Лекция 2.
Динамическая память. Указатели и ссылки. Строки С

СОДЕРЖАНИЕ

1 ПРИНЦИПЫ РАБОТЫ С ПАМЯТЬЮ В ЯЗЫКЕ C++

1.1 Подход C++ к использованию памяти

1.1.1 Обзор подхода

Язык C++ унаследовал практически все черты языка C, а тот, в свою очередь, разрабатывался как высокоуровневая (независимая от платформы) замена ассемблеру, в котором все обращения к памяти ЭВМ производились явно. Принцип «презумпции ненужности» препятствовал появлению сугубо языковых (недоступных для модификации программисту) средств работы с памятью. С другой стороны, тот факт, что стандарт C++ описывает выполнение программы на абстрактной машине, побудил создателей языка разработать гибкую структуру организации памяти с точки зрения C++, которая как не была бы привязана к аппаратной платформе, так и оставляла бы разработчикам компиляторов и программ максимум свободы выбора реализации. Из всего названного следуют характерные черты подхода C++ к использованию памяти:

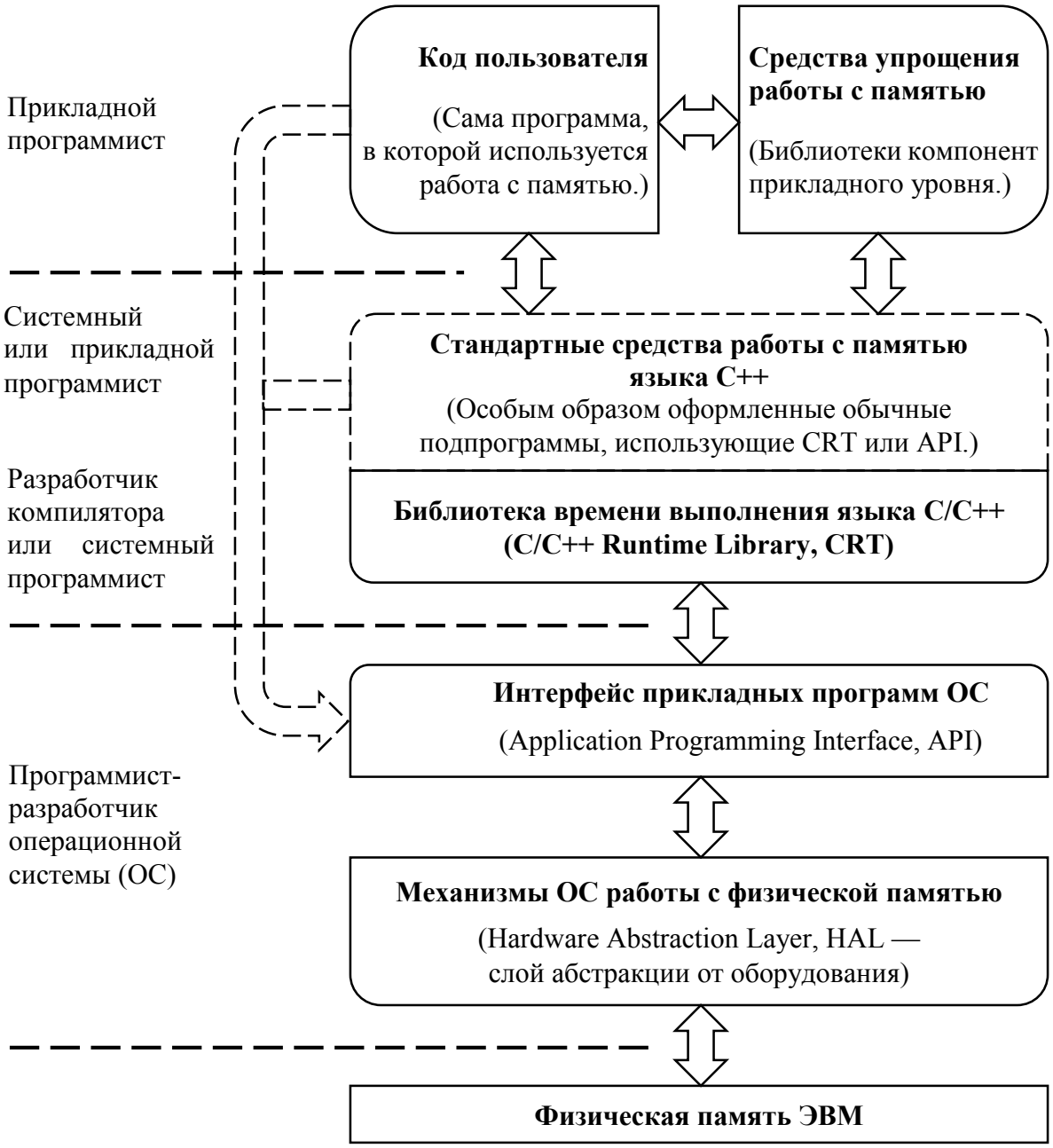
а) Всегда доступны низкоуровневые средства (операции), позволяющие пользоваться фактом расположения всех данных в памяти ЭВМ.

