приложения просто используется действие “сложить два комплексных числа” и не приходится задумываться о том, как в действительности эта функциональность реализуется.

Созданный таким образом АТД для комплексных чисел, назовём его *Complex*, может быть использован точно также, как и любые из встроенных в систему программирования типов данных, например, целые, вещественные и т.п.

2.3 Родовые абстрактные типы данных

Абстрактные типы данных используются для определения новых типов из которых создаются экземпляры этих типов данных (переменные). Как видно из примера списка, иногда эти экземпляры могли бы оперировать также и над другими типами данных. Например, можно было бы создать как список целых чисел, так и список строк символов, и даже список списков. В любом из этих случаев семантическое определение списка остаётся одним и тем же. Изменяется только тип элементов, обработка которых требуется в данный конкретный момент.

Эта дополнительная информация о типе элементов могла бы быть задана посредством *родового параметра*, который задаётся во время создания соответ-

ствующей сущности. Таким образом сущность *родового АТД* в действительности представляет собой сущность определённого варианта этого АТД. Например, список строк символов мог бы быть объявлен следующим образом:

```
List<String> listOfStrings;
```

Здесь в угловых скобках заключён тип данных для которого создаётся вариант родового АТД *List*. Сущность *ListOfStrings* предоставляет тот же самый интерфейс, что и любой другой список, но оперирует сущностями тип *Strings*.

2.4 Один из способов описания АТД

Описание любого АТД состоит из двух частей:

- **Данные**: Эта часть описывает структуру данных, используемую в АТД.
- **Операции**: Эта часть описывает операции, относящиеся к этому АТД, а, следовательно, и интерфейс этого АТД. Помимо прочих, задаются специальные операции: **constructor** (конструктор) и **destructor** (деструктор). Конструктор описывает действия, выполняемые при создании сущности этого АТД, а деструктор описывает действия, выполняемые при уничтожении сущности. Для каждой операции описываются её *аргументы*, а также задаются *предусловия* и *постусловия*.

Рассмотрим пример описания АТД *Integer* (целое). Обозначим через k любое выражение, результатом которого является целое число. В этом случае описание АТД *Integer* выглядит следующим образом:

ADT *Integer* is

Данные

Последовательность цифр, перед которой может стоять знак плюс или минус. Обозначим такое целое число как N.

Операции

constructor Создаёт новое целое.

add(k) Создаёт новое целое, которое является суммой N и k. Следовательно, *пост-условием* этой операции является условие $sum = N+k$. В данном случае знак “=” означает знак равенства, а не операцию присваивания.

sub(k) Подобно операции *add*, эта операция создаёт новое целое, которое является разностью двух целых значений. Поэтому постусловием для этой операции является условие $sum = N$-k.

set(k) Устанавливает N равным k. Пост-условием для этой операции является условие $N = k$.

...

end

Приведённое выше описание представляет собой *спецификацию* АТД *Integer*.

Замечание по поводу использования “add” и “+”.

2.5 Абстрактные типы данных и объектный подход

АТД позволяют создавать сущности с хорошо определёнными свойствами и поведением. В объектном подходе АТД представляются в виде *классов*. В объектно-ориентированном подходе класс определяет свойства *объектов*. То есть понятие объекта и понятие сущности с точки зрения ООП тождественны.

АТД определяют функциональность акцентируя внимание на используемых данных, их структуре, операциях, а также аксиомах и пред-условиях. Объектно-ориентированный подход объединяет функциональность различных АТД для решения конкретной задачи. Благодаря этому сущности (объекты) абстрактных типов данных (классов) динамически создаются, используются и уничтожаются.

3 Концепции объектно-ориентированного подхода

3.1 Реализация абстрактных типов данных

Абстрактные типы данных представляют собой абстрактное представление, позволяющее определить свойства множества сущностей. Объектно-ориентированные языки программирования должны предоставлять инструмент для *реализации* этих типов данных. Как только АТД реализован, то можно говорить, что доступно определённое представление этого АТД.

Рассмотрим снова АТД *Integer*. В таких языках программирования, как Pascal, C, Modula-2 уже имеется реализация этого типа, то есть этот тип данных уже реализован в соответствующей системе программирования. Как только создаётся переменная этого типа, то можно использовать относящиеся к этому типу данных операции. Например:

```
int i, j, k;    /* Определяются три целых */

i = 1;          /* Присвоить значение 1 переменной i */
j = 2;          /* Присвоить значение 2 переменной j */
k = i + j;      /* Присвоить значение суммы i и j переменной k */
```

Покажем связь между приведённым фрагментом и АТД *Integer*. В первой строке определяется три экземпляра АТД *Integer*, а именно переменные i, j и k. Для каждого из этих экземпляров должна быть вызвана специальная операция *конструктор*. В приведённом примере эта операция вызывается компилятором. Компилятор резервирует память под значение целого и “связывает” с ней соответствующее имя переменной. Например, обращение к переменной i в действительности означает обращение к области памяти, которая была “сконструирована” (создана) посредством определения переменной i. В принципе, компилятор мог бы и инициализировать выделенную память значением 0.

Следующая строка

```
i = 1;
```

устанавливает значение переменной i в 1. С точки зрения описания АТД эта строка может быть описана следующим образом:

Выполнить операцию установить *с аргументом* 1 *для экземпляра* i *АТД* Integer. *Это записывается следующим образом:* i.set(1).

Таким образом имеется два уровня представления. Первый уровень является

уровнем формального описания поведения экземпляра АТД при использовании определённой операции. Для формального описания используются предусловия и постусловия. В следующем примере эти условия заключены в фигурные скобки.

{ Пред-условие: $i = n$, где $n \in$ Integer }
i.set(1)
{ Пост-условие: $i = 1$ }

Второй уровень является уровнем реализации. На этом уровне выбирается реальное *представление* для операции. В языке программирования C символ "=" реализует операцию *set()*. Однако, в языке программирования Pascal выбрано следующее представление:

```
i := 1;
```

Рассмотрим следующий оператор:

```
k = i + j;
```

Очевидно, что знак "+" выбран для реализации операции *add*. То есть выражение "i + j" может интерпретироваться следущим образом: "прибавить к значению i значение j". На уровне описания АТД это будет представлено в виде:

{ Пред-условие: $i = n_1$ и $j = n_2$, где $n_1, n_2 \in$ Integer }
i.add(j)
{ Пост-условие: $i = n_1$ и $j = n_2$ }

Пост-условие определяет, что i и j не изменяют своих значений. Спецификация операции *add* определяет, что при выполнении этой операции создаётся **новое** целое (новый объект Integer), значения которого представляет собой сумму соответствующих значений. Следовательно, необходимо обеспечить механизм доступа к этому новому экземпляру. Это может быть сделано с помощью операции *set*, применённой к экземпляру k:

{ Пред-условие: $k = n$, где $n \in$ Integer }
k.set(i.add(j))
{ Пост-условие: $k = i + j$ }

Во многих языках программирования выбирается представление, которое совпадает с математической записью, используемой в пред- и пост-условиях.

3.2 Класс

Класс является реальным *представлением* АТД и подробно описывает реализацию используемой структуры данных и операций. Например, описание класса для АТД *Integer* может выглядеть следующим образом:

```
class Integer {
attributes:
  int i

methods:
  setValue(int n)
  Integer addValue(Integer j)
}
```

Приведённый выше способ описания никак не связан с каким-либо конкретным языком программирования. В этом описании `class {...}` обозначает определение класса. Секции `attributes:` и `methods:` определяют реализацию структуры данных и операций соответствующего АТД.

Определение 3.2.1 (Класс) Класс *является реализацией абстрактного типа данных (АТД). Класс определяет* **атрибуты** *и* **методы**, *которые реализуют структуру данных и операции АТД соответственно.*

Экземпляры классов называются *объектами*. Следовательно, классы определяют свойства и поведение множества объектов.

