Базисные типы данных в языках программирования. Основные проблемы, связанные с базисными типами и способы их решения в различных языках. Понятие абстрактного типа данных и способы его реализации в современных языках программирования.

1. Простые типы данных, операции над ними

Все значения простых типов данных (называемых иногда *элементарными* или *примитивными*) являются атомарными, то есть не имеют внутренней структуры. Классификация простых типов данных приведена на Рис. 5-1. В этом пункте рассмотрим все простые типы, кроме подпрограммного, поскольку его имеет смысл обсуждать вместе с понятием подпрограммы (глава 6).

Арифметические типы данных

В компьютерах эти типы данных представляют числа, поэтому действительно являются основными.

Как мы уже отмечали, в некоторых языках (JavaScript) существует единственный числовой тип (Number), который представляет все допустимые числа. С точки зрения упрощения программ это, конечно, удобно, однако большинство индустриальных языков программирования разделяет арифметические типы на два вида: целые (для представления целочисленных значений) и вещественные (для представления чисел с дробной частью). Основная причина этого состоит в том, что представление целых и вещественных чисел в современных компьютерах различно, различается и набор операций. Например, операции сложения целых и вещественных чисел представляются разными машинными командами. Сама номенклатура машинных операций над целыми и вещественными числами различается (например, для вещественных чисел отсутствуют побитовые операции). Реализация операций над вещественными числами сложнее с аппаратной точки зрения, поэтому в ряде архитектур вещественные числа вообще могут отсутствовать. Для того, чтобы программист мог учитывать эти нюансы, многие

языки (в том числе, Паскаль, C++, Java, C#) разделяют числовые типы на два вида.

Целые типы данных

При рассмотрении целых типов возникают следующие основные вопросы:

- универсальность (насколько полно учтены машинные типы);
- наличие (или отсутствие) беззнаковых типов;
- представление (размер значения, диапазоны значений);
- надежность (какие ошибки могут возникать при выполнении операций с целыми значениями);
- набор операций.

Универсальность. Язык Паскаль предоставляет единственный целочисленный тип данных — integer. Этого вполне достаточно для учебного языка программирования, но неприемлемо в индустриальном программировании. Языки C++, Java и C# представляют универсальную номенклатуру целых типов, которая соответствует большинству современных архитектур. В эти языках определены однобайтовые целые числа (char в C++, byte в Java и C#), короткие целые (short), основные целые (int), длинные целые (long).

Беззнаковые типы. Практически все компьютерные архитектуры в дополнение к знаковым целым числам поддерживают и беззнаковые типы, то есть целочисленные типы, содержащие только неотрицательные значения. Это обусловлено необходимостью выполнения операций над адресами в машинных программах.

Адреса представляются беззнаковым целым типом (вспомним, что адрес — это номер ячейки памяти, начинающийся с нуля). Операции над адресами

называются *адресной арифметикой.*

Вторая причина использования беззнаковых чисел состоит в том, что при одинаковом размере максимальное значение беззнакового типа больше, чем максимум знакового (ведь не нужно хранить информацию о знаке). В случае, если диапазон целых значений невелик, использование беззнакового типа иногда необходимо.

Язык Java не содержит беззнаковых типов, что упрощает реализацию JVM и позволяет избежать ряда проблем, связанных с надежностью программ (см. ниже). Языки C++ и C# для каждого размера целого типа содержат знаковый и беззнаковый варианты. Прежде всего, это диктуется требованием универсальности.

Представление. Языки Паскаль и C++ не фиксируют представление целого типа. Размер и диапазон значений определяются реализацией. Это связано с тем, что эти языки были реализованы для большого количества машинных архитектур, существенно различавшихся по представлению чисел. Фиксация представления дала бы необоснованное преимущество конкретной архитектуре, поскольку реализации на других архитектурах были бы более сложными и менее эффективными. Язык C++ даже не фиксирует представление однобайтового типа char. В зависимости от реализации он может быть как знаковым (signed char), так и беззнаковым (unsigned char). Про размеры типов в C++ известно, что:

```
sizeof(char)=1
sizeof(char)<=sizeof(short)<=sizeof(int)<=sizeof(long)
```

Здесь `sizeof` — это статическая операция C++, которая применима как к именам типов, так и к объектам данных, и возвращает размер типа (объекта данных) в байтах.

С другой стороны, языки Java и C# полностью регламентируют размер и диапазон значений всех типов данных. Это связано с тем, что архитектуры, для

которой разрабатывались языки, вполне определены (для Java — это JVM, для C# - платформа .NET на архитектуре IA-32). Представление типов описывается в таблицах 1 и 2.

Размер	Знаковый	Диапазон	Беззнаковый	Диапазон
1	`sbyte`	-128..127	`byte`	0..255
2	`short`	-32768..32767	`ushort`	0..65535
4	`int`	$-2^{31}..2^{31}-1$	`uint`	$0..2^{32}-1$
8	`long`	$-2^{63}..2^{63}-1$	`ulong`	$0..2^{64}-1$

Таблица 1. Целые типы в языке C#

Размер	Имя	Диапазон
1	`byte`	-128..127
2	`short`	-32768..32767
4	`int`	$-2^{31}..2^{31}-1$
8	`long`	$-2^{63}..2^{63}-1$

Таблица 2. Целые типы в языке Java

Надежность. Надежность языковых конструкций определяется тем, насколько эти конструкции предохраняют программиста от случайных ошибок. Наличие беззнаковых типов снижает надежность работы с целыми числами.

Рассмотрим следующий фрагмент программы на C#:

```
int cnt = 0;
for (uint i = 256; i >= 0; i--) cnt++;
```

Этот цикл никогда не закончится, так как переменная цикла i всегда неотрицательна. Проблема в том, что в большинстве машинных архитектур целочисленные операции сложения и вычитания не генерируют ошибку при выходе результата за пределы допустимого диапазона. Решение о том, корректно ли значение операции, возлагается на программиста. Поэтому в

примере значение переменной i достигает нуля, а после этого вычитание 1 дает в результате (на архитектуре x86) сразу 2^{32}-1 (максимальное значение типа `uint`). Результат неверный, но ошибки не возникает и программа «зацикливается». Аналогичный результат получится и при переписывании фрагмента на язык C++ (`uint` нужно заменить на `unsigned`). Правда, язык C# дает возможность проконтролировать корректность результата целочисленных операций. Для этого служит конструкция `checked`:

```
checked
{
  for (uint i = 256; i >= 0; i--) cnt++;
}
```

В блоке, следующем за ключевым словом `checked`, все целочисленные операции проверяются на корректность, поэтому при возникновении переполнения (при попытке вычесть 1 из нулевого беззнакового значения) генерируется ошибка. Разумеется, время выполнения операций растет, поэтому по умолчанию операции не контролируются.

Заметим, что в языке Java сделать подобную ошибку просто невозможно.

Ещё один источник возможных ошибок при работе с целочисленными типами — преобразования значений одного типа в значения другого типа. Общий синтаксис преобразований типов в языках C++, Java, C#:

(X)`выражение`

Здесь X — это имя типа. Пусть `выражение` вычисляет значение типа Y. Тогда говорят, что приведенная выше конструкция осуществляет *преобразование из типа Y в X* (или *приведение типа Y к X*). Преобразования арифметических типов бывают двух видов: расширяющие и сужающие. Если множество значений типа Y есть подмножество значений типа X, то преобразование - *расширяющее*, в противном случае - *сужающее*.

Очевидно, что расширяющие преобразования — безопасны, так как преобразуемое значение сохраняется. Сужающие преобразования потенциально

опасны. Рассмотрим фрагмент программы на C#:

```
short i = -1;
ushort ui = (ushort)i;
int k = 256;
sbyte sb = (sbyte)k;
```

Здесь два сужающих преобразования: из `short` в `ushort` (знаковый и беззнаковый типы одного размера) и из `int` в `sbyte` (знаковые типы разных размеров). В обоих случаях результат может обескуражить: из -1 получилось 65535, а их 256 — 0. Если переписать фрагмент на языке C++, то получим такие же результаты на компьютерах архитектуры x86. На других архитектурах результат может быть другим, но в любом случае преобразуемое значение будет потеряно.

Конечно, не все сужающие преобразования ведут к ошибке: если преобразуемое значение типа Y одновременно входит и в тип X, то оно не изменится при приведении (например, нуль принадлежит всем числовым типам в наших языках, поэтому его безопасно приводить к любому числовому типу).

В языке C++ корректность сужающих преобразований не контролируется (программист берет на себя всю ответственность), в языке C# - только внутри `checked`-блока.

Ещё опасней ситуация при неявных преобразованиях. Преобразование типа — *неявное*, если оно вставляется или выполняется транслятором. В языках C# и Java неявными могут быть только расширяющие преобразования. А вот в C++ абсолютно все преобразования между любыми двумя арифметическими типами (не только целыми) могут быть неявными. Говоря другими словами, в C++ значение одного арифметического типа можно присваивать переменной любого другого арифметического типа. Компилятор просто вставит неявное преобразование. Никакого контроля корректности преобразования не проводится. Такая ситуация может приводить к ошибкам, которые трудно обнаружимы.

Набор операций.

Языки C++, Java, C# поддерживают практически единый набор операций над целыми значениями:

- арифметические (сложение «+», вычитание «-», умножение «*», деление нацело «/» - , остаток от деления «%»);
- побитовые (побитовое «или» «|», побитовое «и» «&», побитовая инверсия «~», побитовое «исключающее или» «^»);
- операции сдвига (влево «<<», вправо «>>»).

Все операции возвращают тот же тип, что и тип операндов (операнда).

Интересно, что отсутствие беззнаковых типов в языке Java имеет следствием появление дополнительной операции: логического сдвига вправо >>>. Дело в том, что машинные операции сдвига вправо имеют разный эффект для знаковых и беззнаковых типов (подробнее о сдвигах можно прочитать, например, в [10]). Для беззнаковых типов используется логический сдвиг (сдвигаемые биты замещаются нулями), а для знаковых типов — арифметический сдвиг (знак сохраняется, и сдвиг эквивалентен делению на степень двойки). При трансляции операции >> компилятор применяет операцию логического сдвига, если операция применяется к беззнаковому типу, и арифметического сдвига для знаковых типов. Но в языке Java есть только знаковые типы (поэтому операция >> всегда — арифметический сдвиг), а операция логического сдвига (игнорирующая знак) иногда необходима. Вот в этих случаях и применяется операция >>>.

К целым типам применимы операции сравнения (равенство «==», неравенство «!=», меньше «<», больше «>», меньше или равно «<=», больше или равно «>=»). Они выдают значения логического типа (см.ниже). Обратим внимание, что равенство задано символами «==», так как символ «=» занят операцией присваивания (эти операции могут смешиваться в выражениях, поэтому нужно различать их лексически).

Кроме перечисленных операций есть операции с побочным эффектом (меняющие операнд и вычисляющие значение):

- префиксный инкремент ++x (увеличение на единицу, возвращается новое значение x);
- постфиксный инкремент x++ (увеличение на единицу, возвращается старое значение x);
- префиксный декремент --x (уменьшение на единицу, возвращается новое значение x);
- постфиксный декремент x-- (уменьшение на единицу, возвращается старое значение x);

Присваивание также является операцией:

```
v = e
```

Она вычисляет значение выражения `e` и присваивает его правой части v. Если типы `v` и `e` различны, то транслятор либо вставляет неявное преобразование к типу правой части `v`, либо выдаст сообщение об ошибке (если язык запрещает такое преобразование). Присвоенное значение будет значением операции присваивания. Ассоциативность операции присваивания — справа налево, в отличие от операций, приведеных выше.