б) Обеспечивается надежная абстракция от физической организации памяти ЭВМ и методов работы ОС с ней.

в) Все важные элементы процесса работы с памятью позволяется модифицировать прикладному программисту.

г) Возможно (и применяется) создание и использование на прикладном уровне механизмов, упрощающих управление памятью, например, автоматическое освобождение по окончании использования.

1.1.2 Схема организации работы с памятью в C++



Современные операционные системы (ОС) полностью обеспечивают управление ресурсами ЭВМ, поэтому прикладным программам физическая память напрямую недоступна. В ядре ОС имеются средства, обеспечивающие единообразную работу с любой физической памятью, — часть *слоя абстракции от оборудования (Hardware Abstraction Layer, HAL).* Для прикладного программиста он недоступен.

Кроме того, многие современные ОС предлагают прикладным программам работать не с реальной памятью, а с *виртуальной (virtual memory),* которая является упрощенной моделью ОЗУ. Например, каждое 32-разрядное приложение в ОС семейства MS Windows NT и выше может работать так, как будто ему монопольно доступно 4 Гб ОЗУ,

вне зависимости от наличия такого объема физической памяти ЭВМ и работы других приложений. Внутри ОС обеспечивает работу виртуальной памяти на базе HAL.

Прикладные программы могут взаимодействовать с механизмами ОС по работе с памятью посредством *интерфейса прикладных программ ОС (Application Programming Interface, API)*. Он представляет собой набор функций в библиотеках, написанных программистами-разработчиками ОС и документирован.

Так как API отличается в разных ОС, а стандартная библиотека C++ едина, разработчик компилятора должен предоставить реализацию стандартной библиотеки под конкретную платформу с использованием API конкретной ОС. Это называется *библиотекой времени выполнения языка C++ (C++ Runtime Library),* и она включает не только средства работы с памятью, но и функции ввода-вывода, обработки ошибок и т.д. Обычно она неявно подключается и активно используется компилятором, но принципиально возможно написание программ и без нее — тогда взаимодействие с ОС потребуется реализовывать самостоятельно. Такая возможность используется редко и требует соответствующей квалификации системного программиста.

Операторы языка C++ для работы с памятью определены таким образом, что являются простыми функциями со специальным синтаксисом, которые обычно действуют при помощи библиотеки времени выполнения (часто они включаются в нее). В большинстве случаев программа на C++ использует именно их. Прикладной или системный программист может переопределить эти функции для своих нужд (например, фиксировать вызовы для контроля за расходом памяти).