3.3 Объект

Объекты уникально идентифицируются *именем*. Поэтому возможно существования различных объектов с одним и тем же множеством значений их атрибутов. Это похоже на то, как в традиционных языках программирования может существовать, например, две целых переменных i и j с одним и тем же значением, равным “5”. Совокупность заданных значений атрибутов объекта определяет *состояние* объекта в данный момент времени.

Определение 3.3.1 (Объект) Объект *является экземпляром класса. Объект уникально идентифицируется своим* **именем** *и определяет* **состояние**, *которое представляется значениями его атрибутов в определённый момент времени.*

Состояние объекта изменяется при выполнении соответсвующих этому объекту методов. Возможная последовательность изменения состояния объекта называется *поведением* объекта:

Определение 3.3.2 (Поведение) Поведение *объекта определяется множеством методов, которые могут быть применены к данному объекту.*

Таким образом, объектно-ориентированное программирование представляет собой средство реализации абстрактных типов данных. Объектно-ориентированная программа представляется в виде множества взаимодействующих объектов и изменение состояний этих объектов ведёт к решению задачи, которая возложена на данную программу. Возникает вопрос: каким образом объекты взаимодействуют друг с другом? В связи с этим, в следующем разделе вводится концепция *сообщения*.

3.4 Сообщение

Выполняющаяся объектно-ориентированная программа представляет собой совокупность объетов, которые создаются, уничтожаются и *взаимодействуют*. Это взаимодействие базируется на *сообщениях*, которые посылаются от одного объекта к другому. Эти сообщения представляют собой запрос к объекту-получателю на выполнение опеределённого метода. Например:

```
Integer i;
/* Определение нового объекта Integer */
i.setValue(1);
/* Установить его значение в 1 */
```

В этом примере объект i должен установить своё значение в 1. Фактически, объекту i посылается сообщение: "Применить метод *setValue* с аргументом 1 по отношению к самому этому объекту". Такой способ записи используется в языке C++. Дргие объектно-ориентированные языки программирования могут использовать другие способы записи, например "->".

Посылка сообщения объекту с запросом на выполнение какого-либо метода похожа на вызов процедуры в "традиционных" языках программирования. Однако, в объектно-ориентированном подходе объекты реагируют на получаемые сообщения выполнением определённых методов.

Определение 3.4.1 (Сообщение) Сообщение *является запросом к объекту навыполнение одного из его методов. Сообщение содержит*

- **имя** *метода и*
- **аргументы** *метода.*

Выполнение метода является реакцией получателя на соответствующее сообщение. Это возможно только в том случае, когда метод известен объекту.

Определение 3.4.2 (Метод) Метод *связан с классом. Объект выполняет методы в ответ на получаемые сообщения.*

3.5 Заключение

Фундаментальным принципом объектно-ориентированного программирования является представление программы как совокупности взаимодействующих объектов. Объекты реагируют в ответ на получаемые сообщения изменяя своё состояние в соответствии с выполняемыми методами, которые, в свою очередь, могут породить другие сообщения, посылаемые другим объектам. Это проиллюстрировано на рисунке 13.

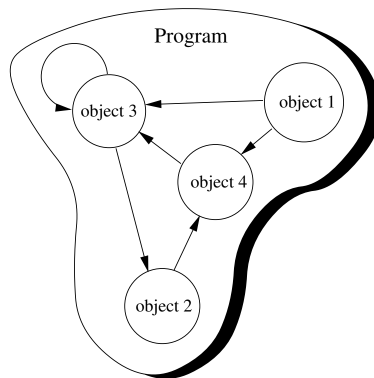

Рис. 13: Программа, состоящая из четырёх объектов.

Пример. Разница между процедурным и объектно-ориентированным подходом. Отображение символа на экране монитора при нажатии клавиши.

Процедурный подход.

1. ожидать нажатия клавиши;
2. получить значение клавиши;
3. отобразить соответствующий символ в текущей позиции курсора;
4. сдвинуть курсор.

Объектно-ориентированный подход. Есть два взаимодействующих объекта: *клавиша* и *экран*. Как только клавиша получает сообщение, что она должна изменить своё состояние на "нажата", соответствующий ей объект посылает сообщение экрану. Это сообщение запрашивает экран на отображение символа, соответствующего нажатой клавише.

3.6 C++: Основные расширения

В языке программирования C переменные должны определяться в начале блока. В языке программирования C++ допускается определять переменные и объекты в любом месте блока. То есть, переменные и объекты могут быть определены там, где они используются.

3.6.1 Типы данных

В языке программирования C++ вводится новый тип данных *ссылка (reference)*. Ссылку можно рассматривать в качестве "алиаса" (другого имени) для "реальных" переменных или объектов. Поскольку алиас не может существовать без соответствующего ему реального объекта, то не могут быть определены только ссылки. Для определения ссылки используется знак амперсанда (&). Например:

```
int ix;         /* ix является "реальной" переменной */
int& rx = ix;   /* rx является "алиасом" для переменной ix */

ix = 1;         /* rx == 1 также */
rx = 2;         /* ix == 2 также */
```

Ссылки могут использоваться в качестве аргументов и возвращаемых значений функций. Это позволяет передавать параметры по ссылки или возвращать "управляющий идентификатор" на полученную в функции переменную или объект.

В таблице 1 приводится список основных объявлений, используемых в языке программирования C++.

В языках программирования C и C++ допускается использование модификатора `const` для объявления переменной (или объекта) в качестве константы. В таблице 2 приводится список возможных комбинаций и описывается их значение. Далее приводится несколько примеров, которые демонстрируют использование модификатора `const`.

Предположим, что имеются следующие объявления:

```
int i;                          // простое целое
int* ip;                        // неинициализированный указатель на
                                // целое
int* const cp = &i;             // указатель-константа на целое
const int ci = 7;               // целая константа
const int* cip;                 // указатель на целую константу
const int* const cicp = &ci;    // указатель-константа на целую константу
```

Следующие операторы присваивания являются **допустимыми**:

```
i = ci;       // присвоить целочисленную константу целочисленной переменной
*cp = ci;     // присвоить целочисленную константу переменной,
              // на которую ссылается указатель-константа
cip = &ci;    // изменить указатель на целочисленную константу
cip = cicp;   // установить значение указателя на целочисленную константу
              // в значение указателя-константы на целочисленную константу
```

Объявление	*name* **есть ...**	**Пример**
type name;	переменная типа *type*	`int count;`
type name[];	массив элементов типа *type*	`int count[];`
type name[*n*];	массив из *n* элементов типа *type* (name[0], name[1], ..., name[*n*-1])	`int count[3];`
*type** name;	указатель на тип *type*	`int* count;`
*type** name[];	массив указателей на тип *type*	`int* count[];`
type (*name)[];	указатель на массив элементов типа *type*	`int (*count)[];`
type& name;	ссылка на объект типа *type*	`int& count;`
type name();	функция, возвращающая тип *type*	`int count();`
*type** name();	функция, возвращающая указатель на тип *type*	`int* count();`
type (*name)();	указатель на функцию, возвращающую тип *type*	`int (*count)();`
type& name();	функция, возвращающая ссылку на тип *type*	`int& count();`

Таблица 1: Объявления.

Объявление	*name* **есть ...**
const *type* name = *value*;	константа типа *type*
*type** const name = *value*;	указатель-константа на тип *type*
const *type** *name* = *value*;	указатель на константу типа *type*
const *type** const *name* = *value*;	указатель-константа на константу типа *type*

Таблица 2: Объявление констант.