Для двуместных операций (таких, как сложение, сдвиги и тому подобных) определена комбинированная операция присваивания, например:

```
v += e
```

Как и обычное присваивание, эта операция вычисляет значение e (возможно с приведением типа), а затем увеличивает значение `v` на вычисленное значение. Новое значение `v` будет значением комбинированной операции.

Возникает вопрос: а есть ли в этих языках оператор присваивания? Непосредственно такого понятия нет (так как присваивание - это операция, а не оператор). Однако в этих языках есть понятие «оператор-выражение».

Оператор-выражение — это любое корректное выражение, после которого поставлена точка с запятой:

```
expr;
```

Поэтому в роли оператора присваивания выступает, например, конструкция:

```
i = x + y;
```

Конечно, присваивание может случиться и в других контекстах.

Вещественные типы данных

Вещественные типы данных служат для представления числовых дробных значений.

Большинство языков использует для хранения таких чисел формат с плавающей точкой и основанием 2. Значение числа представляется в виде:

$(-1)^S * M * 2^P$

где `S` — это бит знака (0 соответствует «+», 1 - «-»), `M` — нормализованная мантисса `(½ <= M < 1)`, `p` – порядок.

За счет нормализованности мантиссы число всегда представимо единственным образом (заметим, что нормализованность мантиссы означает, что старший бит мантиссы — всегда 1, поэтому эту единицу хранить не надо — она подразумевается).

Языки C++, C# и Java содержат два плавающих типа: `float` и `double`.

Как и для целых типов C++ не фиксирует конкретный формат представления чисел, можно сказать только, что:

```
sizeof(float) <= sizeof(double)
```

C# и Java поддерживают один и тот же формат представления чисел,

который определяется международным стандартом IEEE-754 [13]. Этот стандарт определяет базовые форматы представления плавающих чисел и правила выполнения арифметических операций над ними. Несмотря на добровольный характер этого стандарта практически все современные архитектуры (в том числе архитектура x86 и JVM) поддерживают его требования. Краткое описание стандарта можно найти в приложении к книге [10]. В языке Java (и спецификации JVM) форматы плавающих типов, и операции с плавающими типами полностью соответствуют стандарту.

IEEE-754 определяет два базовых формата чисел с плавающей точкой — с одинарной (32 бита) и двойной (64 бита) точностью. Формату одинарной точности соответствует тип `float`, а двойной точности — `double`.

В формате одинарной точности один бит занимает знак, 23 бита — мантисса, 8 битов — порядок. В формате двойной точности мантисса занимает 52 бита, порядок — 11 битов.

Два значения порядка (максимальное и минимальное) используются для представления специальных значений.

Легко вычислить минимальное (по абсолютной величине) и максимальное значение типа `float` (учитывая, что старший бит мантиссы есть всегда «1» и он не хранится):

$MAX\_FLOAT=(1-2^{-24})*2^{126}$; $MIN\_FLOAT= 1/2*2\ 2^{-126}$

Аналогичные значения можно вычислить для двойной точности.

Самое важное, что нужно помнить про плавающие типы, это то, что арифметические вычисления над ними неточны, и ошибка вычислений является относительной. Так для типа `float` точность сложения есть 2^{-23}, если операнды находятся между ½ и 1. Но для операндов порядка 2^{24} точность уже порядка 1.

Поэтому задачи с большим объемом вычислений с плавающей точкой требуют применения специальных алгоритмов, устойчивых к погрешностям

вычислений и входных данных.

Однако некоторые приложения требуют вычислений с фиксированной точностью, которая не зависит от абсолютной величины операндов (типичный пример — финансовые вычисления). Эти вычисления требуют представления дробных чисел с фиксированным числом знаков в дробной части.

Типы данных с таким представлением называются *фиксированными*.

Языки C++ и Java не содержат в базисе таких типов. Их приходится моделировать с помощью механизма классов.

Язык C# содержит специальный тип `decimal`, который служит для фиксированного представления чисел с дробной частью (заодно он может представлять целочисленные значения, не представимые целыми типами). Размер значений типа `decimal` — 128 битов. В них хранится бит знака `S`, коэффициент c $(0 < c < 2^{96})$ и масштаб e $(0 <= e <= 28)$. Значение определяется формулой:

$$(-1)^S * c * 10^{-e}$$

Таким образом, минимальное (по абсолютной величине) значение есть 10^{-29}, а максимальное — приблизительно $7.9 * 10^{28}$. Точность представления — 28-29 десятичных цифр.

Такой тип подходит и для финансовых расчетов и для ряда других приложений, требующих абсолютной погрешности в вычислениях.

Операции над вещественными числами включают в себя базовый арифметический набор (сложение, вычитание, умножение и деление), а также операции сравнения (надо помнить, что в силу погрешностей вычислений операция сравнения на равенство может давать неожиданные результаты).

Символьные типы данных

Символьные типы данных служат для представления

- символов (литер) алфавитов естественных языков,
- символов, управляющих работой устройств ввода/вывода (например, символ перехода на новую строку, табуляции и тому подобные),
- специальных символов (валютный знак).

Каждый символ имеет свое название (например, «возврат каретки», «маленькая кириллическая буква я», «знак копирайта»). Символьные типы основаны на понятии множества символов (character set).

Множество символов определяет, во-первых, набор символов, во-вторых, *кодировку* этого набора - отображение набора на диапазон целых чисел. Значения символьного типа — это и есть значения из этого диапазона.

Таким образом, символы могут представляться целым типом данных, что и было принято в ряде языков программирования. Например, в языке C целый тип `char` использовался для представления символов. «По наследству» такое представление перешло и в язык C++. Однако отсутствие специального типа для представления символов снижает надежность программ, обрабатывающих текстовую информацию (а удельный вес таких программ в индустрии программирования постоянно возрастает).

В самом деле, если применение арифметических операций неподобии вычитания кода одного символа из кода другого символа или сложения кода символа с константой вполне допустимо, то перемножение, деление кодов и тому подобные операции, вполне допустимые для целых, бессмысленны (а значит, ошибочны) для символьных типов. Поэтому современные языки программирования определяют отдельный символьный тип, несовместимый с целыми (хотя значения этого типа, конечно, представляют собой целые числа).

Вторая проблема, связанная с символьными типами, - это проблема представления произвольных множеств символов. Ранние языки программирования (как и виртуальные архитектуры, на которых они основывались) использовали однобайтовые множества символов. В этих

множествах число различных символов не превосходило двух с небольшим сотен (а при использовании только английского алфавита хватало и 127 различных символов), поэтому для представления символа хватало одного байта (вспомним, что размер типа `char` в языке C — один байт). Однобайтовые множества символов породили ряд проблем.

Первая состояла в невозможности представления объёмных иероглифических алфавитов (китайский, японский и другие).

Вторая состояла в том, что однобайтовое множество по отдельности покрывает небольшое множество алфавитов. Исторически сложилось так, что почти каждое множество символов содержит общий для всех набор, включающий управляющие символы, символы английского алфавита, цифры, знаки препинания и тому подобные. Коды символов из этого набора соответствуют американскому стандарту ASCII-7 и принимают значения в диапазоне 0-127. Остальные 128 символов в однобайтовой кодировке заполняются символами других алфавитов. Очевидно, что даже европейские алфавиты предоставляют существенно больший набор символов. Поэтому пришлось разрабатывать множества символов отдельно для западноевропейских языков, кириллицы, турецкого, арабского, греческого и других языков. Можно представить, какие проблемы возникали при создании программ, одновременно обрабатывающих тексты на разных языках (например, русском, французском и турецком).

Ещё одной проблемой стало обилие кодировок даже для одного алфавита (для русского набора символов в настоящее время широко используются по меньшей мере четыре однобайтовые кодировки).

Ситуация изменилась после появления в 1991 году универсального множества символов UNICODE. UNICODE содержит символы почти всех реально используемых алфавитов. Их достаточно много (тысячи), поэтому каждый символ в этом множестве кодируется двухбайтовым числом без знака (промежуток 0 — 65535). Это число называется кодовой точкой (code-point).

Для каждого алфавита выделен свой диапазон кодовых точек, например, кириллица находится в промежутке от 1040 (большая буква А) до 1298(буква Ҋ — большая буква Л с крючком). Современная версия UNICODE определяет кодовые точки для символов со значениями и больше 65535.

Поэтому почти все языки, появившиеся после 1991 года, используют UNICODE.

Тип `char` в языках Java и C# тоже использует это множество символов. Множество операций над типом включает в себя операции сравнения (в соответствии со стандартом UNICODE), присваивания и операцию «+» сцепления со строкой. Например, в следующем фрагменте строка `s` получит значение `”line 1”`:

```
char c = '1';
string s = ”line ” + c;
```

Значения типа `char` могут неявно приводиться к целым типам данных, если это безопасно (например, к `int` могут, а к `byte` не могут), поэтому в следующем фрагменте операция «+» обозначает операцию сложения целых чисел:

```
char c = 'x';
int k = x + 1; // то же самое, что и  (int)x + 1
```

Обратная операция преобразования (приведение целых значений к символьному типу) допустима только явно:

```
sbyte b = -1;
char c = (char)b;
```

При этом корректность преобразования в C# контролируется при включении операции в checked-блок (см. выше).

В языке C++, как уже говорилось, символьные типы рассматриваются как целочисленные. Кроме типа `char`, в языке C++ появился тип `wchar_t` специально для представления символов из UNICODE. Этот тип является

целым беззнаковым двухбайтовым типом.

Логические типы данных

Логический тип данных обозначается в языках C++ и C# Java ключевым словом `bool (boolean` в Java). Он состоит их двух значений: `true` (истина) и `false` (ложь).

Набор операций состоит из логического «и» - «`&&`», логического «или» - «`||`» и отрицания «не» - «`!`» с обычным математическим смыслом.

Интересной особенностью вычисления логических операций `&& и ||` является их «ленивость». *Ленивость* исполнения логических операций состоит в том, что они вычисляются слева направо, и если значение левого операнда определяет значение всего выражения (для операции && это `false`, а для операции `||` -`true`), то правые операнды вычисляться не должны.

В языках C# и Java логический тип не может приводиться к целым (да и другим) типам данных (ни явно, ни, тем более, неявно).

В языке C++ ситуация более сложная. Дело в том, что логический тип был включен в язык не сразу, а вначале C++ унаследовал правила языка C. В этом языке логические операции были (перечисленные выше), а отдельного логического типа не было. Логические операции возвращали целое значение 1 в качестве «истины» и 0 – в качестве «лжи». Если в каком-то месте от выражения произвольного типа требовалось логическое значение, то это выражение (при необходимости) приводилось к целому типу, вычислялось, и ненулевое значение соответствовало «истине», а нулевое - «лжи». Поэтому в C (и C++) вполне можно писать такие фрагменты:

```
while (n-m)
{
  if (n > m)
    n -= m;
  else
```

```
        m -= n;
    }
```

В языках C# и Java в этом месте компилятор выдаст ошибку, а правильный фрагмент программы выглядит так:

```
while (n-m != 0) …
```

Также, конечно, можно писать и на C/C++.

Таким образом, тип `bool` в C++ неявно приводится к целому (`true` - в 1, `false` — в 0) и наоборот, что, конечно, снижает его ценность (по сравнению с логическим типом в C# и Java).