В прикладных программах и стандартной библиотеке C++ широко распространены средства упрощения работы с памятью. Например, можно сделать и делается так, чтобы участок памяти автоматически освобождался, когда он перестает использоваться в программе. Кроме того, часто предусмотрены способы гибкой настройки работы с памятью для отдельных частей приложения. Примером может служить оптимизация размещения в памяти большого числа мелких объектов, которая «прозрачна» для кода контейнера, где эти объекты хранятся. Реализация перечисленного выполняется прикладным программистом на базе стандартных средств C++, библиотеки времени выполнения и системного API.

Благодаря такому количеству возможностей для модификации механизма работы с памятью программы на C++, обеспечивается большая гибкость управления столь важным ресурсом. Это одна из причин, по которой C++ является популярным для высокопроизводительных и низкоуровневых приложений.

1.2 Организация памяти ЭВМ с точки зрения C++

1.2.1 Модель памяти C++

Независимо от физического исполнения ОЗУ ЭВМ и от организации памяти, предлагаемой ОС, с точки зрения языка C++ память ЭВМ всегда описывается одинаково. Это описание называется *моделью памяти C++*.

Наименьшая единица памяти в C++ — байт. Память, доступная программе на C++, состоит из последовательностей байт (ненулевой длины), называемых *областями памяти (memory location)*. У каждого байта памяти есть уникальный адрес в памяти, который принято записывать как шестнадцатеричное число с префиксом «0x» [1, подраздел 1.7].

1.2.2 Стек и куча (динамическая память)

Среди областей памяти различают, в частности, стек (stack) и кучу (heap).

Область стека может служить для хранения значений переменных, а также использоваться компилятором при вызове функций и для промежуточных вычислений. Размер каждой области памяти, отводимый под переменную на стеке, известен на этапе компиляции. Количество памяти стека ограничено компилятором и, как правило, значительно меньше размера кучи (на платформе x86 для компилятора MSVC++ размер стека по умолчанию равен 1 МБ).

Корректный доступ к стеку осуществляется только посредством использования значений переменных. Так как стек может использоваться компилятором произвольно, изменение байт стека непосредственно по их адресам ведет к неопределенному поведению. На практике, так как компилятор часто хранит на стеке адреса инструкций, которые надлежит выполнить, последствия порчи стека могут быть самыми непредсказуемыми и трудно диагностируемыми.

Куча (динамическая память) — это область, части которой могут быть зарезервированы (выделены, allocated) программой во время выполнения для размещения данных, а также удалены (освобождены, deallocated), когда они больше не требуются. Размер областей памяти, выделяемых в куче, может быть известен только на этапе выполнения.

Область памяти, размещенная в куче, остается зарезервированной даже в том случае, если в программе не остается способов к ней обратиться или освободить ее. Такая ситуация называется *утечкой памяти (memory leak)* и нежелательна, поскольку количество свободной (доступной для выделения) памяти сокращается без возможности восстановления.

1.2.3 Время жизни данных и способы их хранения

Время жизни объекта (object lifetime) — это период во время выполнения программы, когда данные находятся в памяти. Любые свойства данных справедливы только в этот период. Время жизни простых типов данных начинается с момента, когда для них (компилятором или программистом) выделена область памяти, а заканчивается с освобождением (явным или неявным) этой области [1, подраздел 3.8].

Минимальное время жизни данных называется *временем хранения (storage duration),* оно зафиксировано в стандарте C++ и всегда известно [1, подраздел 3.7].

Статическое время хранения означает, что данные находятся в памяти на протяжении всего времени выполнения программы [1, п. 3.7.1]. Им обладают данные в глобальных переменных (если не используется динамическая память), и в локальных, если они объявлены с ключевым словом **`static`** (например, **`static int i = 0`**).

Автоматическое время хранения длится до тех пор, пока выполняется блок, в котором данные были размещены в памяти [1, п. 3.7.3]. Этим временем хранения обладают по умолчанию локальные переменные и все данные вне динамической памяти.

Динамическое время хранения применяется к данным, размещенным в областях памяти, выделенных программистом явно. Оно длится до тех пор, пока эта область памяти не будет явно освобождена [1, п. 3.7.4]. Выделение и освобождение памяти программистом осуществляется специальными операторами C++ (см. 3.5).