Следующие операторы присваивания **не допускаются**:

```
ci = 8;       // нельзя изменять значение константы
*cip = 7;     // нельзя изменять целочисленную константу на которую ссылается
              // указатель
cp = &ci;     // нельзя изменять значение указателя-константы
ip = cip;     // это позволило бы изменить значение
              // целочисленной константы *cip с использованием *ip
```

3.6.2 Функции

Язык программирования C++ обеспечивает возможность перегрузки (overloading) функций так, как это определено в разделе 3.17. Например, можно определить две различных функции *max()*, одна из которых возвращает значение наибольшего целого среди двух целых, а вторая — наибольшую строку символов среди двух строк:

```
#include <stdio.h>

int max(int a, int b) {
  if (a > b) return a;
  return b;
```

```
}

char* max(char *a, char * b) {
  if (strcmp(a, b) > 0) return a;
  return b;
}

int main() {
  printf("max(19, 69) = %d\n", max(19, 69));
  printf("max(abc, def) = %s\n", max("abc", "def"));
  return 0;
}
```

В привеённом выше примере определяются две функции, которые отличаются списком параметров, и, следовательно, они определяют две различные функции. При первом вызове функции *printf()* в функции *main()* происходит вызов первой версии функции *max()*, поскольку на вход ей подаётся в качестве аргументов два целых значения. При втором вызове функции *printf()* происходит вызов второй версии функции *max()*.

Для передачи параметров можно использовать ссылки. Это обеспечивает возможность передавать параметры по адресу, что позволяет изменять значения аргументов функции внутри этой функции:

```
void f(int byValue, int& byReference) {
  byValue = 42;
  byReference = 42;
}

void bar() {
  int ix, jx;

  ix = jx = 1;
  f(ix, jx);
  /* ix == 1, jx == 42 */
}
```

3.7 C++: Первое объектно-ориентированное расширение

В этом разделе представлено, каким образом используются объектно-ориентированные концепции, представленные в разделе 3, в языке программирования C++.

3.7.1 Классы и объекты

Язык программирования C++ позволяет определять классы. Сущности классов называются *объектами*. Вернёмся к рассмотрению примера программы рисования из раздела 3.9.

Представление класса *Point* на языке программирования может выглядеть следующим образом:

```
class Point {
  int _x, _y;        // координаты точки

public:              // начало описания интерфейса
  void setX(const int val);
```

```
    void setY(const int val);
    int getX() { return _x; }
    int getY() { return _y; }
};

Point apoint;
```

В этом примере определяются класс *Point* и объект *apoint*. Определение класса можно рассматривать как определение структуры, в которую в качестве полей добавлены функции (или "методы"). В дополнение к этому можно определить различные *права доступа (access rights)*. Например, поля *_x* и *_y* являются **скрытыми (private)**, так как по умолчанию элементы класса являются скрытыми. Таким образом, необходимо явно "переключить" права доступа для определения **общедоступных** элементов. Это осуществляется с использованием ключевого слова `public`, после которого ставится двоеточие. К каждому элементу, объявленному с использованием ключевого слова `public` возможен доступ за пределами данного класса.

Можно переключится снова на описание скрытой части класса с использованием ключевого слова `private`. Например:

```
class F {
  // скрытые элементы по умолчанию ...

public:
  // секция общедоступных элементов ...

private:
  // секция скрытых элементов ...

public:
  // секция общедоступных элементов.
};
```

В языке программирования C++ существуют следующие виды доступа к элементам класса:

- *общедоступный (public)* — та часть интерфейса класса, в которой даются определения, "видимые" для всех объектов-пользователей данного класса;
- *защищённый (protected)* — та часть интерфейса класса, в которой даются определения, "видимые" только для объектов, относящихся к подклассам данного класса;
- *скрытый (private)* — та часть интерфейса класса, в которой даются определения, "скрытые" для объектов всех других классов.

В языке программирования C++ структура `structure` рассматривается как класс, все элементы которого являются общедоступными по умолчанию:

```
class Struct {
public:       // Элементы структуры являются общедоступными по умолчанию
  // элементы, методы
};
```

Таким образом, элементы класса (определённого с использованием ключевого слова `class` являются скрытыми по умолчанию, элементы же структуры (определённой с использованием ключевого слова `struct` являются общедоступными

по умолчанию. Однако, в структуре также можно использовать ключевое слово `private` для изменения прав доступа к элементам структуры.

Вернёмся к рассмотрению класса *Point*. Описание интерфейса этого класса начинается с описания общедоступной секции, в которой определяется четыре метода. Методы задания значения координат (методы *set()*) только объявлены и их функциональность должна быть где-то определена. Методы получения значений координат (методы *get()*) определены вместе с их реализацией (телом функции). Реализация этих методов определяется *внутри* класса. Такого рода методы называют *inline методами*.

Этот тип определения метода используется в тех случаях, когда реализация метода достаточно небольшая и простая. Это также позволяет уменьшить время выполнения всей программы, за счёт того, что тело таких методов "копируется" в те места кода, где осуществляется вызов соответствующих методов.

В противоположность этому, вызов методов *set()* является "реальным" вызовом функции. Реализация этих методов определяется вне определения класса, что делает необходимым каким-либо образом описывать к какому именно классу относится описание реализации данного метода. Например, какой-либо другой класс может также определять метод *setX()*, который отличается от соответствующего метода класса *Point*. Поэтому, необходимо явно задавать класс, к которому относится метод, реализация которого описывается. Это осуществляется с помощью операции "::":

```
void Point::setX(const int val) {
  _x = val;
}

void Point::setY(const int val) {
  _y = val;
}
```

В данном случае определяется метод *setX()* (*setY()*), соответствующий классу *Point*. Объект *apoint* может использовать эти методы для задания и получения своих координат:

```
Point apoint;

apoint.setX(1);     // Инициализация
apoint.setY(1);

...

int x = apoint.getX();
```

Возникает вопрос, каким образом методы "узнают" из каких объектов они вызываются. Это осуществляется посредством неявной передачи указателя на соответствующий объект в данный метод. Доступ к этому указателю внутри метода осуществляется с использованием специальной переменной с зарезервированным именем `this`. Это может быть проиллюстрировано следующим образом:

```
void Point::setX(const int val) {
  this->_x = val;   // Использование ссылки на соответствующий
                    // объект
}
```

```
void Point::setY(const int val) {
  this->_y = val;
}
```

В данном примере явно используется указатель this для явного обращения к элементам объекта. Компилятор автоматически "вставляет" такой указатель для элементов класса и поэтому в действительности может быть использовано первое определение методов *setX()* и *setY()*. Однако, иногда бывает необходимо и явное использование указателя this.

3.7.2 Конструкторы

Конструкторы представляют собой методы, которые вызываются с целью инициализации объекта в момент его объявления. Например, требуется, чтобы при создании объекта класса *Point* создавалась точка с координатами (0, 0):

```
class Point {
  int _x, _y;

public:
  Point() {
    _x = _y = 0;
  }

  void setX(const int val);
  void setY(const int val);
  int getX() { return _x; }
  int getY() { return _y; }
};
```

Имя конструктора совпадает с именем класса. У конструкторов остсутствуюти возвращаемые значения. Как и другие методы, конструкторы могут принимать аргументы. Например, для создания точки с координатами, отличными от (0, 0), в классе *Point* можно определить второй конструктор, который принимает два целых аргумента:

```
class Point {
  int _x, _y;

public:
  Point() {
    _x = _y = 0;
  }
  Point(const int x, const int y) {
    _x = x;
    _y = y;
  }

  void setX(const int val);
  void setY(const int val);
  int getX() { return _x; }
  int getY() { return _y; }
};
```

Конструкторы неявно вызываются при объявлении объектов соответствующих классов:

```
Point apoint;           // Point::Point()
Point bpoint(12, 34);   // Point::Point(const int, const int)
```

Для того чтобы создать новый объект на основе другого объекта необходимо скопировать свойства одного объекта во вновь создаваемый объект. Например, предположим что есть класс *List*, в котором динамически выделяется память для элементов списка. Для создания второго списка, который будет содержать те же элементы, что и первый, необходимо выделить память и скопировать каждый элемент из первого списка. Для обеспечения такой возможности при работе с классом *Point*, необходимо добавить третий конструктор:

```
class Point {
  int _x, _y;

public:
  Point() {
    _x = _y = 0;
  }
  Point(const int x, const int y) {
    _x = x;
    _y = y;
  }
  Point(const Point &from) {
    _x = from._x;
    _y = from._y;
  }

  void setX(const int val);
  void setY(const int val);
  int getX() { return _x; }
  int getY() { return _y; }
};
```

В качестве аргумента третий конструктор принимает ссылку на объект класса *Point* и присваивает элементам $_x$ и $_y$ соответствующие значения этого объекта.