Порядковые типы данных

Порядковые типы данных делятся на перечислимые типы и типы диапазона.

Перечислимый тип задается прямым перечислением констант-значений. Каждое значение задается своим именем.

Впервые перечислимые типы появились в языке Паскаль и сразу стали популярными как среди разработчиков языков программирования, так и среди программистов. Дело в том, что использование перечислимых типов делает программу более понятной, поскольку все значения имеют имена, которые характеризуют смысл значения.

С точки зрения реализации каждое значение перечислимого типа отображается в целое число, соответствующее его порядковому номеру в определении типа. Этот номер называется *ординалом* значения. По умолчанию нумерация ординалов начинается с нуля, однако некоторые языки позволяют придать свои значения ординалам.

Перечислимые типы реализованы в языках C++, C# и Java (начиная с 2005 года). Однако в реализации этих типов есть различия.

Начнем с языка C++. Представим себе задачу моделирования светофора. В ней важную роль имеет цвет светофора. Его удобно представить перечислимым типом:

```
enum TrafficColors {
        RED, YELLOW, GREEN
};
```

Пусть у нас есть класс TrafficLights, представляющий светофор, и переменная `light1` этого класса. Тогда следующий вызов не нуждается в дополнительных комментариях:

```
light1.SetSignal(RED);
```

Очевидно, это переключение светофора в красный цвет.

Цвета нумеруются у нас с 0 по 2 (`RED=0, YELLOW=1, GREEN=2`). Иногда конкретный номер цвета не важен, но мы можем расширить задачу, потребовав, чтобы числовые значения констант перечислимого типа соответствовали значениям цветов в цветовой системе RGB-24 (в этой системе каждому цвету соответствует целое значение, составленное из трех байтов — значение каждого байта есть интенсивность красного, зеленого и синего цветов в диапазоне от 0 до 255).

Тогда объявление нашего типа изменится (используется шестнадцатеричная форма целых констант — каждая пара цифр соответствует байту со значением 255 или 0):

```
enum TrafficColors {

        RED = 0xFF0000,
        YELLOW = 0xFFFF00,
        GREEN = 0x00FF00

};
```

Теперь значения цветов можно использовать в функциях рисования, требующих указания значения цвета, например:

```
brush.SetBrushColor(YELLOW);
```

Здесь мы предполагаем, что функция `SetBrushColor` имеет один параметр целого типа, означающий цвет в системе RGB-24.

Таким образом, значения перечислимого типа неявно преобразуются в значения типа `int`. Обратное, однако, неверно. Попытка присвоить переменной перечислимого типа какое-либо целое значение рассматривается компилятором, как ошибка:

```
TrafficColors cl = 127; // ошибка!!!
```

Если мы хотим придать переменной новое числовое значение, то должны написать явное преобразование:

```
TrafficColors cl = (TrafficColors)-1;
            // теперь формально правильно
```

Однако ясно, что такое преобразование не всегда приводит к ожидаемым результатам (какой реальный цвет будет у переменной `cl`?), почему и требуется выписывать явное преобразование.

Однако у реализации перечислимых типов в C++ есть ряд недостатков. Первый недостаток состоит в том, что константы перечислимого типа имеют ту же область действия, что и имя перечислимого типа (так называемый «*неявный импорт имен*»). Это может приводить к конфликту имен. Например, если мы одновременно пытаемся моделировать семафор, то появляется еще одно перечисление:

```
enum SemaColors {
        RED, GREEN
};
```

Это приводит к конфликту. Для учебных программ это не страшно (имена в своих программах поменять легко). Но в условиях индустриального программирования, когда широко используются библиотеки от сторонних производителей (например, одна библиотека моделирует светофор, другая - семафор), такие конфликты доставляют немало хлопот программистам.

Вторая проблема: фиксированное представление перечислимых типов в C++— они всегда реализуются как целые (на базе основного целого типа `int`). Иногда это неудобно и приводит к трате памяти (ведь хватило бы и более короткого типа).

В языке C# эти недостатки преодолены. Заметим, что приведенные выше объявления типов `TrafficColors` и `SemaColors` являются вполне корректными в языке C# (надо только убрать концевые точки с запятой). Однако имена перечислимых констант локализованы в типе и могут использоваться за пределами объявления типа только в виде:

```
имя_типа.имя_константы
```

Например:

```
light1.SetSignal(TrafficColors.RED);
```

Кроме того, значения перечислимого типа могут храниться как значения любого целочисленного типа (такой тип называется *базовым* для перечисления). По умолчанию базовым является основной целый `int`, но базовый тип легко поменять:

```
enum SemaColors : byte
{
        Red, Green
}
```

В отличие от C++ значения перечислимых типов должны преобразовываться в базовый тип только явно:

```
brush.SetBrushColor(TrafficColors.YELLOW);
// в C# - ошибка — SetBrushColor требует целый тип
brush.SetBrushColor((int)TrafficColors.YELLOW);
// теперь правильно!
```

В языке Java перечислимые типы появились относительно недавно (с 2005

года) и реализованы наиболее «экзотично»: на самом деле перечислимые типы в Java — это классы. Не вдаваясь в тонкости (о них можно прочитать, например, в [14]), заметим, что объявление и использование перечислений внешне похоже на C#:

```
enum TrafficColors
{
        Red, Yellow, Blue
}
TrafficColors cl = TrafficColors.Yellow;
```

Неявные преобразования в целый тип и обратно также запрещены, но можно использовать методы перечислимого класса `ordinal()`, возвращающий порядковый номер константы, и `values()`, возвращающий массив констант значений перечислимого типа (индекс в массиве — номер константы в типе):

```
int colorNumber = TrafficColors Red.ordinal();
        //  colorNumber получает значение 0
TrafficColors c =
TrafficColors.values[colorNumber+1];
// c  получает значение TrafficColors. Yellow
```

Напоследок заметим, что во всех рассмотренных языках к перечислимым типам применимы операции сравнения (сравниваются ординалы значений).

Скажем несколько слов о диапазонных типах. Они появились в языке Паскаль и позволяли ограничить значения какого-либо целого или перечислимого типа.

Например:

```
type Index = 0..N; // диапазон целого типа
type DaysOfWeek = (Mon, Tue, Wed, Thu, Fri, Sat,
Sun);
// перечислимый тип — дни недели
type WorkDays = Mon..Fri; // диапазон типа DaysOfWeek
    WeekendDays = Sat..Sun;// другой диапазон
```

```
DaysOfWeek
```

Чаще всего диапазонные типы используются в качестве типа индекса в массивах. Иногда это удобно, так как позволяет использовать произвольные границы индексов (например, с 1, или с отрицательного числа). В современных языках программирования индексы в массиве начинаются только с 0 и могут быть только целыми, поэтому область применения типов диапазонов существенно сужается.

В языках C++, Java и C# диапазонных типов нет.

Указательные типы данных

Указатели являются абстракцией понятия машинного адреса. Средства работы напрямую с адресами, с одной стороны, удобны, так как дают возможности, сравнимые с возможностями программирования на машинном языке (то есть использовать все возможности машинной архитектуры без ограничений), с другой — опасны (если нет ограничений, то можно сделать все, что угодно, в том числе и ошибки).

Опасности применения указателей перевешивают их удобства, поэтому языки C# и Java не используют понятие указателя (точнее, в C# понятие указателя есть, но оно используется крайне ограниченно и только для работы с библиотеками, написанными на C или других императивных языках, в нашем курсе эти средства не рассматриваются). Вместо указателей в этих языках используется более абстрактный и безопасный тип данных - ссылки (см. следующий пункт).

Указатели есть в языке C++ (где они без изменений заимствованы из языка C). Синтаксис объявления указательной переменной в C/C++:

```
имя_типа_данных * имя_указателя
```

Например:

```
int * pi; X * pX; char * message = ”Hello!”;
```

Зачем вообще нужны указатели (другими словами, работа с адресами) в языках программирования? Можно назвать следующие основные причины использования указателей:

- передача адресов объектов данных в подпрограммы;
- работа с объектами из динамической памяти;
- использование адресной арифметики;
- использование адресов подпрограмм как объектов данных (передача параметров-подпрограмм в другие подпрограммы и тому подобное).

Рассмотрим подробнее каждую причину.

Передача адресов данных в подпрограммы. Если подпрограмма меняет содержимое своего параметра - объекта данных, то единственный способ это сделать — передать адрес объекта. Кроме того, если параметр — большой объект, то копировать его в подпрограмму накладно, поэтому и передают только адрес объекта.

Для получения адреса объекта служит адресная операция «&» (не путайте её с побитовой операцией «и» - последняя имеет два аргумента):

```
&переменная
```

Если переменная имеет тип T, то адресная операция возвращает тип указателя на T (то есть, T *).

Например, функция форматного ввода `scanf` из стандартной библиотеки языка C вводит (то есть, изменяет) значение целой переменной `k`, поэтому необходимо передать туда адрес `k`:

```
scanf (”%d”, &k);
```

Первый параметр (формат) `”%d”` сообщает функции, что она должна ввести целое число, представленное в десятичной системе счисления. Если бы параметром был `“%x”`, то число должно было бы быть представлено в

шестнадцатиричной системе.

Как получить доступ к объекту, адрес которого находится в указателе? Для этого служит операция «*»:

```
int j = -1;
int * pi = &j; // в pi — адрес  j
*pi = 0;//*pi — это j, поэтому j получает значение 0
```

Очевидно, что для любой переменной `var` верно: `var` эквивалентно `*(&var)`.

В языке C++ можно не передавать указатели в подпрограммы, а воспользоваться ссылками (как и рекомендуется делать), однако в C такой возможности нет.

Работа с объектами из динамической памяти. Это самая важная (и нередко единственная) причина использования указателей.

В C++ объект типа T размещается в динамической памяти с помощью операции `new имя_типа`:

```
T * p = new T;
```

Отведенная динамическая память обязана быть освобождена с помощью операции `delete указатель`:

```
delete p;
```

`Указатель` должен содержать адрес, полученный от операции `new`. Иначе говоря, это адрес объекта, размещенного в динамической памяти. Объект после выполнения `delete` перестаёт существовать, и все ссылки на него становятся ошибочны.

Если в памяти необходимо разместить массив (непрерывную последовательность) объектов типа T, то нужно использовать другие формы операций `new` и `delete`:

```
T * pArr = new T[256]; ... delete [] pArr;
```

Операция `new[]` возвращает адрес первого элемента размещенного массива. Обратите внимание, что вид объявления указателя не зависит от того, указывает он на единственный объект, или на последовательность объектов (массив). Такая неразличимость указателя на объект и указателя на массив (точнее, на первый элемент массива) — характерная особенность языков C/C++.

Ещё одна особенность языков C/C++ - неразличимость адресов, полученных от адресной операции и от операции `new`.

Размещение объектов данных в динамической памяти — ценная возможность современных языков. Некоторые языки, как уже отмечалось, размещают объекты классов и массивы только в динамической памяти, правда, используя для этого аппарат ссылок, более надежный и абстрактный, чем указатель. Поэтому операция `new` присутствует во всех рассматриваемых здесь языках.

Однако наличие (и требование!) операции явного освобождения памяти (`delete`) становится причиной частых и труднообнаруживаемых ошибок в программах. Две ошибки работы с динамическими объектами, которые могут возникнуть при выполнении (или невыполнении) операции `delete` — это «висячие ссылки» и «мусор».