2 ССЫЛКИ

Ссылка (reference) — это переменная, связанная с другой переменной таким образом, что обращение к ссылке равносильно обращению к переменной [1, п. 8.3.2].

Удобно использовать ссылки в тех случаях, когда не желательно копирование объекта при передаче в функцию по значению или во время присваивания. Также ссылки используются для организации выходных параметров функций.

```
1       #include <cmath>
2       #include <cstdio>
3
4       bool solveQuadricEquation(
5            double a, double b, double c, double& x1, double& x2)
6       {
7            double D = b*b - 4*a*c;
8            if (D < 0.0f) {
9                 return false;
10           } else {
11                x1 = (-b + sqrt(D)) / (2*a);
12                x2 = (-b - sqrt(D)) / (2*a);
13                return true;
14           }
15      }
16
17      int main() {
18           double root1 = 0.0f, root2;
19           if (solveQuadricEquation(1.0f, 3.0f, 2.0f, root1, root2)) {
20                printf("Корни x^2 + 3*x + 2 = 0: %.2f и %.2f\n",
21                     root1, root2);
22           } else {
23                puts("Действительные корни уравнения не найдены!");
24           }
25           return 0;
26      }
```

Листинг 1. Пример использования ссылок.

Функция `solveQuadricEquation()`, объявленная на строке 4, пытается решить в действительных числах квадратное уравнение с коэффициентами `a`, `b` и `c`; если это возможно, она передает значения найденных корней выходным параметрам `x1` и `x2` и возвращает **`true`**, иначе просто возвращает **`false`**, не изменяя значений выходных переменных. Тип выходных параметров `x1` и `x2` — ссылка на переменную типа **`int`** (**`int&`**).

На строке 17 происходит вызов функции `solveQuadricEquation()` с передачей ей локальных переменных `root1` и `root2` по ссылке.

Значения переменных доступны по ссылкам на них. Так например, если бы код функции `solveQuadricEquation()` обращался к значению `x1` до его присвоения, было бы получено корректное значение `0.0f`. Следует помнить, однако, что значения переменных до первого присвоения непредсказуемы: значение, которое могло бы быть получено при обращении в коде функции к `x2` (до строки 10) непредсказуемо.

3 УКАЗАТЕЛИ

3.1 Понятие об указателях и размещении данных в памяти

Указатель — это переменная, содержащая адрес памяти.

Необходимо различать значение переменной-указателя (адрес) и значение, на которое переменная указывает (данные). Рисунок 1 демонстрирует пример расположения данных в памяти. Рассмотрим важнейшие свойства указателей на этом примере.

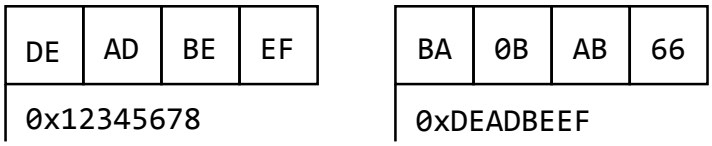


Рисунок 1. Пример расположения данных в памяти.

Начиная с адреса `0x12345678` (адреса принято записывать в шестнадцатеричной системе), в памяти расположены 4 байта, которые могут соответствовать представлению числа `0xDEADBEEF`. Если `0x12345678` является адресом переменной-указателя, то значение по этому адресу (`0xDEADBEEF`) следует воспринимать как адрес другой ячейки памяти. Начиная с последней, в памяти расположены другие 4 байта: `0xBA`, `0x0B`, `0xAB` и `0x66`.

3.1.1 Классификация указателей

Неизвестно, как нужно интерпретировать последние 4 байта. Если это целое число со знаком (**`int`**), это значение (−1173640346), а если целое число без знака (**`unsigned int`**), — значение 3121326950. С другой стороны, это зачастую ясно при написании программы, поэтому в C++ указатели бывают двух видов:

а) Типизированные, для которых на этапе компиляции известен тип данных, расположенных по адресу, который будет во время выполнения храниться в переменной-указателе. В Delphi это такие типы, как `Integer^`, `Real^` и т. п.