Такого рода конструктор называется *конструктором копирования*. Конструктор копирования рекомендуется определять для любых создаваемых классов. Пример использования конструктора копирования:

```
Point apoint;            // Point::Point()
Point bpoint(apoint);    // Point::Point(const Point &)
Point cpoint = apoint;   // Point::Point(const Point &)
```

Использование конструкторов позволяет выполнить одно из требований, предъявляемых к абстрактным типам данных, а именно, требование инициализации в момент объявления соответствующего объекта.

3.7.3 Деструкторы

Рассмотрим класс *List*. Элементы списка динамически добавляются и удаляются. Для создания начального пустого списка используется конструктор. Однако, когда программа выходит из области видимости, соответствующей данному списку, необходимо освободить выделенную память. С этой целью определяется специальный метод — *деструктор*. Деструктор вызывается для соответствующего объекта в момент его удаления:

```
void foo() {
  List alist;      // List::List() создание
                   // пустого списка.
  ...              // добавление/удаление элементов
}                  // Вызов деструктора!
```

Удаление объекта происходит в момент, когда завершается выполнение фрагмента программы, соответствующего диапазону видимости данного объекта или в случае явного удаления этого объекта. Последнее соответствует случаю, когда объект создаётся динамически и, затем, когда он больше не нужен, удаляется.

Деструкторы объявляются подобно конструкторам, за исключением того, что имя деструктора соответствует имени класса, перед которым стоит знак тильда (˜):

```
class Point {
  int _x, _y;

public:
  Point() {
    _x = _y = 0;
  }
  Point(const int x, const int y) {
    _x = xval;
    _y = yval;
  }
  Point(const Point &from) {
    _x = from._x;
    _y = from._y;
  }

  ~Point() { /* Ничего не выполняется! */ }

  void setX(const int val);
  void setY(const int val);
  int getX() { return _x; }
  int getY() { return _y; }
};
```

Деструктор не принимает аргументов, поскольку деструкторы неявно вызываются в момент удаления объекта, и, поэтому, отсутствует возможность передачи аргументов.

3.8 C++: Перегрузка операторов

На языке программирования C++ описание абтсрактного типа данных *Complex* может выглядеть следующим образом:

```
class Complex {
  double _real,
         _imag;

  public:
    Complex() : _real(0.0), _imag(0.0) {}
    Complex(const double real, const double imag) :
```

```
      _real(real), _imag(imag) {}

    Complex add(const Complex op);
    Complex mul(const Complex op);
    ...
  };
```

Для вычисления суммы двух комплексных чисел можно использовать следующий фрагмент:

```
Complex a(1.0, 2.0), b(3.5, 1.2), c;

c = a.add(b);
```

В данном случае комплексной переменной c присваивается сумма комплексных переменных a и b. Для использования с этой целью оператора "+" в языке программирования C++ существует возможность *перегрузки (overload)* большинства из операторов языка для вновь создаваемых типов данных. Например, для класса *Complex* это будет выглядеть следующим образом:

```
class Complex {
  ...

public:
  ...

  Complex operator +(const Complex &op) {
    double real = _real + op._real,
           imag = _imag + op._imag;
    return(Complex(real, imag));
  }

  ...
};
```

В этом случае оператор + является элементом класса *Complex*. Выражение

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

транслируется в вызов метода

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

Таким образом для бинарного оператора + требуется только один аргумент. Первый операнд неявно предоставляется соответствующим объектом (в данном случае a).

Однако, вызов оператора может быть проинтерпретирован как вызов обычной функции:

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

В этом случае, перегруженный оператор **не** является элементом класса, а определяется как обычная перегружаемая функция. Например:

```
class Complex {
  ...

public:
  ...

  double real() { return _real; }
  double imag() { return _imag; }

  // No need to define operator here!
};

Complex operator +(Complex &op1, Complex &op2) {
  double real = op1.real() + op2.real(),
         imag = op1.imag() + op2.imag();
  return(Complex(real, imag));
}
```

В этом случае необходимо определить методы для доступа к действительной и мнимой частям класса, поскольку оператор определяется вне класса. Однако, данный оператор довольно тесно связан с классом, и, поэтому, имеет смысл предоставить ему доступ к скрытым элементам класса. Это возможно сделать посредством объявления его в качестве *друга* класса *Complex*.

3.9 Отношения

Отношение "Подобно" ("A-Kind-Of")

Предположим, что необходимо написать программу рисования. Эта программа должна позволять рисовать различные *объекты*, такие как точки, окружности, прямоугольники, треугольники и т.п. Для каждого объекта задаётся определение *класса*. Например класс "Точка" определяет точку посредством её координат:

```
class Point {
attributes:
  int x, y

methods:
  setX(int newX)
  getX()
  setY(int newY)
  getY()
}
```

Далее определим класс для описания окружностей. Окружность определяется точкой, которая является центром окружности и радиусом:

```
class Circle {
attributes:
  int x, y,
      radius

methods:
  setX(int newX)
```

```
    getX()
    setY(int newY)
    getY()
    setRadius(int newRadius)
    getRadius()
  }
```

При сравнении этих двух классов можно обратить внимание на следующее:

- Оба класса имеют два одинаковых элемента, а именно x и y. В классе *Point* эти элементы определяют позицию точки, а в классе *Circle* они определяют позицию центра окружности. То есть, элементы x и y означают одно и то же в обоих классах: они описывают позицию ассоциированного с ними объекта, определяя соответствующую точку.

- Оба класса определяют одно и то же множество методов для получения и присваиваивания значений элементам x и y.

- В классе *Circle* "добавлен" новый элемент данных *radius* и соответствующие методы доступа.

Зная свойства класса *Point* можно описать окружность как точку с добавлением понятия радиуса и методов для доступа к значению радиуса. Таким образом, окружность является "подобием" точки. Однако, окружность представляет собой более "специализированный" объект. Это проиллюстрировано графически на рисунке 14.

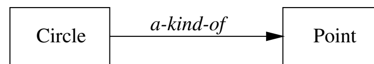

Рис. 14: Иллюстрация отношения "подобия" ("a-kind-of").

Отношение "Является подобным" (Is-A)

Предыдущее отношение используется на уровне класса для описания отношений между двумя похожими классами. Если создаются объекты двух этих классов, то их отношение определяется как "является подобным" ("объект является подобным другому объекту") (отношение "is-a").

Поскольку класс *Circle* подобен классу *Point*, то объект класса *Circle*, например *acircle*, является подобным объекту *point*[1]. Следовательно, поведение любой окружности похоже на поведение точки. Например, при изменении значения x точка сдвигается в направлении координаты x. Подобно этому при изменении значения атрибута x по отношению к окружности данная окружность также сдвигается в соответствующем направлении.

На рисунке 15 проиллюстрирован этот вид отношения. Здесь и далее объекты изображаются с помощью прямоугольников с закруглёнными углами и они именуются только маленькими буквами.