Висячие ссылки — это адреса уже уничтоженных объектов. Попытка обратиться к объектам по этим адресам ошибочна (по этому адресу, например, могут располагаться совершенно другие объекты, размещенные уже после «смерти» уничтоженного объекта), однако проконтролировать и обнаружить эту ошибку очень трудно.

Вот пример возникновения висячей ссылки:

```
int * pX = new int; // объект размещён в памяти
int * pY = pX;
// теперь pX и pY ссылаются на один и тот же объект
delete pX; pX = 0;// объект уничтожен
```

```
*pY = -1; // ошибка!!! Объект * pY не существует
```

Здесь указатель `pY` становится висячей ссылкой после выполнения `delete pX`. Предсказать, как, когда и где проявится эта ошибка практически невозможно. Если бы в примере обнулили не только `pX`, но и `pY`, то висячая ссылка бы исчезла, и ошибка немедленно обнаружилась бы при попытке обратиться к нулевому адресу памяти (виртуальные компьютеры современных ОС генерируют в этом случае ошибку защиты памяти).

Конечно, можно легко избежать висячих ссылок, если вообще не уничтожать объекты (не использовать `delete`). Но это лекарство ещё хуже болезни, поскольку приводит к другой ошибке при работе с динамическими объектами - возникновению мусора.

Мусор — это неуничтоженные объекты из динамической памяти, на которые отсутствуют ссылки. Раз нет ссылок на них, то объекты не могут использоваться (бесполезны), но и не могут быть уничтожены (зря занимают память) — типичный мусор! Если мусор заполняет всю доступную память, то размещение новых объектов становится невозможным, и программа аварийно завершается. Проблема мусора в том, что эта ошибка проявляется при длительной работе с программой, иногда уже на стадии эксплуатации. Обнаружить её ещё труднее, чем висячую ссылку.

Пример возникновения мусора:

```
int * pX = new int; // разместили первый объект
int * pY = new int; // разместили второй объект
pY = pX; // ошибка!!! Первый объект стал мусорным
```

Как избавиться от мусора и висячих ссылок? Во-первых, не использовать операцию явного освобождения памяти (избавимся от висячих ссылок), во-вторых, добавим в виртуальную машину языка (точнее, в подсистему времени выполнения) компоненту — автоматический сборщик мусора. Диспетчер динамической памяти должен отслеживать (запоминать) все ссылки на объекты из динамической памяти. Если все ссылки отслеживаются, то автоматический

сборщик мусора может найти неиспользуемые объекты и пометить соответствующие участки памяти как свободные. Как правило, сборщик мусора дополнительно собирает все используемые объекты в непрерывную область памяти (ссылки на них, конечно, изменяются) — этот процесс называется *дефрагментацией памяти*. Когда вызывать автоматический сборщик мусора? Во-первых, при выполнении операции `new`, когда свободной памяти нужного размера нет. Во-вторых, современные системы могут выполнять автоматическую сборку мусора параллельно с выполнением программы.

Конечно, автоматическая сборка мусора влечет повышенные накладные расходы — эффективность программ снижается. Однако повышение надежности программы компенсирует этот недостаток, поэтому современные языки, такие как Java и C#, используют автоматическую сборку мусора.

В C++ реализовать эту возможность затруднительно в силу того, что указатели на динамические объекты (адрес получен от `new`) и нединамические (адрес получен путем адресной операции &) не отличаются друг от друга, поэтому отследить все ссылки в программе практически невозможно. Эта особенность языка приводит, кстати, ещё к одной причине сбоев в программах — когда ошибочно вызывается операция `delete` для адреса нединамического объекта.

Интересно, что реализация языка C++ для платформы .NET использует автоматическую сборку мусора (так как любой язык в .NET должен её использовать), однако она реализована не для указателей C++, а для переменных специального нестандартного типа, который эквивалентен ссылкам языка C# [15].

Использование адресной арифметики. Адресная арифметика — это применение операций сложения и вычитания к указателям.

Пусть, например, указатель `p` содержит адрес объекта целого типа `(adr)`. Тогда операция `p + 1` возвращает адрес, равный `adr+sizeof(int)`.

Аналогично, операция `p - 1` возвращает адрес, равный `adr - sizeof(int)`. В общем случае, пусть `p` объявлен как указатель на тип `T (T * p;),ie` – выражение целого типа, `Addr` – значение (машинный адрес), хранящееся в `p`. Тогда `p + ie` – это значение (машинный адрес) типа `T*`, которое равно `Addr + ie * sizeof(T)`. Аналогично, `p - ie` – это значение (машинный адрес) типа `T*`, которое равно `Addr - ie * sizeof(T)`.

Нетрудно видеть, что разность между двумя указателями `p1` и `p2` на тип T есть целое значение, равное разности значений машинных адресов из `p1` и `p2`, деленной на `sizeof(T)`.

Использование адресной арифметики позволяет очень эффективно работать с последовательностями объектов данных одного типа (то есть с массивами).

Однако непосредственная работа с адресами не может быть проконтролирована ни статически (в момент трансляции), ни динамически (во время выполнения программы). В самом деле — при вычислении адреса никак нельзя гарантировать, что полученный адрес корректен, то есть указывает на правильный объект нужного типа. Поэтому программы, использующие адресную арифметику, могут содержать ошибки, которые трудно обнаружить.

Использование адресов подпрограмм. С точки зрения машинного языка имя подпрограммы соответствует машинному адресу первой команды тела подпрограммы. Поэтому в C/C++ имена подпрограмм рассматриваются как указатели особого функционального типа. Подробнее этот тип рассматривается в главе 6.

Ссылочные типы данных

Только что мы увидели, что понятие указателя слишком близко к машинному языку, и употребление указателей может привести к хитрым ошибкам в программах. Понятие ссылки тоже является абстракцией адреса, но

лишено недостатков указателя.

В языке C++ ссылка — это аналог имени (можно рассматривать ссылку на объект данных как альтернативное имя объекта). Синтаксис объявления ссылки на данные типа T следующий (сравните с синтаксисом указателя):

```
T & имя_ссылки
```

К ссылкам языка C++ применима единственная операция — инициализация.

Как именно выглядит инициализация ссылки, зависит от контекста, в котором определена ссылка. Например, если ссылка объявлена как переменная (локальная или глобальная), то инициализация выглядит так:

```
T & имя_ссылки = объект_типа_T
```

Например,

```
int k = -1; // объект типа int
int& kk = k; // инициализация ссылки kk
kk++; // это - обращение к k. Теперь k содержит 0
int b = 4;
kk = b; //и это - обращение к k. Теперь k содержит 4
```

После инициализации любое употребление ссылки эквивалентно употреблению объекта, на который она ссылается (из инициализатора).

Другой важный контекст инициализации ссылки, когда формальный параметр функции объявлен как ссылка. В этом случае инициализация формального параметра-ссылки объектом - фактическим параметром - происходит в момент вызова функции (подробнее см. главу 6). Во время выполнения функции формальный параметр обозначает объект - фактический параметр, поэтому операции над формальным параметром выполняются на самом деле над фактическим.

В языках C# и Java ссылки — это единственный способ обращения к

объектам. В этих языках принята *референциальная объектная модель*. Поясним, что она собой представляет.

В C# и Java выделены референциальные типы — это классы, массивы и интерфейсы. Объекты этих типов располагаются только в динамической памяти, являются анонимными и доступны только через ссылки. Объект может быть создан либо с помощью операции `new`, либо путем создания копии объекта («клонирования»). Пример на языке C#:

```
X obj = new X(); // создание нового объекта
X obj = obj.Clone(); // создание копии объекта
```

На языке Java операция создания копии выглядит так:

```
X obj = obj.clone();
```

Остальные типы в языке C# называются типами-значениями. К ним относятся все простые типы данных, а также структуры (структуры языка C# рассмотрим вместе с классами). В Java термина «типы-значения» нет по той причине, что нереференциальные типы — это простые типы данных (за исключением перечислимого типа, который есть класс особого вида). Аналога понятия «структура» в Java нет. Мы будем употреблять термин «типы-значения» для любых нереференциальных типов в этих языках.

Объекты типов-значений могут располагаться в любом классе памяти — статической, квазистатической (то есть в стеке), динамической (в составе объектов референциальных типов).

Пусть X – класс. Объявление «`X a;`» не размещает объект, а объявляет ссылку на X. Начальное значение такой ссылки — пустое. Пустая ссылка обозначается ключевым словом «`null`». Пустая ссылка совместима с любым классом. Рассмотрим следующий фрагмент (он применим как к C#, так и к Java):

```
X a,b; // это не объекты классы, а ссылки
a = new X(); // объект размещен, а ссылается на него
```

```
b = a; // b и a ссылаются на один и тот же объект
a = new X(); // ещё один объект
// a ссылается на второй объект
// b ссылается на первый объект
```

Ссылки C# и Java отличаются от ссылок C++ тем, что ссылка C++ «навсегда», то есть в течение всего времени жизни ссылки, полностью ассоциированы с объектом. Даже если ссылка стоит в левой части операции присваивания, все равно речь идет о присваивании объекту, которым инициализирована ссылка. В C# и Java присваивание ссылке означает новую ассоциацию. Поэтому нельзя считать, что ссылки C# и Java — это альтернативное имя объекта (как в C++).

Однако, и ссылки в C++, и ссылки в C# и Java похожи в том смысле, что после установления ассоциации с объектом ссылка идентична самому объекту, поэтому не требуется никакая операция разыменования (типа операции `*` с указателем).

2. Составные типы данных

Составные типы данных — это типы, значения которых состоят из подобъектов, то есть имеют внутреннюю структуру. Самый популярный составной тип данных (и первый составной тип, появившийся в языках программирования) – это массив.

Одномерные массивы

Массив – это непрерывная последовательность элементов одного типа.

Основная операция, применимая к любому массиву, это операция индексирования, обозначаемая чаще всего как «[ ]». Операция записывается в следующем виде: `A[I]`. Она имеет 2 аргумента: объект-массив A и значение I некоторого дискретного типа (к дискретным типам относятся целые, перечислимые и диапазоны). I называют индексом элемента массива.

Индексирование возвращает ссылку на элемент массива:

```
[ ] : A, I->T&
```

Отметим, что возвращаемое значение — именно ссылка, а не просто значение элемента массива, так как результат операции присваивания может стоять в левой части операции присваивания. Основное требование к операции индексирования — однородность и эффективность. *Однородность* означает, что время выполнения операции не зависит от значения индекса, *эффективность* — то, что операция должна выполняться быстро.

Перечислим атрибуты массива:

- базовый тип (T)
- тип индекса (I)
- диапазон индекса (L и R — нижняя и верхняя граница) и связанная с ним характеристика: длина массива.

Не вдаваясь в обсуждение вариантов реализации массивов в различных языках программирования, отметим, что в большинстве современных языков концепция массива упростилась: тип индекса всегда целый, нижняя граница диапазона индексов (L) — всегда 0, верхняя граница (R) — N-1, где N — длина массива.

Поэтому основной вопрос, касающийся массивов в современных языках, - это время связывания базового типа T и длины массива N с объектом-массивом.

Базовый тип элементов связывается с массивом всегда статически — в объявлении массива. Длина массива N в C/C++ - тоже статическая (и тоже задается в объявлении), а в C# и Java — динамическая и задается при размещении объекта в динамической памяти (то есть при обращении к операции `new`).

Одномерные массивы в языке C++.