б) Нетипизированные, о которых компилятору «известно» только то, что данная переменная хранит адрес памяти. В Delphi это тип `Pointer`.

При работе с типизированными указателями компилятор отслеживает правильность использования значения, расположенного по адресу в указателе, и автоматически преобразует его в нужный тип.

Можно заметить, что две рассматриваемые области памяти отличаются друг от друга лишь условно: интерпретацией как значения указателя и как значения обычной переменной. Более того, можно обе области памяти интерпретировать как значения

указателей, получив указатель на указатель. Такая конструкция действительно возможна и применяется. Указатели на указатели на указатели и т. д. возможны, но на практике почти не используются.

3.1.2 Нулевой указатель

Есть специальное значение указателя, *нулевой указатель*, которое не соответствует никакой ячейке памяти. Удобно, например, присваивать нулевой указатель переменной, которая не содержит адреса каких-либо данных. Нулевой указатель обозначается `0` или **`nullptr`** (начиная с C++11), а в старых программах на C — как `NULL` (совместимо с C++). Прямой аналог в языках Pascal и Delphi — значение **`NIL`**. Нулевой указатель может быть присвоен указателю любого типа, а в условиях (**`if`**, **`while`** и т. п.) преобразуется в **`false`** [1, подраздел 4.10].

3.1.3 Порядок байт в памяти

Если представление (representation) значения некоторого типа данных занимает область памяти из нескольких байт (например, 32-битное число), возникает вопрос, в каком порядке эти байты располагаются. Выбранный способ размещения называется *порядком байт (byte order или endianness).*

Например, 16-битное число `0xABCD` состоит из двух байт: `0xAB` и `0xCD`. Если это число расположено по какому-то адресу, то в ячейке с этим адресом можно разместить байт `0xCD`, и в ячейке с адресом на единицу больше — `0xAB`, то есть записать байты *от младшего к старшему (little-endian).* Можно поступить и наоборот, записав расположив байты *от старшего к младшему (big-endian).* Тот же вопрос возникает при чтении значения из памяти, и ответы должны совпадать в обоих случаях. В первом примере подразумевается big-endian (см. рис. 1).

Порядок байт зачастую важен при организации взаимодействия программы со внешним окружением, например:

а) при загрузке двоичных файлов (описание формата зачастую включает указание порядка байт);

б) при обработке текста (порядок байт может специально указываться в тексте или быть предметом соглашения);

в) при сетевом взаимодействии (данные по сети почти всегда передаются в big-endian, из-за чего этот порядок называют иногда «сетевым»).

Выбор порядка байт в ЭВМ делается на аппаратном уровне или на уровне ОС. С точки зрения C++, порядок байт определяется компилятором и обязательно

документируется производителем. Для процессоров семейства Intel x86 принят little-endian (который из-за этого называют иногда «порядком Intel»).

Существует также *смешанный порядок байт (middle-endian),* когда байты в пределах машинного слова записываются в одном порядке, а сами машинные слова — в обратном, но такой способ применяется редко, неудобен в работе и здесь не рассматривается.

3.2 Операции над указателями

3.2.1 Разыменование и взятие адреса

С указателями естественным образом связаны две операции [1, п. 5.3.1]:

а) взятие адреса переменной, который можно занести в указатель;

б) доступ к данным в ячейке памяти по адресу, хранящемуся в указателе (разыменование, dereference).

Нетипизированные указатели разыменовывать нельзя, потому что неизвестно, как интерпретировать значение, на которое они указывают. Разыменование нулевого указателя приводит к неопределенному поведению.