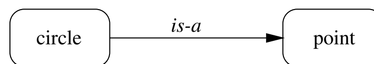

Рис. 15: Иллюстрация отношения "является подобным" ("is-a").

[1]Маленькие буквы используются для обозначения объектов.

Отношение “Является частью” (“Part-Of”)

Иногда возникается необходимость построения объектов посредством объединения одних объектов с другими. Например, в процедурном программировании тип данных “структура” или “запись” позволяют объединять вместе данные различных типов.

Предположим, что необходимо создать специальную фигуру, которая представляет собой некоторый логотип, и которая состоит из окружности и треугольника (предполагается, что уже определён класс *Triangle*, описывающий треугольник). Таким образом, логотип состоит из двух *частей* или, другими словами, окружность и треугольник *являются частями* логотипа:

```
class Logo {
attributes:
  Circle circle
  Triangle triangle

methods:
  set(Point where)
}
```

Это проиллюстрировано на рисунке 16.

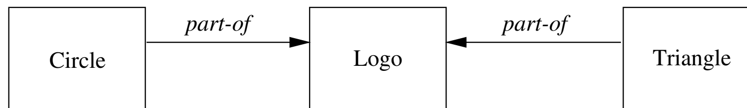

Рис. 16: Иллюстрация отношения “является частью” (“part-of”).

Отношение “Содержит” (“Has-A”)

Это отношение является обратным к отношению “является частью” и проиллюстрировано на рисунке 17.

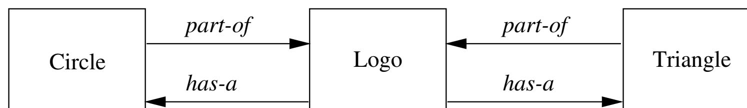

Рис. 17: Иллюстрация отношения “Содержит” (“has-a”).

3.10 Наследование

С помощью наследования реализуются отношения подобия (“kind-of” и “is-a”). Классы, которые подобны другим классам, содержат в себе свойства этих других классов. Таким образом, можно сказать, что окружность *наследует* свойства точки:

```
class Circle inherits from Point {
attributes:
  int radius

methods:
```

```
  setRadius(int newRadius)
  getRadius()
}
```

Класс *Circle* наследует все элементы данных и методы из класса *Point*. Нет необходимости определять их дважды, то есть используются уже существующие определения данных и методов.

На уровне объектов теперь возможно использовать окружность точно так же, как и точку, поскольку окружность является подобием точки. Например, можно определить объект "окружность" и установить координаты центра этой окружности следующим образом:

```
Circle acircle
acircle.setX(1)        /* Унаследовано от класса Point */
acircle.setY(2)
acircle.setRadius(3)   /* Добавлено в классе Circle */
```

Отношение "является подобным" ("is-a") означает, что использование объекта "окружность" возможно везде, где допускается использование объекта "точка". Например, можно реализовать функцию или метод, например *move()*, который должен перемещать точку по координате x:

```
move(Point apoint, int deltax) {
  apoint.setX(apoint.getX() + deltax)
}
```

Поскольку объект окружность является наследником объекта точка, то эта функция может быть использована с аргументом circle с целью перемещения центра окружности, и, следовательно, всей окружности:

```
Circle acircle
  ...
move(acircle, 10)   /* Move circle by moving */
                    /* its center point */
```

Определение 3.10.1 (Наследование) Наследование *представляет собой механизм, который позволяет классу A наследовать свойства класса B. В этом случае говорится, что "Класс A является наследником класса B" ("Класс A порождён из класса B"). Таким образом, объекты класса A имеют доступ к атрибутам и методам класса B, что исключает необходимость их переопределения.*

Следующее определение описывает два термина, которые используются для обозначения классов, когда используется наследование.

Определение 3.10.2 (Суперкласс/Подкласс) *Если класс A является наследником класса B, то класс B называется* **суперклассом** *по отношению к классу A. Класс A называется* **подклассом** *класса B.*

Объекты подкласса могут использоваться там же, где используются объекты соответствующего суперкласса. Это возможно, поскольку объекты подкласса обладают тем же поведением, что и объекты суперкласса.

Суперклассы также могут называться *родительскими классами*, а подклассы могут называться *дочерними классами* или просто *порождёнными классами*.

Возможно определить дальнейшее наследование из подкласса, что делает этот класс суперклассом по отношению к новому подклассу. Таким образом создаётся иерархия отношений суперкласс/подкласс. Изображение этой иерархии представляет собой *граф наследования* (Рисунок 18).

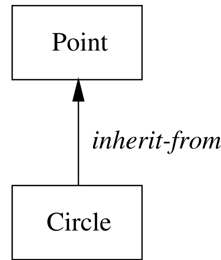

Рис. 18: Простейший граф наследования.

3.11 Множественное наследование

Одним из основных механизмов объектно-ориентированного подхода является множественное наследование. Множественное наследование означает, что у одного подкласса может быть *более чем один* суперкласс. Это позволяет подклассу наследовать свойства более чем одного суперкласса и объединять их свойства.

В качестве примера вновь рассмотрим программу рисования. Предположим, что уже существует класс *String*, который связан с обработкой текста. Например, в этом классе мог бы быть метод, служащий для *добавления (append)* текста. В программе можно было бы использовать этот класс для добавления текста к объектам рисования. Также неплохо было бы использовать уже существующие функции, например *move()* для перемещения текста. Следовательно, для текста должна быть определена точка, которая описывает его положение в области рисования. Для выполнения этих условий порождается новый класс *DrawableString*, который наследует свойства из классов *Point* и *String*, что проиллюстрировано на рисунке 19.

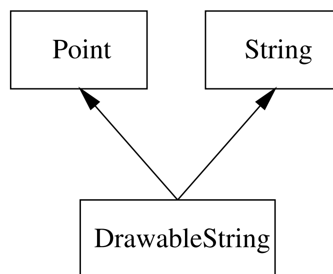

Рис. 19: Порождение класса, который наследует свойства классов *Point* и *String*.

На псевдо-коде этом может быть записано следующим образом (несколько суперклассов перечисляются через запятую):

```
class DrawableString inherits from Point, String {
attributes:
            /* Все атрибуты наследуются из суперклассов */

methods:
            /* Все методы наследуются из суперклассов */
}
```

Объекты класса *DrawableString* могут использоваться и как точки, и как строки. Поскольку объект *drawablestring* является подобным *точке*, то он может

быть сдвинут:

```
DrawableString dstring
...
move(dstring, 10)
...
```

Так как этот объект является подобным *строке*, то к нему можно добавить текст:

```
dstring.append("The red brown fox ...")
```

Определение 3.11.1 (Множественное наследование) *Если класс A является наследником более чем одного класса, то есть класс A является наследником классов B_1, B_2, ..., B_n, то это называется* **множественным наследованием**. *Множественное наследование может привести к* **конфликту имён** *в классе A, если по крайней мере в двух из его суперклассов определяются свойства с одним и тем же именем.*

Предположим, что в классе *String* определён метод *setX()*, который записывает в строку последовательность из "X" символов. Возникает вопрос, какой из методов будет унаследован классом *DrawableString* — метод из класса *Point*, или метод из класса *String*, или ни один из них?

Этот конфликт имён может быть разрешён по крайней мере двумя способами:

- Порядок, в котором расположены суперклассы, определяют свойство какого именно класса будет доступно при конфликте имён. Соответствующие свойства остальных классов будут "скрыты".
- Подкласс должен разрешать конфликт, за счёт какого-либо дополнительного описания такого рода свойств.

Первое решение не очень хорошее, поскольку тогда необходимо соблюдать определённый порядок при перечислении классов-родителей. Во втором случае подклассы должны явно переопределять свойства, которые связаны с конфликтом имён.