Объявление массива выглядит так:

```
T att[LEN];
```

Здесь T — произвольный тип данных, а LEN — статическое (то есть вычисляемое компилятором) выражение. В зависимости от контекста объявления массив размещается либо в статической, либо в динамической памяти. Примеры размещения массива в динамической памяти см. выше в п.5.1.

Массив в C++ не является «полноценным» объектом данных. Это проявляется, например, в том, что один массив нельзя присвоить другому:

```
int a1[256]; int a2[256];
a1 = a2; // ошибка !!!
```

Операция индексирования в языке не контролирует корректность индекса:

```
for (int k = 0; k < 300; k++)
        a1[k] = a2[k];
```

В примере индекс, начиная со значения 256, выходит за верхнюю границу, но ни при компиляции, ни во время выполнения никакой диагностики выдано не будет. Поведение программы при таких ошибках непредсказуемо и зависит от реализации.

Имя массива трактуется в C/C++ как указатель, то есть адрес первого элемента массива (напомним, что индекс первого элемента массива — 0), а операция индексирования тесно связана с адресной арифметикой, а именно: для любого массива `array` и целого значения `i` верны тождества:

```
&(array[i]) ≡ array+i
```

или по другому:

```
array[i] ≡ *(array+i)
```

Как мы уже отмечали, корректность адресной операции не может быть проконтролирована, поэтому и операция индексирования не контролируется,

даже в очевидных случаях, типа `array[-1]`.

Отличие имени массива типа T от указателя типа T* состоит в том, что имя массива трактуется как указатель-константа, поэтому этому имени ничего нельзя присвоить (что вполне резонно, так как иначе доступ к элементам массива может исчезнуть). Более того, любой указатель можно трактовать как имя массива и применять к нему операцию индексирования:

```
int * p; ... p[0] = 0; // то же самое, что *p=0;
```

Поэтому когда операция `new[]` (размещение массива в динамической памяти) возвращает указатель, это вполне согласуется с правилами языка.

Таким образом, понятие указателя перекрывает понятие массива, поэтому программисты на C/C++ нередко предпочитают работать с массивом через указатель (это, как правило, более эффективно). Вот, например, как можно суммировать элементы массива на C/C++:

```
int arr[N];
int * pCurr = arr;
int * pEnd = arr+N;// pEnd указывает ЗА конец массива
int sum = 0;
while ( pCurr < pEnd)sum += *pCurr++;
```

Или ешё короче:

```
for (int *pCurr = arr, *pEnd=pCurr+N, sum =0;
                                    pCurr < pEnd; pCurr+
+)
              sum += *pCurr;
```

C++ - действительно выразительный язык!

Фактически, понятие одномерного массива в C/C++ используется, в основном, как шаблон для размещения последовательностей в статической или квазистатической памяти. Заметим, что ещё одно отличие имени массива от указателя состоит в том, операция `sizeof` для указателя возвращает размер указателя (то есть машинного адреса), а для имени массива — размер всей

последовательности элементов, составляющей массив.

Одномерные массивы в языках C# и Java. Объявление массива в этих языках выглядит так:

```
T [] array;
```

Напомним, что `array` — это только ссылка, а собственно массив размещается так:

```
array = new T[Len];
```

Здесь `Len` — это произвольное целочисленное выражение (не обязательно статическое).

Можно использовать и явную инициализацию:

```
int [] arr;
arr = new int [] {1, 3, 5, 7, 9};
```

Длина массива является *квазидинамическим* свойством, то есть она задается динамически в момент создания, но измениться уже не может. Узнать длину массива во время выполнения программы можно с помощью свойства `Length` в C# и `length` в Java. Приведем пример суммирования элементов массива:

```
double [] arr;
double sum = 0;
for (int i = 0; i < arr.Length; i++) sum += arr[i];
```

Операция индексирования всегда контролируется при каждом обращении к ней. Если индекс выйдет за пределы диапазона 0.. `Length-1`, то генерируется исключительная ситуация (исключения описаны в главе 9) (в Java – `ArrayIndexOutOfBoundsException`, а в C# - `IndexOutOfRangeException`).

В отличие от C/C++ массивы в C# и Java — полноправные объекты. Они

принадлежат к классам специального вида, и оба языка предоставляют большой набор операций над массивами, включающий в себя копирование элементов, поиск значений, слияние и разбиение и тому подобное. В Java эти операции сосредоточены в классах `System` и `Arrays`, а в C# - в классе `Array`.

Ещё раз отметим, что длина массива не может измениться после его размещения. В случае, если нужен массив с переменной длиной, то можно использовать класс `ArrayList` из стандартной библиотеки (он даже называется одинаково для обоих языков). Этот класс позволяет добавлять и удалять элементы из массива. В C++ соответствующий класс из стандартной библиотеки STL называется `vector`.

Многомерные массивы

Многомерные массивы в большинстве языков рассматриваются как массивы массивов (единственное исключение, пожалуй, составляет Фортран), например, двумерная матрица рассматривается как одномерный массив из одномерных массивов. Рассмотрим объявление матрицы на языке C/C++:

```
float matr[N][M];
```

Это одномерный массив длины `N`, каждый элемент которого (то есть строка матрицы) — одномерный массив длины `M`. Такой многомерный массив все равно остаётся непрерывной последовательностью элементов, вначале располагаются элементы первой строки, за ними — второй и так далее. Если последовательно пробегать по элементам многомерного массива, то индексы меняются слева направо (быстрее всех — самый левый индекс), как в следующем примере суммирования элементов трехмерного массива:

```
float m [N1][N2][N3], sum = 0.0;;
for (int i = 0; i < N1; i++)
        for (int j = 0; j < N2; j++)
            for(int k = 0; k < N3; k++)
                sum += m[i][j][k];
```

Такие массивы называют прямоугольными.

Однако в языках с референциальной моделью возникает следующая проблема. Массивы в таких языках представляются ссылками, поэтому, например, для двумерного массива вместо прямоугольного массива мы имеем массив из ссылок на массивы-столбцы матрицы. Массивы уже не обязаны иметь одну и ту же длину. Такие массивы называются ступенчатыми. В C/C++ подобные массивы можно построить, используя массив из указателей. Посмотрим на пример ступенчатого массива в языке Java (он же есть пример на C#):

```
int [][] jagged;
jagged = new int [4][];
// создали массив из ссылок на массивы int
for (int i = 3; i>=0; i--)
         jagged[i] = new int[i+1];
// создали треугольную матрицу
...
int det = 1;
for (int i = 0; i < 5; i++)det *= jagged[i][i];
// вычислили определитель треугольной матрицы
```

В следующем примере массивы выглядят прямоугольными (длина столбцов - одинакова), но это все равно ступенчатые массивы:

```
double [][]a = new double [5][5];
// создали квадратную матрицу 5x5
int [][]b = new int [][] {{1,2,3},{3,4,5}};
// создали и инициализировали массив 2x3
```

В языке Java все многомерные массивы — ступенчатые, создать прямоугольный массив, элементы которого образуют в памяти непрерывную последовательность, в этом языке нельзя. Однако заметим, что доступ к элементам ступенчатого массива менее эффективен, чем к элементам массивов в C/C++ (объясните, почему). По этой причине в язык C# добавлен ещё один вид многомерных массивов — прямоугольные:

```
int [,] a = new int [5,6];
```

```
int sum = 0;
for (int i = 0; i < 5; i++)
        for (int j = 0; j < 6; j++)
            sum += a[i,j];

double [,] x = new double[4,]; // ошибка!!!
int [,] b = {{1,2},{3}}; // ошибка!!!
int [,] c = {{1,2},{3,4},{5,6}};
// создан прямоугольный массив 3x2
```

Необходимость добавления прямоугольных массивов обусловлена двумя главными причинами: эффективностью и возможностью обрабатывать массивы, совместимые с моделью данных языков типа C.

Динамические строки

Динамическая строка (далее будем называть просто «строкой») - это последовательность символов произвольной длины. Длина строки определяется при размещении в памяти и далее не меняется. На первый взгляд, строка может быть реализована (по крайней мере, в языках C# и Java) как массив из символов, однако это не так. Языки C# и Java содержат специальный встроенный тип (точнее, класс) — `String` (в C# можно употреблять вместо этого имени ключевое слово `string`). В C++ встроенного типа строки нет, но зато есть класс `string` из стандартной библиотеки STL.

Необходимость введения специального типа обусловлена несколькими причинами. Во-первых, набор операций для строк существенно шире и специфичней набора операций для обычных массивов. Этот набор включает много вариантов поиска (символа, подстроки, с учетом регистра), сравнения (с учетом разных правил упорядочения) и тому подобных, редко применяемых для массивов.

Кроме того, строки должны интегрироваться со встроенным в язык понятием строковой константы (”`literal constant`”). Сравните инициализацию строки и символьного массива (C#, Java):

```
char [] strArr = {'l','i','n','e'};
String str = ”line”;
```

Но главный аргумент в пользу специальной реализации строкового типа тот, что строки реализуются как *неизменяемый объект*. Это означает, что содержимое строки после её размещения и инициализации рассматривается как константа. Любые операции над строками (даже те, которые манипулируют содержимым строки) не меняют её содержимое, а вырабатывают новое строковое значение. Операция индексирования (`s[i]`) применима к строкам, но в отличие от массивов, она возвращает не ссылку на символ, а значение символа, поэтому для строки `s` нельзя писать `s[0] = ' '`, так как значение не может быть левой частью операции присваивания.

Для обычных массивов модификация отдельных элементов является обычной операцией, также как и сортировка, меняющая порядок элементов и тому подобное.

Зато ряд операций над строками в силу их неизменяемости можно реализовать существенно эффективней, чем операции над массивом символов.

Отметим, что строки — это полноценные классы, обладающие богатым набором операций. Самая частая операция над строками — конкатенация (сцепление) строк, обозначаемая как «+»:

```
String s1  = ”Hello”, s2 = ”world!”, s;
s = s1 + ”, ” + s2; // получилось: Hello, world!
```

Правда, следует отметить, что использовать операцию конкатенацию для конструирования новых строк нужно с осторожностью. Рассмотрим пример на языке C#, который строит строку из входного текста, удаляя из нее все концы строк :

```
string s="", curr;
while ((curr = Console.ReadLine()) != null) s +=
curr;
```

```
Console.WriteLine(s);
```

Здесь мы воспользовались тем фактом, что функция `ReadLine()` из класса `Console` не включает символ конца строки в возвращаемое значение, поэтому этого символа нет в выдаваемом тексте. На Java можно написать аналогичную программу, используя класс `System`.

Проблема в том, что мы используем операцию конкатенации для построения постоянно разрастающейся строки. Если входной текст достаточно длинный, то возникает эффект фрагментации памяти: мы требуем все более длинные куски памяти, освобождая куски, которые короче. Конечно, автоматический сборщик мусора исправит ситуацию, но производительность программы существенно упадет. Неслучайно и C#, и Java содержат стандартный класс StringBuilder, который специально спроектирован для конструирования строк возрастающей длины. Он работает с памятью более оптимально, чем реализация операции «+». После завершения конструирования строки достаточно вызвать функцию `ToString` (в Java `toString`) для получения сконструированной строки:

```
string curr;
StringBuilder sb = new StringBuilder();
while ((curr = Console.ReadLine()) != null)
        sb.Append(curr);
Console.WriteLine(s);
```

Записи (структуры)

В традиционных императивных языках программирования, таких как Паскаль или C, записи (в C они называются структурами) наряду с массивами используются очень широко.