Рассмотрим синтаксис C++ для работы с указателями на примере (Листинг 2).

```
float value = 10.0f;
float *address = &value;
float anotherValue = *address;
anotherValue += 20.0f;
*address = *address + 30.0f;
int wrongTypeValue = *address; // Не компилируется.
```

Листинг 2. Пример использования операций взятия адреса и разыменования указателя.

Целочисленная переменная `value` расположена на стеке и имеет значение 10.

В следующей строке объявляется переменная `address` типа «указатель на число с плавающей запятой» (`float*`) и ей присваивается адрес переменной value.

Символ амперсанда (`&`) и есть оператор взятия адреса, а астериск (`*`) перед именем переменной — синтаксис объявления указателей. Существует множество типов данных, являющихся типизированными указателями — `int*`, `char*` и т. д., но только один вид нетипизированных указателей — `void*`. Астериск ставится перед каждым идентификатором в списке, который требуется объявить, как указатель, — в противном случае, он будет объявлен как переменная типа `тип`. Поэтому не рекомендуется объявлять несколько указателей одним выражением, равно как и одним выражением — переменные-указатели и обычные.

В целочисленную переменную `anotherValue` при инициализации записывается значение, находящееся по адресу, содержащемуся в `address`. Чтобы обратиться именно к этому значению (целочисленному), а не к значению указателя (адресу), используется оператор разыменования в виде астериска (*).

Если теперь увеличить значение `anotherValue` на 20, изменится только эта переменная, поскольку она не является указателем, и ее значение расположено в собственной области памяти.

Чтобы изменить значение `value` косвенным образом, нужно воспользоваться оператором разыменования. Будучи задействован в правой части присваивания, он позволит получить значение переменной `value`. Когда оператор разыменования используется в левой части оператора присваивания, это означает, что результат будет помещен по адресу, содержащемуся в указателе. Таким образом, значение переменной `value` увеличивается.

Попытка присвоить результат разыменования указателя типа **`float`*** целочисленной переменной `wrongTypeValue` приведет к ошибке компиляции. Это и есть контроль транслятором действий с типизированными указателями.

3.2.2 Указатели на массивы и адресная арифметика

Существует неявное преобразование между переменной-массивом и указателем на первый (с индексом 0) элемент этого массива [1, подраздел 4.2]. С другой стороны, операция доступа к элементу массива (`массив[индекс]`) может быть применена к указателю [1, п. 5.2.1]. Эти особенности связаны с т. н. адресной арифметикой.

Адресная арифметика — это мощный и выразительный инструмент для работы с указателями в языках C и C++. Она представляет собой набор правил выполнения арифметических операций над указателями и целыми числами. Адресная арифметика — одна из причин популярности C и C++ для низкоуровневого программирования.

Сложение типизированного указателя `T* p` с целым числом `n` дает указатель `(p + n)`, сдвинутый относительно `p` на `n` элементов типа `T`. Таким образом, `*(p+n) == p[n]` для любого типизированного указателя. На основании предыдущего правила также действуют операторы `+=`, `-=`, `++`, `--`. Вычитание указателей на элементы одного массива дает число элементов между ними, а тип этой разности — `ptrdiff_t` [1, подраздел 5.7]. Указатели можно сравнивать так же, как обычные целые числа, включая «больше — меньше» [1, подразделы 5.9—5.10].

3.2.3 Оператор `sizeof`

Оператор **`sizeof`** позволяет во время компиляции вычислить размер своего аргумента в байтах [1, п. 5.3.3]. В частности, с его помощью можно узнать размер массива:

```
1  int array[10];
2  int *end = array + 10;
3  int arraySizeBySizeof = sizeof(array) / sizeof(array[0]);
4  int arraySizeByOffset = end - array;
```

В первой строке объявляется массив `array` размером 10 элементов. Во второй строке объявляется переменная `end`, указывающая на последний на ячейку памяти сразу за последним элементом массива (последний — `array[9]`). Узнать размер массива можно двумя способами:

а) Вычислить размер всего массива при помощи оператора **`sizeof`** и разделить его на размер одного элемента, который тоже вычисляется **`sizeof`** (строка 3).