Особый тип конфликта имён возникает в случае, когда класс D создаётся посредством множественного наследования из суперклассов B и C, которые, в свою очередь, порождены из одного суперкласса A. Это приводит к графу наследования, изображённому на рисунке 20.
Возникает вопрос, какие свойства в действительности наследует класс D из его суперклассов B и C. Некоторые языки программирования решают эту проблему, порождая класс D со свойствами:

- свойства класса A плюс
- свойства классов B и C **за исключением** свойств, которые они унаследовали от класса A.

В этом случае в классе D отсутствует конфликт имён класса A. Однако, если классы B и C добавляют свои собственные свойства с одинаковыми именами, то в классе D опять возникает конфликт имён.

Другое возможное решение заключается в том, что класс D наследует все свойства классов-родителей. В этом случае, класс D владеет **двумя** копиями свойств класса A: одна из копий наследуется из класса B, а другая — из класса C.

В принципе, множественное наследование может быть смоделировано простым наследованием.

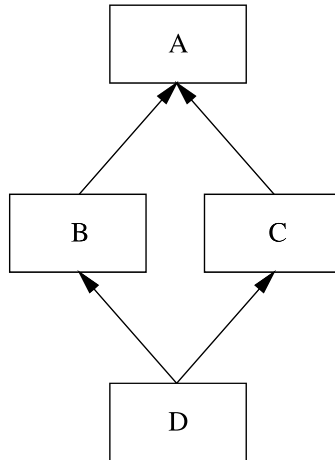

Рис. 20: Конфликт имён при множественном наследовании.

3.12 Абстрактные классы

Иногда необходимо только описать набор свойств, присущих некоторому множеству объектов, без определения реального поведения этих объектов. Например, в программе рисования каждый объект должен предоставлять метод для рисования самого себя в области отображения. Однако, алгоритм рисования каждого из объектов зависит от его формы. Например, функция рисования окружности отличается от функции рисования прямоугольника.

Пусть метод рисования объекта называется *print()*. Для того чтобы каждый рисуемый объект включал такой метод необходимо определить класс *DrawableObject*, из которого порождается любой другой класс, наследуя свойства объектов отображения:

```
abstract class DrawableObject {
attributes:

methods:
  print()
}
```

Здесь вводится новое ключевое слово **abstract**. Оно используется для того, чтобы отразить тот факт, что классы-наследники должны "переопределить" свойства, необходимые для выполнения требуемой функциональности. Таким образом, с точки зрения абстрактных классов, свойства только *специфицируются*, а не *определяются* полностью. Полное определение этих свойств, включая их семантику, должно быть обеспечено классами-наследниками.

Теперь каждый класс и примере программы рисования наследует свойства из основного класса, описывающего отображаемые объекты. Поэтому, класс *Point* измениться следующим образом:

```
class Point inherits from DrawableObject {
attributes:
  int x, y

methods:
  setX(int newX)
  getX()
```

```
    setY(int newY)
    getY()
    print()    /* Redefine for Point */
  }
```

Теперь каждый отображаемый объект будет содержать метод *print*, который должен обеспечивать отображение объекта в области отображения. Суперкласс для всех объектов отображения, а именно, класс *DrawableObject*, не предоставляет какой-либо функциональности для отображения себя самого. Также, нельзя создавать объекты этого класса. Этот класс только специфицирует свойства, которые должны быть определены в каждом из классов-наследников. Такой особый тип классов называется *абстрактным классом*:

Определение 3.12.1 (Абстрактный класс) *Класс A называется* **абстрактным классом**, *если он используется только как суперкласс по отношению к другим классам. Класс A только специфицирует свойства. Он не используется для создания объектов. Соответствующие классы-наследники должны определить свойства класса A.*

Таким образом, абстрактные классы позволяют специфицировать общие характеристики множества классов.

3.13 C++: Наследование

В языке C++ словосочетание “наследуется (порождён) из” заменяется двоеточием. В качестве примера рассмотрим класс, создаваемый для описания точек в трёхмерном пространстве. Для этого можно использовать уже существующий класс *Point*.

```
class Point3D : public Point {
  int _z;

public:
  Point3D() {
    setX(0);
    setY(0);
    _z = 0;
  }
  Point3D(const int x, const int y, const int z) {
    setX(x);
    setY(y);
    _z = z;
  }

  ~Point3D() { /* Nothing to do */ }

  int getZ() { return _z; }
  void setZ(const int val) { _z = val; }
};
```

3.13.1 Виды наследования

Наличие ключевого слова `public` в первой строке определения класса (в его *заголовке*) задаёт вид наследования. В языке программирования C++ используются следующие виды наследования: *public (общедоступный)*, *protected (за-*

щищённый) и *private (скрытый)*. По умолчанию используется скрытый вид наследования.

Вид наследования определяет права доступа к элементам суперклассов.

Рассмотрим класс *Derived*, порождённый из класса *Base*:

- **class Derived : private Base** : общедоступные (`public`) и защищённые (`protected`) элементы класса *Base* могут быть использованы только методами и *друзьями* класса *Derived*.
- **class Derived : protected Base** : общедоступные (`public`) и защищённые (`protected`) элементы класса *Base* могут быть использованы только методами и *друзьями* класса *Derived* и порождённых из него классов.
- **class Derived : public Base** : общедоступные (`public`) элементы класса *Base* могут быть использованы любым методом производных классов. Кроме того, защищённые (`protected`) элементы класса *Base* могут быть использованы только методами и *друзьями* класса *Derived* и порождённых из него классов.

3.13.2 Создание объектов

При создании объекта класса *Point3D* вызывается его конструктор. Поскольку класс *Point3D* порождён из класса *Pint*, то также вызывается конструктор класса *Point*. Причём, конструктор класса *Point* вызывается *перед* вызовом конструктора класса *Point3D*. В общем случае, перед вызовом определённого конструктора вызываются конструкторы каждого из суперклассов, которые инициализируют соответствующие части создаваемого объекта.

При объявлении объекта

```
Point3D point(1, 2, 3);
```

вызывается второй конструктор класса *Point3D*. Перед вызовом этого конструктора вызывается конструктор *Point()* с целью инициализации той части объекта, которая соответствует классу *Point*. При вызове данного конструктора координаты *_x* и *_y* инициализируются значением 0 (ноль). После этого вызывается конструктор *Point3D(const int, const int, const int)* класса *Point3D*, который явно переопределяет координаты *_x* и *_y* посредством вызова методов *setX()* и *setY()*, а также устанавливает значение координаты *_z*.

Недостатком использовании такого способа создания объекта класса *Point3D* является то, что координаты *_x* и *_y* инициализируются дважды, хотя в классе *Point* определён конструктор, который принимает два аргумента. Таким образом, необходимо обеспечить возможность вызова вместо *конструктора по умолчанию Point()* параметризованного конструктора *Point(const int, const int)*. Это может быть сделано следующим образом:

```
class Point3D : public Point {
  ...

public:
  Point3D() { ... }
  Point3D(
    const int x,
    const int y,
    const int z) : Point(x, y) {
      _z = z;
  }
```

```
    ...
};
```

Если используется несколько суперклассов, то вызовы их конструкторов перечисляются через запятую. Также этот механизм используются для создания объектов, которые содержатся внутри данного класса. Например, предположим что в классе *Part* определён единственный конструктор с одним аргументом. Для корректного создания объекта класса *Compound* необходимо вызвать конструктор *Part()* с соответствующим аргументом:

```
class Compound {
  Part part;
  ...

public:
  Compound(const int partParameter) : part(partParameter) {
    ...
  }
  ...
};
```

Такой способ инициализации также может быть использован по отношению к встроенным типам данных. Например, конструкторы класса *Point* могут быть записаны следующим образом:

```
Point() : _x(0), _y(0) {}
Point(const int x, const int y) : _x(x), _y(y) {}
```

3.13.3 Удаление объектов

При удалении объекта, например, при выходе из блока, вызывается деструктор соответствующего класса. Если класс является порождённым из других классов, то также вызываются и их деструкторы.