Запись — это совокупность объявлений переменных, которые объединены в отдельный объект. Эти переменные называются *полями записи* и доступны с помощью операции «точка». Приведем пример структуры на C++:

```
struct Complex
```

```
{
        double Re, Im;
}; // объявление структуры с двумя полями
Complex ImagOne = {0.0, -1.0};
        // объявление переменной типа структуры
        // с одновременной инициализацией
Complex c1, c2; // две переменных типа структуры
c1.Re = 1.0; c2.Im = 0.0; // Обращение к полям
c2 = c1; // структуры можно присваивать
```

Понятие класса в объектно-ориентированных языках заведомо шире и универсальней понятия записи, поэтому в современных языках структуры либо упразднены (как в Java), либо являются частным случаем класса (как в C++). В C# есть понятие структуры, но мы рассмотрим его вместе с понятием классов C#

Другие составные типы данных

Ряд языков программирования содержит другие составные типы данных, отличные от массивов и записей. Например, язык Паскаль включает в себя понятия файла и множества.

Что касается типов (и операторов), абстрагирующих ввод/вывод, то большинство языков программирования не содержат в базисе соответствующие конструкции. Весь ввод/вывод отнесен к стандартным или специализированным библиотекам. Причина этого — разнообразие и эволюция устройств ввода/вывода. Фиксация каких-либо концепций ввода/вывода в конструкциях языка неизбежно приведет к неадекватности этих конструкций в более поздних реализациях. Например, файлы языка Паскаль отлично подходили к самым популярным устройствам хранения данных в 60-х годах 20 века (это ленточные устройства). Однако концепция файлов языка Паскаль совершенно не соответствовала интерактивному вводу/выводу, который стал стандартом, начиная с 80-х годов. Поэтому ряд диалектов Паскаля (например, Турбо Паскаль) игнорировал эти конструкции, вводя вместо них свои понятия.

Сменить библиотеку значительно проще, чем реализацию.

То же самое относится к введению в базис языка различного рода контейнеров, то есть структур данных, предназначенных для хранения и выборки данных. Современные языки индустриального программирования содержат единственный контейнер — массив. Более специализированные контейнеры предоставляются стандартными библиотеками.

Это связано с тем, что не существует универсального и абсолютно лучшего для любых применений способа реализации контейнеров (это относится и к алгоритмам). Поэтому в стандартную библиотеку включают реализации контейнеров, подходящие «для большинства» приложений. Если же стандартная реализация не подходит, то можно использовать вместо стандартной специализированную библиотеку с тем же стандартизованным интерфейсом.

3. Инкапсуляция. Абстрактные типы данных

Инкапсуляция – это языковой механизм, позволяющий ограничить доступ к отдельным членам класса. Иногда инкапсуляцию называют *упрятыванием* или *защитой* информации. Инкапсуляция позволяет скрыть внутреннее представление объекта, обеспечивая его целостность, за счет того, что пользователь не может «испортить» внутренние данные объекта.

Например, недостатком класса Stack из п.7.1 является то, что его данные не инкапсулированы. Поэтому можно, например, изменить значение переменной top, присвоив ей нулевое значение и «опустошив» тем самым весь стек. Также можно непосредственно модифицировать тело стека (массив body) и тому подобное.

Рассмотрим, как реализована инкапсуляция в объектно-ориентированных языках программирования.

Во-первых, в языках с классами возможно ограничение доступа к

отдельным членам класса. Единицей инкапсуляции (или единицей защиты) является член класса. При этом правила защиты применяются единообразно ко всем членам, будь то данные, методы, специальные функции-члены, вложенные классы.

Во-вторых, атомом защиты является весь класс целиком, что означает, что правила защиты применяются единообразно ко всем экземплярам класса. Нельзя устанавливать отдельные правила доступа к экземпляру класса, отличные от правил доступа к другим экземплярам этого класса.

Инкапсуляция в языке C++. В языке C++ все члены класса относятся к одной из *областей доступа*:

- `public` – открытый, доступный любым функциям;
- `protected` – защищенный, доступный только собственным методам и методам производных классов;
- `private` – закрытый, доступный только собственным методам;

Члены класса, находящиеся в закрытой области (`private`), недоступны для использования со стороны внешних функций. Напротив, члены класса, находящиеся в открытой секции (`public`), доступны для использования из любых функций, в том числе и внешних. При объявлении класса каждый член класса помещается в одну из перечисленных выше областей доступа следующим образом:

```
class  имя_класса  {
 private:
определение_закрытых_членов_класса
     public:
          определение_открытых_членов_класса
     protected:
          определение_защищенных_членов_класса
     …
};
```

Порядок следования областей доступа и их количество в классе –

произвольны.

Ключевое слово, определяющее первую область доступа, может отсутствовать. Умолчание зависит от того, с какого ключевого слова начинается объявление класса.

Объявление класса может начинаться с ключевого слова `class` (как выше) или `struct`.

Класс C++ отличается от структуры C++ **только** определением по умолчанию первой области доступа в их описании (а также определением по умолчанию способа наследования, см. раздел 8.1):

- для структур умолчанием является открытый доступ (`public`)
- для классов умолчанием является закрытый доступ (`private`).

Различия в умолчаниях связаны с обеспечением совместимости программ на языке C с языком C++. Они позволяют рассматривать «старые» структуры в программах на C как классы C++ без функций-членов и с открытыми данными (то есть «плохие», но все-таки классы).

Модель управления доступом, основанная на трех уровнях доступа (только для класса, только для иерархии классов, для любых классов и функций), является достаточно простой. Однако эта модель иногда является слишком ограничительной, так как не позволяет различать права доступа для внешних функций и классов. «Чужаки» (то есть внешние классы и функции) либо все сразу имеют доступ (к открытым членам), либо все сразу не имеют доступа (к закрытым и защищенным). На практике выясняется, что иногда некоторые внешние классы или функции нельзя рассматривать как «чужаков».

Для примера рассмотрим класс `String`, инкапсулирующий структуру динамической строки. Все ли операции с этим классом имеет смысл реализовывать через его методы? Например, удобно определить операцию конкатенации (сцепления) двух строк и использовать для неё перегрузку стандартной операции сложения «+».

Есть два варианта перегрузки операции «+». Можно перегрузить её как функцию-член:

```
class String {
public:
      String operator + (const String & S);
          ... // другие члены, в том числе закрытые
};
```

А можно перегрузить и как внешнюю функцию:

```
String operator + (const String &S1, const String &S2);
```

Обращение к операции в обоих случаях выглядит одинаково:

```
String m1, m2, m3;
m1 = m2 + m3; //
```

Но компилятор трактует вызов операции «+»по-разному:

```
m1 = m2.operator + (m3); // функция-член
m1 = operator + (m2, m3); // внешняя функция
```

Допустима только одна интерпретация, которую должен выбрать программист.

В первом варианте первый параметр сложения играет выделенную роль, во втором варианте оба параметра равнозначны. Второй вариант лучше отвечает общепринятой семантике сложения, а это важно при перегрузке стандартных операций (если перегруженная операция не отвечает общепринятой семантике, то это верный путь к ошибкам в программе).

Кроме того, второй вариант более универсален, например, он позволяет единообразно трактовать неявные преобразования. Поясним это на примере класса `String`.

Предположим, что класс `String` имеет конструктор преобразования из вещественного типа, который переводит число в текстовое представление:

```
String(double d); // слово explicit отсутствует!
```

Такой конструктор аналогичен методу `toString()` языка Java. Этот

метод применим ко всем объектам данных (даже простых типов). Если контекст использования объекта требует строкового значения, то компилятор Java автоматически подставляет вызов метода `toString()` для объекта. Таким образом, можно считать, что метод `toString()` реализует неявное преобразование объекта данных языка Java в строку (и это единственное неявное преобразование, допустимое в Java). Итак, в Java можно писать:

```
String s = ”Line ”;
String ss = s + 6; // ss получает значение ”Line 6”;
```

Но теперь (при наличии конструктора преобразования) такой код верен и для класса `String` в языке C++ (и это достигнуто чисто языковыми средствами). Компилятор подставляет цепочку из неявных преобразований – одно пользовательское, второе – стандартное:

```
String ss = s + (String)(double)6;
```

А теперь вернемся к вопросу, как перегружена операция сложения для класса String? Если функцией-членом, то приведенный выше пример работает корректно:

```
String ss = s.operator+((String)(double)6);
```

А вот перестановка операндов приводит к ошибке:

```
String ss = 6 + s;
```

Если же перегрузить внешней функцией, то преобразование будет работать корректно в обоих случаях:

```
String ss = operator+(s, (String)(double)6);
String ss = operator+((String)(double)6, s);
```

Таким образом, можно привести общую рекомендацию: двуместные «симметричные» операции, в которых оба аргумента равноправны лучше перегружать как внешние функции, а «асимметричные» операции с выделенным левым операндом (к такому типу относятся, например,

комбинированные операции присваивания типа «+=») - функциями -членами.

Однако, если выбрать второй вариант, то правило инкапсуляции мешает – внешняя операция не может получить доступ к закрытым переменным. Это пример функции, которая, с одной стороны, должна быть внешней по отношению к классу, а с другой - должна иметь доступ ко всем членам класса. Таким образом, модель языка C++ нуждается в расширении: некоторые внешние функции должны иметь «привилегированный» доступ.

Такое расширение реализовано в C++ посредством *друзей класса*.

Друг класса X - это внешняя функция (глобальная, либо член другого класса), имеющая такие же права доступа к членам X, как и у членов этого класса. Понятие друга декларируется самим классом («друзей не навязывают»), понятие друга не является транзитивным («друг моего друга необязательно мой друг»), друзья класса не наследуются (если производный класс хочет иметь в друзьях друзей базового класса, то он должен явно их объявить).

Объявление друга должно содержаться внутри объявления класса, объявляющего друга. Объявление друга имеет три модификации:

```
friend прототип_глобальной_функции;
friend прототип_функции_члена_класса;
friend имя_класса;
```

Примеры:

```
class X
{
  friend void f(X&);
      friend void Y::AccessX(X&);
  friend class Z; // все функции-члены Z – друзья X
};
class String
{ ...
 friend String operator + (const String &, const String &);
  ...
};
```

Инкапсуляция в языках C# и Java. В языках C# и Java имеются такие же

уровни доступа (и такие же ключевые слова), как и в C++: `public`, `protected`, `private`, но, кроме того, модель управления доступом дополнена другими уровнями, которые исключают необходимость в друзьях.

Дело в том, что в C# и Java можно набор логически связанных классов объединить в некоторую сущность и использовать эту сущность как целое (например, распространять, импортировать из неё классы и тому подобное). Эта сущность играет роль библиотеки (классов). Конечно, оттранслированные классы на C++ тоже можно объединять в библиотеки, но понятие «библиотеки» на языковом уровне нет ни в C, ни в C++. Приходится использовать соответствующее понятие из виртуального компьютера операционной системы.

В языке C# роль библиотеки играет понятие сборки (assembly), а в Java - понятие пакета (package).

Исходные тексты программ как на C#, так и на Java хранятся в текстовых файлах (расширение .cs и .java, соответственно). Напомним, что программы состоят только из определений типов (классов, перечислений, интерфейсов). Принадлежность класса сборке (C#) определяется при трансляции в командной строке, вызывающей компилятор C# (интегрированная среда разработки сама генерирует командную строку из установок проекта). В Java есть специальная конструкция package, которая указывает, какому пакету принадлежит класс (классы) из файла программы. Эта конструкция должна быть первой в файле. Форма конструкции очень проста:

```
package имя-пакета;
```

Например:

```
Package ru.soft-company.common.graphics;
```

Поскольку классы из сборок (пакетов) должны быть логически связаны, то они имеют привилегированный доступ друг к другу по сравнению с классами из внешних сборок (пакетов).