б) Вычесть из указателя `end` указатель на начальный элемент массива (строка 4).

3.3 Приведение `reinterpret_cast`

С указателями и ссылками связан особый вид приведения типов: **`reinterpret_cast`**. Его смысл — рассмотрение области памяти, доступной по указателю или по ссылке, как данных заданного типа. Например, можно область памяти, содержащую беззнаковое целое число (**`unsigned int`**) рассмотреть как знаковое целое (**`int`**):

```
unsigned int source = 0xFEDCBA98;
int *destination = reinterpret_cast<int*>(&source);
```

Листинг 3. Пример использования `reinterpret_cast`.

На платформе x86 (32 бита) старший бит знакового целого используется для хранения знака числа. Так как в беззнаковом целом `0xFEDCBA98` (`1111`…) он равен 1, то значение `*destination` отрицательное.

Такие приведение типов должно использоваться только для низкоуровневых операций, когда программисту точно известно содержимое используемого блока памяти и его интерпретации на данной платформе. Например, данные по сети или из файла поступают в виде массива байт, и без рассмотрения нескольких байт как единого значения не обойтись. Накладываются минимальные ограничения: **`reinterpret_cast`** возможен только меду порядковыми типами, указателями и ссылками.

При помощи операции разыменования и взятия адреса возможно выполнить и приведение в стиле C (C-style cast), подобное **`reinterpret_cast`**, но полностью без ограничений на типы данных: `*((T*)&v)`. При этом берется адрес переменной `v` (тип —

указатель), рассматривается как указатель на `T`, после чего он разыменовывается (тип результата — `T`). Такое приведение типов является наиболее опасным, однако распространено на практике, особенно в коде, написанном изначально на C, где не было **`reinterpret_cast`**.

3.4 Отличия указателей от ссылок

Использование указателей и ссылок схоже. Указатели предоставляют больше возможностей для косвенной адресации, но ссылки использовать часто проще и безопаснее. Отличия указателей от ссылок:

1) Не существует «нулевой ссылки», подобной нулевому указателю.
2) Не существует ссылок на ссылки, указателей на ссылки и массивов ссылок (см. 5)
3) Не бывает переменных типа **`void`**, поэтому не бывает и ссылок на **`void`**.
4) Разыменование ссылки не имеет смысла (ссылка — не адрес, а псевдоним).
5) У ссылки может не быть собственного адреса, поэтому операция взятия адреса ссылки срабатывает для переменной, на которую ссылаются.
6) Ссылки отсутствуют в языке C.

3.5 Выделение и освобождение динамической памяти

Для выделения области в динамической памяти служат операторы **`new`** (для одного элемента) и **`new[]`** (для массива), а для освобождения — **`delete`** и **`delete[]`** соответственно [1, п. п. 5.3.4 — 5.3.5], например:

```
int length = 10;
int *dynamicArray = new int[length];
int *dynamicObject = new int;
delete [] dynamicArray;
delete dynamicObject;
```

В примере выделяются 2 блока памяти: `dynamicArray` под 10 целых чисел и `dynamicObject` под одно целое число, а затем они освобождаются.

Если память была выделена оператором **`new`** или **`new[]`**, освобождать ее необходимо соответственно операторами **`delete`** или **`delete[]`**. Освобождение памяти другим оператором из пары приводит к неопределенному поведению, даже если размер массива, выделенного **`new`**, был 1. Обращение к области памяти после ее освобождения ведет к неопределенному поведению.

При освобождении памяти оператором **`delete[]`** не требуется указывать размер области памяти или массива. Компилятор обеспечивает освобождение области памяти

нужного размера. Однако, нет штатных способов получить размер выделенного блока памяти, имея только указатель на него.