3.13.4 Множественное наследование

В языке C++ допускается множественное наследование. В этом случае, при создании подкласса его суперклассы перечисляются через запятую:

```
class DrawableString : public Point, public DrawableObject {
  ...

public:
  DrawableString(...) :
    Point(...),
    DrawableObject(...) {
      ...
  }
  ~DrawableString() { ... }
  ...
};
```

3.14 C++: Абстрактные классы

Абстрактные классы определяются также, как и обычные классы, за исключением того, что реализация некоторых из их методов должна быть определена в подклассах. В языке C++ такие методы определяются посредством добавления "= 0" после объявления заголовка соответствующего метода:

```
class DrawableObject {
  ...
public:
  ...
  virtual void print() = 0;
};
```

Такое определение класса требует, чтобы в каждом классе-наследнике была определена реализация метода *print()*. Такого рода методы называются *чистыми (pure) методами*.

Чистые методы должны быть объявлены с использование ключевого слова `virtual`, поскольку они будут использоваться только в порождённых классах. Классы, в которых определяются чистые методы называются *абстрактными классами*.

3.15 C++: Друзья

В языке программирования C++ возможно определелить функции и классы, которые должны быть друзьями класса, что обеспечивает им возможность прямого доступа к скрытым элементам данных. Например:

```
class Complex {
  ...

public:
  ...

  friend Complex operator +(
    const Complex &,
    const Complex &
  );
};

Complex operator +(const Complex &op1, const Complex &op2) {
  double real = op1._real + op2._real,
         imag = op1._imag + op2._imag;
  return(Complex(real, imag));
}
```

Если объявление дружественных методов и классов происходит в программе достаточно часто, то это значит, что необходимо изменить граф наследования.

3.16 Статическое и динамическое связывание

В языках программирования со строгой типизацией обычно необходимо *объявить* переменную до первого её использования в программе. Это также предполагает *определение* переменной, в результате чего компилятор выделяет место для переменной. Например, в языке програмиирования Pascal выражение

```
var i : integer;
```

объявляет переменную i типа *integer*. Кроме этого, выделяется необходимое количество памяти для хранения целого значения.

С помощью этого объявления осуществляется *связывание* имени i с типом *integer*. Это связывание действительно в рамках области действия, в которой переменная i объявлена. Это позволяет отслеживать корректность использования и целостность типов на этапе компиляции. Например, следующий фрагмент вызовет ошибку несоответствия типов:

```
var i : integer;
...
i := 'string';
```

Такой тип связывания называется "статическим", поскольку связывание происходит на этапе компиляции.

Определение 3.16.1 (Статическое связывание) *Если тип T явно ассоциирован с именем переменной N посредством объявления, то говорится, что переменная N* **статически связана** *с типом T. Процесс установления такой связи называется* **статическим связыванием**.

Однако, существуют языки программирования, в которых не используется явное определение типов переменных. Например, некоторые языки программирования позволяют вводить переменные по мере их необходимости:

```
...        /* Переменная i не пояляется */
i := 123   /* Создание переменной i целого типа */
```

Тип переменной i становится известным, как только ей присваивается определённое значение. Поскольку в приведённом примере *значение* переменной i представляет собой целое число, то *типом* переменной i является тип integer.

Определение 3.16.2 (Динамическое связывание) *Если тип T переменной с именем N неявно определяется текущим значением этой переменной, то говорится, что переменная N* **динамически связана** *с типом T. Процесс установления такой связи называется* **динамическим связыванием**.

Эти два вида связывания различаются моментами времени, в которые происходит связывание переменной с определённым типом данных. Рассмотрим следующий пример, выполнение которого возможно только при динамическом связывании:

```
if somecondition() == TRUE then
  n := 123
else
  n := 'abc'
endif
```

Тип переменной n после выполнения условного оператора `if` зависит от значения функции *somecondition()*. Если значения этой функции является TRUE, то переменная n принимает тип integer, в другом случае эта переменная принимает тип string.

3.17 Полиморфизм

Свойство полиморфизма позволяет сущности (например, переменной, функции или объекту) принимать различные представления. Определим возможные разновидности полиморфизма.

Первая разновидность полиморфизма подобна концепции динамического связывания. В этом случае тип переменной зависит от типа её значения в определённый момент времени:

```
v := 123        /* v имеет тип integer */
...             /* ипользование v в качестве integer */
v := 'abc'      /* v "переключается" на тип string */
...             /* использование v в качестве string */
```

Определение 3.17.1 (Полиморфизм (1)) *Концепция динамического связывания позволяет переменной принимать различные типы в зависимости от её значения в определённый момент времени. Эта особенность переменной называется* **полиморфизмом**.

Другая разновидность полиморфизма определяется по отношению к функциям. Например, предположим, что необходимо определить функцию *isNull()*, которая возвращает значение TRUE в том случае, когда её аргумент равен 0, и FALSE — в другом случае. Для целых чисел эта функция выглядит следующим образом:

```
boolean isNull(int i) {
  if (i == 0) then
    return TRUE
  else
    return FALSE
  endif
}
```

Однако, если такую же функцию необходимо использовать для проверки вещественных чисел, то необходимо использовать другой вид сравнения, что связано с ошибками округления:

```
boolean isNull(real r) {
  if (r < 0.01 and r > -0.01) then
    return TRUE
  else
    return FALSE
  endif
}
```

В обоих случаях хотелось бы, чтобы функция имела одно и то же имя *isNull*. В языках программирования, которые не обеспечивают свойство полиморфизма для функций, невозможно объявить эти две функции, поскольку, в этом случае, имя *isNull* будет объявлено дважды, что является ошибкой в таких языках (системах) программирования. Однако, язык программирования мог бы принимать во внимание *параметры* функции. Тогда функции (или методы) могли бы уникально идентифицироваться:

- *именем* функции (или метода) и
- *типами* её *параметров*.

Поскольку список параметров обеих функций *isNull* различается, то компилятор способен определить корректный вызов функции, используя актуальные типы соответствующих аргументов:

```
var i : integer
var r : real

i = 0
r = 0.0

...

if (isNull(i)) then ...   /* Используется isNull(int) */
...
if (isNull(r)) then ...   /* Используется isNull(real) */
```

Определение 3.17.2 (Полиморфизм (2)) *Если функция (или метод) определяется комбинацией*

- *её имени и*
- *списком типов её параметров*

то такая функция является **полиморфной**.

Эта разновидность полиморфизма позволяет использовать одно и то же имя для функций (или методов), у которых отличается список параметров. Иногда эта разновидность полиморфизма называется *перегрузкой (overloading)*.

Последняя разновидность полиморфизма позволяет объекту выбирать корректные методы.

Предположим, что существует функция *display()*, которая должна использоваться для изображения отображаемых объектов. Объявление этой функции могло бы выглядеть следующим образом:

```
display(DrawableObject o) {
  ...
  o.print()
  ...
}
```

Хотелось бы использовать эту функцию также для объектов классов, порождённых из класса *DrawableObject*:

```
Circle acircle
Point apoint
Rectangle arectangle

display(apoint)      /* Должен быть вызван метод apoint.print() */
display(acircle)     /* Должен быть вызван метод acircle.print() */
display(arectangle)  /* Должен быть вызван метод arectangle.print() */
```

Актуальный вызываемый метод должен определяться *значением (содержанием)* объекта *o* в функции *display()*. Рассмотрим следующий абстрактный пример:

```
class Base {
attributes:
```

```
methods:
  virtual f()
  bar()
}

class Derived inherits from Base {
attributes:

methods:
  virtual f()
  bar()
}

demo(Base o) {
  o.f()
  o.bar()
}

Base abase
Derived aderived

demo(abase)
demo(aderived)
```

В данном примере определяются классы *Base* и *Derived*. В каждом из этих классов определяются методы *f()* и *bar()*. Первый из методов определяется как `виртуальный (virtual)`. Это означает, что при вызове такого метода вызывается метод, соответствующий данному конкретному объекту.