В результате, появляется ещё один уровень доступа (промежуточный между закрытым и открытым). В языке C# этот уровень доступа называется внутренним (и обозначается ключевым словом `internal`). В Java – аналогичный уровень называется пакетным, он является умолчательным и не имеет ключевого слова. Внутренний (пакетный) уровень доступа означает разрешение доступа со стороны всех классов, входящих в сборку (пакет).

Внутренний (пакетный) доступ позволяет обойтись без понятия «друг класса». Полного аналога друга класса в C# и Java нет, поскольку закрытый доступ в этих языках запрещает доступ любым другим классам. Однако, если в сборке (пакете) есть классы, которые нуждаются во взаимном доступе (например, функции преобразования в C#), то нужно обеспечить внутренний (пакетный) доступ к членам классов.

Все классы в сборке (пакете) считаются логически связанными друг с другом (отсюда и особый уровень доступа). Здесь возникает тонкий момент, связанный с понятием наследования. Являются ли производные классы логически связанными с классами из «родительской» сборки (пакета)? Понятно, что производный и базовый классы тесно связаны, поэтому и возникает (во всех языках) защищенный уровень доступа. Но если классы в сборке логически связаны с базовым классом, то они должны быть связаны и с производными классами, но насколько тесно? Язык Java разрешает всем классам из пакета иметь доступ к защищенным членам классов из этого же пакета (тем самым ослабляя понятие защищенного уровня по сравнению с C++). Понятие защищенного доступа в языке C# аналогично C++, но появляется ещё один (пятый) уровень доступа, эквивалентный защищенному уровню Java: внутренний защищенный (`protected internal`), который определяется как «внутренний ИЛИ защищенный».

Теперь окончательно уточним определения уровней доступа.

Язык C#:

- `public` – открытый (неограниченный), доступный методам любых

классов из любых сборок;

- `internal` – внутренний, доступный методам всех классов из этой же сборки;
- `protected internal` – внутренний или защищенный, доступный только собственным методам, методам производных классов (из любых сборок) и методам всех классов из этой же сборки;
- `protected` защищенный, доступный только собственным методам и методам производных классов (из любых сборок);
- `private`-закрытый, доступный только собственным методам;

Примечание: строго говоря, можно было бы ввести и ещё один уровень доступа - внутренний И защищенный, то есть доступный только собственным методам и методам производных классов из этой же сборки. Такой уровень есть в промежуточном языке платформы .NET и в реализации C++/CLI для этой платформы, но авторы C# не стали включать его в язык.

Уровни доступа в языке Java:

- `public` – открытый (неограниченный), доступный методам любых классов из любых пакетов;
- (нет ключевого слова) - пакетный, доступный методам всех классов из этого же пакета;
- `protected` –защищенный, доступный только собственным методам, методам производных классов из любых пакетов и методам всех классов из этого же пакета;
- `private`-закрытый, доступный только собственным методам;

Синтаксически в C# и Java нет областей доступа, так что модификатор доступа распространяется только на член, перед которым он стоит. При отсутствии модификатора доступа в C# подразумевается `private` (как для

классов, так и для структур), а в Java – пакетный.

В заключение отметим, что в C# и Java есть возможность управлять доступностью классов из сборки (пакета), отсутствующая в C++.

Если перед классом стоит модификатор `public`, то он доступен из любой сборки (пакета). Отсутствие модификатора означает доступность класса только изнутри сборки (пакета).

Абстрактные типы данных

Вспомним, что в современных языках программирования тип данных определяется как пара: множество значений и множество операций.

Если тип данных рассматривать как класс, то множество значений определяется набором членов-данных, а множество операций – набором методов класса.

Абстрактный тип данных (АТД) – это тип, в котором внутренняя структура данных полностью инкапсулирована. Другими словами, с точки зрения пользователя абстрактный тип данных представлен только множеством операций. Класс является абстрактным типом данных, если открытыми членами являются только методы.

Говорят, что совокупность открытых членов класса составляет *интерфейс* класса. Поэтому интерфейс АТД представлен только операциями.

Конечно, некоторые методы могут быть закрытыми, их называют вспомогательными. Главное, чтобы структура класса (члены-данные) была скрыта.

Объектно-ориентированная парадигма подразумевает широкое использование АТД. Некоторые языки, например, SmallTalk, вообще запрещают открытые члены-данные, тем самым любой класс является в этом языке абстрактным типом данных.

Языки C++, C#, Java позволяют открывать члены-данные, но это считается «дурным тоном» с точки зрения объектно-ориентированного стиля. Даже если некоторые операции сводятся к присваиванию и/или считыванию значения некоторой переменной – члена класса, то и в этом случае предлагается использовать не открытый доступ к члену-данному, а методы класса, называемые *селекторами*. Функции-селекторы – это пара функций, одна из которых (функция get) считывает значение члена-данного, а другая (функция set) присваивает новое значение. Отсутствие одной из функций селекторов означает запрещение соответствующей операции. Например, класс Vector на C++ (см. выше п. 7.2) может содержать get-селектор, возвращающий длину вектора, но set-селектора в классе нет, поскольку длина вектора не меняется в процессе его жизни:

```
class Vector
{
     public:
          ...
          int GetLength() { return size; }
          ...
};
```

Функции-селекторы подчеркивают дуализм данных и операций, поскольку позволяют абстрагироваться от того, как именно реализована сущность – как данное, или как операция. Важно, что функции-селекторы могут реализовываться не только как чтение-запись некоторого данного, но и как полноценные операции, содержащие нетривиальные вычисления. Внутреннее устройство селекторов недоступно и неинтересно пользователю.

Язык C# поддерживает абстракцию функций-селекторов, вводя конструкцию «свойство» (property).

Свойство синтаксически выглядит, как член-данное класса. Обращение к свойству неотличимо от обращения к члену-данному (за некоторыми исключениями). Объявление свойства выглядит как объявление члена-данного, сразу за которым следует объявление get- и set-селекторов в фигурных скобках. При присваивании свойству значения вызывается set-селектор, в его теле

присваиваемое значение доступно через идентификатор `value` (в контексте тела селектора `value` – это ключевое слово, а в других контекстах - идентификатор). При считывании свойства вызывается get-селектор, который должен вернуть значение свойства. Простой пример:

```
class PropSample
{
          private int _datum = 0;
          public int Datum // выглядит как член-данное
          {    // но это свойство
               get { return _datum; } // чтение
               set { _datum = value; } // запись
          }
}
PropSample ps = new PropSample();
ps.Datum = -1; // вызов set-селектора с value=-1
Console.WriteLine(ps.Datum); // вызов get-селектора
```

Часто свойства доступны только на чтение (не содержат set-селектора), что позволяет сохранить целостность объекта. Приведем пример класса Stack из п.7.1 на языке C# (обратите внимание на свойства):

```
public class Stack
{
          int [] body;
          int top = 0;
          public Stack(int size)
          {
               body = new int[size];
          }
          public int Pop() { return body[--top]; }
          public void Push(int x) { body[top++] = x; }
          public bool Empty
      {
           get { return top ==0;}
      }
          public bool Full
      {
           get {return top == body.Length;}
      }
      public int Length
      {
          get { return body.Length; }
      }
}
```

В заключение отметим, что понятие АТД выходит за рамки объектно-

ориентированного подхода и широко используется и в других парадигмах. Ещё один подход к реализации понятия АТД с объектно-ориентированной точки зрения представлен интерфейсами и обсуждается в следующей главе.

4. Интерфейсы и абстрактные классы

При проектировании полиморфных иерархий классов часто возникает понятие абстрактного класса.

Абстрактный класс – это класс, который предназначен исключительно для того, чтобы быть базовым классом. Экземпляры абстрактного класса нельзя создавать, но можно (и нужно) использовать ссылки (указатели) на абстрактный класс. Интерфейс абстрактного класса используется для работы с объектами производных классов, на которые указывают ссылки абстрактного типа. Абстрактные классы тесно связаны с полиморфными иерархиями.

В объектно-ориентированных языках абстрактные классы реализуются тремя способами:

- Перед классом ставится модификатор abstract. Такой способ используется в языках C# и Java.
- Класс содержит хотя бы один абстрактный метод. Абстрактный метод – это виртуальный метод без тела. Он должен быть обязательно замещен в одном из классов-наследников. В языках C# и Java абстрактный метод указывается модификатором abstract перед объявлением метода (модификатор нужно тогда поставить и перед объявлением класса). В C++ абстрактные методы называются чистыми виртуальными функциями.
- Если в классе, производном от абстрактного класса с абстрактными методами или интерфейса (об интерфейсах см. ниже), не замещен хотя бы один абстрактный метод, то класс тоже является абстрактным. В C# и Java незамещенные абстрактные методы должны быть явно объявлены как абстрактные.

В языке C# абстрактными могут быть и свойства:

```
abstract int Length { get;}
```

Это означает, что соответствующая функция-селектор (в данном случае только get-селектор) является абстрактным методом.

Поясним понятие абстрактного метода, поскольку именно наличие абстрактных методов чаще всего приводит к появлению абстрактных классов. В C# и Java возможно объявление абстрактных классов без абстрактных методов, но это используется относительно редко (а в C++ абстрактных классов без абстрактных методов вообще нет).

При проектировании иерархий классов для некоторой проблемной области полезно соблюдать следующее правило: следует выбирать базовый класс – вершину иерархии - максимально обобщенным (в рамках проблемной области). Иногда возникают настолько общие классы, что невозможно указать реализацию некоторых их методов. Эти методы имеют смысл для конкретных классов-наследников, но не для абстрактного класса – базового в иерархии. Такие методы и являются абстрактными.

Пример проблемной области, хорошо иллюстрирующей понятие абстрактных классов и методов – экранные графические объекты.

Какими свойствами должен обладать каждый графический объект (для краткости будем называть графический объект фигурой)? Все эти свойства нужно вынести в базовый класс иерархии. Например, к этим свойствам относится точка привязки - она есть у каждой фигуры. Точку привязки будем обозначать целочисленными координатами. Также общим является поведение фигуры: способность каждой фигуры отрисовывать себя, менять размеры, перемещаться по экрану и так далее. Поведение должно реализовываться методами:

```
void Draw();// отрисовать объект
```

```
void Resize(); // изменить размер
void Move (int dx, int dy); // сместиться на (dx, dy)
```

Как можно реализовать эти методы для произвольной фигуры? Понятно, что методы Draw() и Resize() невозможно сформулировать в терминах произвольной фигуры, потому они слишком специфичны (одна реализация для точки, вторая для отрезка, третья для окружности и так далее).

А вот универсальная реализация метода Move() в принципе возможна, например, можно изменить координаты точки привязки и послать уведомление об изменениях (чтобы прикладная программа, использующая фигуры, перерисовала бы содержимое экрана).

Итак, метод `Move()` – пример конкретного метода. Его можно заместить (если приведенная выше реализация не подойдет), но у него есть своя осмысленная реализация, которую можно использовать. Другой случай - методы `Draw()` и `Resize()` – для них невозможно привести универсальную и осмысленную реализацию. Поэтому эти методы удобно сделать абстрактными.