Затем определяется функция *demo*, которая принимает в качестве аргумента объект класса *Base*. Следоваетльно, эта функция также может быть использована и для объектов класса *Derived*, поскольку соблюдается отношение подобия (is-a). Эта функция вызывается для объета класса *Base* и для объекта класса *Derived* соответственно.

Предположим, что методы *f()* и *bar* просто печатают своё имя и класс, в котором они определены. Результат будет выглядеть следующим образом:

```
Вызван метод f() класса Base.
Вызван метод bar() класса Base.
Вызван метод f() класса Derived.
Вызван метод bar() класса Base.
```

Первый вызов метода *demo()* использует объект класса *Base*. Таким образом, аргумент функции замещается объектом класса *Base*. Когда вызывается метод *f()*, то его действительная функциональность выбирается на основе текущего значения (содержимого) соответствующего объекта *o*. В данный момент времени это объекта класса *Base*. Следовательно, вызывается метод *f()*, определённый именно в классе *Base*.

Поскольку метод *bar()* не помечен, как `виртуальный (virtual)`, то вызывается метод *bar()*, опеределённый в классе *Base* и это можно не обсуждать.

При втором вызове функция *demo()* принимает в качестве аргумента объект класса *Derived*. Таким образом, аргумент *o* замещается объектом класса *Derived*. Однако, сам объект *o* представляет собой только ту часть объекта *aderived*, которая соответствует классу *Base*.

В данном случае, вызов метода *f()* соответствует вызову этого метода из класса *Derived*. С другой стороны, в качестве метода *bar()* вызывается метод *bar()*, определённый в классе *Base*.

Определение 3.17.3 (Полиморфизм (3)) *Объекты суперклассов могут быть замещены объектами соответствующих им подклассов. Операторы и методы подклассов могут быть определены так, что могут быть выбраны в следующих контекстах:*

1. *Контекст, основанный на типе, что приводит к выбору в рамках суперкласса.*
2. *Контекст, основанный на содержимом объекта, что приводит к выбору в рамках подкласса.*

Второй тип контекста называется **полиморфизмом**.

3.18 C++: Полиморфизм

Свойство полиморфизма реализуется в языке программирования C++ на основе механизма виртуальных методов. Например:

```
class DrawableObject {
public:
  virtual void print();
};
```

В классе *DrawableObject* определён виртуальный метод *print()*. Из этого класса могут быть порождены другие классы:

```
class Point : public DrawableObject {
  ...
public:
  ...
  void print() { ... }
};
```

Функция отображения объектов *display()* может быть определена следующим образом:

```
void display(const DrawableObject &obj) {
  ...
  obj.print();
}
```

При использовании виртуальных методов некоторые компиляторы выдают ошибку, если в соответствующих классах также не объявлены виртуальные деструкторы. Это необходимо при использовании указателей на (виртуальные) подклассы при их уничтожении.Поскольку, как правило, используется указатель на суперкласс, то, соответственно, вызывается деструктор суперкласса. Если же определён виртуальный деструктор, то вызывается деструктор соответствующего объекта. Например:

```
class Colour {
public:
  virtual ~Colour();
};
```

```
class Red : public Colour {
public:
  ~Red();      // Виртуально наследуется из класса Colour
};

class LightRed : public Red {
public:
  ~LightRed();
};
```

Используя эти классы можно определить следующий массив указателей *pallete*:

```
Colour *palette[3];
palette[0] = new Red;   // Динамическое создание объекта класса Red
palette[1] = new LightRed;
palette[2] = new Colour;
```

Операция `new` динамически создаёт новый объект заданного типа и возвращает указатель на него.

Для уничтожения объектов, созданных с помощью оператора `new` используется оператор `delete`, который явно уничтожает объект, с которым связан соответствующий указатель. При использовании оператора *delete* по отношению к элементам массива *pallete* будут вызваны следующие деструкторы:

```
delete palette[0];
// Вызов деструкторов ~Red() и ~Colour()
delete palette[1];
// Вызов деструкторов ~LightRed(), ~Red() и ~Colour()
delete palette[2];
// Вызов деструктора ~Colour()
```

Если бы деструкторы не были бы объявлены как виртуальные, то во всех случаях вызывался бы только деструктор *~Colour()*.

3.19 Родовые типы данных

При определении класса в действительности создаётся *тип данных, определённый пользователем.* Некоторые из таких типов могут оперировать и другими типами данных. Например, могли бы быть определены списки целых чисел, списки комплексных чисел, списки строк символов и даже списки списков.

Определение класса обеспечивает возможность создания *шаблона (template)* класса, который будет в действительности создан. В этом случае, определение реального класса создаётся в момент объявления соответствующего объекта. Например, описание шаблона класса может выглядеть следующим образом:

```
template class List for T {
  attributes:
    ...       /* Структура данных, необходимая для реализации */
              /* списка */

  methods:
    append(T element)
    T getFirst()
    T getNext()
    bool more()
}
```

Приведённый выше шаблон класса *List* похож на определение любого другого класса. Однако, в первой строке определения объявляется, что класс *List* представляет собой шаблон для различных типов данных. Идентификатор *T* используется в качестве "заменителя" реального типа данных. Например, метод *append()* принимает в качестве аргумента один элемент. Тип этого элемента будет тем же самым, что и тип данных, с которым создаётся реальный списковый объект. Например, если существует определение для типа данных Complex, то можно объявить список комплексных чисел:

```
List for Complex complexList
Complex aComplex,
        anotherComplex
complexList.append(anotherComplex)
complexList.append(aComplex)
```

В первой строке объявляется объект *complexList*, который должен представлять собой список комплексных чисел. В то же время компилятор использует определение шаблона, и заменяет каждое вхождение *T* на *Complex* и создаёт реальное определение соответствующего класса. Таким образом, определение класса выглядит примерно следующим образом:

```
class List {
  attributes:
    ...       /* Структура данных, необходимая для реализации */
              /* списка */

  methods:
    append(Complex element)
    Complex getFirst()
    Complex getNext()
    bool more()
}
```

Необходимо иметь в виду, что это не является точным представлением того, что генерирует компилятор. Компилятор должен обеспечить возможность создания множества списков для различных типов элементов в любой момент времени. Например, если необходимо создать другой список, например, типа String, то можно было бы это записать следующим образом:

```
List for Complex complexList
List for String stringList
...
```

В обоих случаях компилятор генерирует реальное определение класса. На самом деле компилятор генерирует уникальные имена для этих классов, в результате чего отсутствует конфликт по имени класса, причём этот процесс скрыт от программиста. В то же время, если объявляется ещё один список комплексных чисел, то компилятор опеределяет, что уже существует реальное определение соответствующего класса и использует именно его для создания нового списка. То есть при выполнении следующего фрагмента:

```
List for Complex aList
List for Complex anotherList
```

будет создано определение реального класса для объекта *aList* и оно будет далее использовано при объявлении объекта *anotheList*.

Определение 3.19.1 (Шаблон класса) *Если класс A параметризуется типом данных B, то класс A называется* **шаблоном класса**. *В момент создания объекта класса A, параметр B заменяется на* действительный тип данных. *Это позволяет определить* **реальный класс**, *основанный на шаблоне, который задан классом A, и действительном типе данных.*

Возможно также создание шаблонов классов с более чем одним параметром.