При добавлении конкретной фигуры в иерархию необходимо заместить все абстрактные методы класса(то есть написать конкретную реализацию методов `Draw()` и `Resize()`). Если этого не сделать, то производный класс останется абстрактным.

Невозможно вызвать несуществующую реализацию `Draw()` и `Resize()` для базового класса, поскольку невозможно создать объект абстрактного класса (транслятор контролирует это статически). Таким образом, механизм абстрактных классов хорошо защищён.

Приведем пример иерархии классов фигур (см. Рис. 8-2) на языке Java:

```
public abstract class Figure
{
          int x,y;
      ...
          public void Move (int dx, int dy)
          {
               // реализация метода
          }
```

```
            public abstract void Draw(); // тело отсутствует
            public abstract void Resize(); // тело отсутствует
            ...
}
class Circle extends Figure {
      void draw() { /* конкретная реализация */ }
      void resize(){ /* конкретная реализация */ }
            ...
}
class Point extends Figure {
      void draw() { /* конкретная реализация */ }
      void resize(){ /* конкретная реализация */ }
            ...
}
... // другие классы
```

На языке C++ абстрактные классы реализуются как классы с чистыми виртуальными методами. Чистый виртуальный метод (ЧВМ) – это метод с объявлением:

```
void Draw() = 0; // тело отсутствует
```

Компоновщик не требует наличия тела у ЧВМ (для всех остальных виртуальных методов реализация обязательна). Также как и для других объектно-ориентированных языков компилятор запрещает создание экземпляров абстрактных классов.

Интерфейсы

Тесно связано с понятием абстрактного класса понятие интерфейса. Интерфейс можно рассматривать как абстрактный класс, доведенный до «абсолюта». Интерфейс состоит только из абстрактных методов. Поскольку реализации методов нет (равно как нет и невиртуальных методов), то интерфейс не имеет нестатических членов (статические члены, например, константы допустимы).

Интерфейс представляет собой «чистый» контракт. Производный класс, наследуя интерфейс, «подписывается» под контрактом. Производный класс, наследующий интерфейс и замещающий все его методы, называют *реализацией*

интерфейса.

Удобство понятия интерфейса состоит в том, что он не накладывает никаких ограничений на реализацию. Например, абстрактный класс Figure, приведенный выше, этим свойством не обладает и накладывает определенные ограничения на то, как надо реализовывать производный класс (учет точки привязки, например). А вот интерфейс может быть реализован любым классом, главное, чтобы все абстрактные методы замещались реализацией. Поэтому программы, в которых классы взаимодействуют друг с другом только посредством явно объявленных и документированных интерфейсов, являются очень гибкими и открытыми для расширений. Например, современные текстовые процессоры объявляют некоторый набор интерфейсов, который позволяет любой программе, корректно реализующей эти интерфейсы, вставлять в текст документа произвольные объекты, манипулировать ими и визуализировать их.

Рассмотрим, как интерфейсы реализованы в современных языках.

Реализация интерфейсов в C++. Собственно конструкции «интерфейс» в Си++ нет, но его можно смоделировать. Интерфейс на Си++ - это класс, в котором нет никаких нестатических данных, все операции являются публичными чистыми виртуальными функциями. Статические члены могут присутствовать в классе-интерфейсе, поскольку не имеют отношения они к экземпляру класса. Существует единственное исключение из правила: «все методы интерфейса - абстрактные». Это исключение – деструктор. Мы уже обсуждали необходимость виртуального деструктора в полиморфных объектах. Заметим, что деструктор C++ не может быть абстрактным (попробуйте объяснить, почему). В классе-интерфейсе деструктор – виртуальная пустая функция, нужная только для размещения строки в ТВМ.

Рассмотрим пример класса-интерфейса C++, описывающего «контракт» понятия «множество» объектов типа T (предполагаем, что тип T доступен в точке объявления интерфейса):

```
class Set{
     public:
      virtual void Incl(T & x) = 0;
      virtual void Excl(T & x) = 0;
               virtual bool IsIn(T & x) = 0;
              // другие абстрактные методы
          virtual ~Set() {} // деструктор
};
```

Для реализации интерфейса нужно выбрать структуру данных для внутреннего представления множества. Пусть, например, мы выбрали класс «линейный список» `SList` для представления множества. В C++ есть, по крайней мере, две возможности реализации интерфейса. В первом случае используется «композиция объектов»: класс-реализация `SetImpl` содержит объект класса `SList` как член-данное. Другой - множественное наследование, допустимое в C++, причем класс-интерфейс наследуется открытым образом, а класс `SList`.- закрытым. Закрытое наследование делает недоступным в классе `SetImpl` внутреннее представление множества, что необходимо с точки зрения инкапсуляции.

```
class SetSListImpl : public Set, private SList
{
     public:
      void Incl(T & x);
      void Excl(T & x);
          bool IsIn(T & x);
        // объявление заместителей других абстрактных методов
          ~ SetSListImpl ();
};
```

Теперь надо только написать функцию-генератор:

```
Set * MakeSet ()
{
          return new SetSListImpl ();
}
```

Чтобы совсем инкапсулировать реализацию класса, можно объявить конструктор умолчания класса `SetSListImpl` приватным, а функцию-генератор – другом класса `SetSListImpl`. В этом случае обращение к `MakeSet`() – это единственная возможность создать объект-множество. При

этом внешние классы вообще ничего не знают о структуре реализации интерфейса. Если реализация по каким-то причинам не подходит, например, по причинам эффективности, то можно создать новую реализацию и сменить функцию-генератор:

```
class SetBitScaleImpl : public Set, private BitScale
{
     private:
     SetBitScaleImpl() {}
// только для запрещения несанкционированного создания
friend Set * MakeSet ();
     public:
      void Incl(T & x);
      void Excl(T & x);
          bool IsIn(T & x);
        // объявление заместителей других абстрактных методов
          ~ SetBitScaleImpl ();
};
Set * MakeSet ()
{
          return new SetBitScaleImpl();
}
```

Исходные тексты классов-клиентов (пользователей интерфейса `Set`) менять не надо (не надо даже перетранслировать), достаточно только скомпоновать их объектный код с объектными модулями, содержащими новую реализацию. Реализация интерфейса полностью инкапсулирована.

Таким образом, понятие интерфейса очень хорошо сочетается с понятием абстрактного типа данных (см. п.7.3).

Отметим, что за счет множественного наследования класс C++ может реализовывать произвольное количество интерфейсов.

Реализация интерфейсов в C# и Java

Эти языки, в отличие от C++, явно поддерживают интерфейсы на языковом уровне. Одна из причин этого то, что в них не реализовано множественное наследование в полном объеме.

C# и Java поддерживают одиночное наследование класса и множественное

наследование интерфейсов. Не вдаваясь в детали, отметим, что множественное наследование классов с членами-данными и виртуальными методами создает некоторые проблемы при реализации. Этих проблем нет, если наследуемые множественным образом классы могут иметь только (абстрактные) методы.

Таким образом, в C# и Java есть конструкция «интерфейс».

Интерфейс состоит только из объявлений прототипов методов. Модификаторов доступа нет, так как скрывать надо только реализацию, а методы интерфейса обязаны быть открытыми. Из данных лдпускаются только статические члены-константы. Допускаются ещё объявления вложенных интерфейсов (для «композиции» интерфейсов).

Примеры объявлений интерфейсов:

```
interface IEnumerable  // стандартный интерфейс C#
{
          IEnumerator GetEnumerator();
}
interface IEnumerator  // стандартный интерфейс C#

{
          object Current { get; }
          bool MoveNext();
          void Reset();
}

interface Iterable  // стандартный интерфейс Java
{
          Iterator iterator();
}
interface Iterator // стандартный интерфейс Java
{
          boolean hasNext();
          Object next();
          void remove();
}
```

Примеры объявлений классов, реализующих интерфейсы:

```
class MyCollection : IEnumerable // C#
{  ....
          public IEnumerator GetEnumerator()
          { .... /* реализация метода */  }
}
```

```
class MyCollection implements Iterable // Java
{  ....
      public Iterator iterator()
      { /* реализация метода */  }
}
```

Это примеры интерфейсов (из стандартной библиотеки C# и Java), которые объявляют «контракт» итераторов, позволяющих последовательно перебирать элементы коллекции.

Класс может реализовывать произвольное количество интерфейсов:

```
class Sample extends Base implements I1, I2, I3, I4 {...}
class SampleCS : Base, I1, I2, I3, I4 {...}
```

Объектов-интерфейсов нет, и это понятно почему: ведь интерфейс – обобщение понятия «абстрактный класс», а объекты абстрактного класса создавать нельзя. Однако можно получать ссылку на интерфейс (точнее, на реализацию интерфейса) из ссылки на класс, реализующий этот интерфейс. Ссылки на интерфейс можно использовать для вызова методов интерфейса.

Таким образом, ссылка на интерфейс синтаксически выглядит и ведет себя как ссылка на объект базового класса. Этой ссылке можно присваивать ссылку на объект класса, реализующего интерфейс (это похоже на неявное преобразование из производного класса в базовый). Также её можно явно, и только явно преобразовывать к ссылке на объект класса, реализующего интерфейс (это похоже на явное преобразование из базового в производный класс).

Пример:

```
void RemoveLongStrings (MyCollection coll, int maxLen)
 Iterable obj = coll;
 // ссылку на коллекцию поместили в интерфейс
 Iterator i = obj.iterator();
 while (i.hasNext())
 {
          String s = (String)i.next();
          if (s.length() > maxLen )
               i.remove();
 }
}
```

Эта функция удаляет из коллекции `coll` все строки с длиной больше заданной величины. Если коллекция содержит нестроковые объекты, то преобразование `(String)i.next()` сгенерирует исключение (см. главу 9).

Интеграция интерфейсов в язык программирования

Как уже отмечалось, интерфейсы – мощное средство интеграции программ, настолько мощное, что его можно использовать и для интеграции механизмов языка и пользовательских классов. C# и Java поддерживают ряд стандартных интерфейсов, позволяющих интегрировать семантику языковых конструкций и классы, реализующие эти интерфейсы.

Например, приведенные выше интерфейсы итераторов интегрируют класс-коллекцию, реализующие эти интерфейсы с циклом `foreach`:

```
MyCollection coll;
...
int sum = 0;
for (int i : coll)
          sum += i;
```

Компилятор языка Java вставит обращения к интерфейсам `Iterable` и `Iterator`, а также распаковку из `Object` в `int`.

Другой пример – интерфейс `IDisposable`, используемый в using-блоке языка C#. Он обсуждается в главе 9.

Язык Java использует понятие пустых интерфейсов (интерфейсов-маркеров), которые специально предназначены для интеграции с транслятором. Интерфейсы-маркеры не содержат никаких методов, их смысл зафиксирован в документации и воплощен транслятором. Например, если класс реализует интерфейс-маркер Cloneable, то это означает, что класс будет реализовывать открытый метод `clone()` создания своей копии:

```
class CloneAttack implements Cloneable
{
          public Object clone()
      {
          // возвращает свою копию
```

```
        }
    }
```

Тут следует отметить, что любой класс уже содержит версию метода `clone()`, унаследованную от класса `Object`. Этот метод возвращает поверхностную копию объекта. Однако проблема в том, что этот метод - защищенный, поэтому может использоваться только из производных классов или из классов своего пакета. Для того, чтобы позволить копирование себя внешним классам и используется интерфейс `Cloneable`.