3.5.1 Операторы `new` и `delete` как функции

Как говорилось ранее, C++ предоставляет возможности всестороннего управления процессами работы с памятью. На самом деле, операторы **`new`** и **`delete`** — это просто сокращенные формы (разг. *синтаксический сахар, syntax sugar*) для вызова особых функций, в частности [1, п. 18.6.1]:

```
void* operator new     (size_t size);
void  operator delete  (void* ptr) noexcept;
void* operator new[]   (size_t size);
void  operator delete[](void* ptr) noexcept;
```

Здесь «`operator new`» и т. п. — имена функций, а смысл **`noexcept`** сейчас не важен. Функции выделения памяти принимают количество элементов и возвращают указатель на выделенный блок, а функции освобождения памяти указатель на такой блок принимают.

Так как функции можно перегружать, по умолчанию определены варианты `operator new` и `operator new[]` с другими параметрами:

```
void* operator new  (size_t size, void* ptr) noexcept;
void* operator new[](size_t size, void* ptr) noexcept;
```

Передавать оператору **`new`** дополнительные параметры можно перед именем типа:

```
new (параметры) имя-типа;
new (параметры) имя-типа [количество-элементов];
```

По умолчанию разрешается передавать один параметр: адрес `ptr`, по которому следует разместить блок памяти, как будто он выделяется в куче:

```
int pool[10];
int *item = new (pool) int;
```

В примере памятью под `item` будет начальный элемент массива `pool`. Заметим, что не требуется вызов **`delete`**, так как `pool` выделен на стеке, и будет освобожден автоматически.

Данная возможность называется *placement new* (обычно не переводится, иногда — «размещающий **`new`**») и применяется вот зачем. Выделение и освобождение динамической памяти — дорогостоящая (в смысле времени) операция, так как в этом процессе может участвовать и сама программа, и библиотека времени выполнения, и ядро ОС, и драйверы. Поэтому, если нужно обеспечить быстрое выделение памяти какому-либо алгоритму, разумно выделить один раз крупный блок (его называют пул), а затем вместо реального выделения памяти **`new`** использовать ячейку пула. Placement new в сочетании с перегрузкой

оператора **new** и позволяет это реализовать, причем алгоритм сможет пользоваться **new** как обычно (говорят: оптимизация «прозрачна»). Перегрузка операторов рассмотрена далее в курсе.

3.5.2 Выделение памяти в стиле C

В языке C нет операторов **new** и **delete**, а работа с динамической памятью реализована в стандартной библиотеке (файл `<cstdlib>`) функциями [1, п. 20.8.13], [2, подраздел 7.11]:

`void* malloc(size_t size)` — для выделения блока памяти размером `size` байт;

`void* calloc(size_t size)` — `malloc()` с обнулением выделенной памяти;

`void* realloc(void* pointer, size_t size)` — для выделения блока памяти размером `size` с копированием в него данных из блока `pointer` в новый (если старый размер был меньше);

`void free(void* pointer)` — для освобождения блока памяти `pointer`.

Они использовались следующим образом:

```
size_t length = 10;
int *array = (int*)malloc(sizeof(int) * length);
free(array);
```

Как можно видеть из примера, требовалось приведение типов и забота программиста о том, чтобы правильно указать размер в байтах, что создавало риск ошибок. Более того, пара `malloc()` и `free()` не равносильна **new** и **delete**, кроме как для самых простых типов, и не обладает гибкостью этих средств C++. Поэтому применять средства C работы с динамической памятью в C++ нет никаких причин. Здесь о них рассказывается потому, что они могут встретиться в реальных программах в коде, изначально написанном на C.

4 СТРОКИ С

4.1 Представление строк С и работа с ними

В языках С и С++ нет встроенного типа для работы со строками. Вместо них применяются массивы символов (**char** и других) и указатели на такие массивы. В стандартной библиотеке С++ имеются и другие средства работы со строками, отсутствующие в С (класс `std::string` и другие).

Строкой С называется не всякий массив символов, а только такой, в котором конец отмечен специальным символом с кодом 0 (символ `'\0'`) — *завершающим нулем* (NUL-terminator). Их полное название — последовательности символов с завершающим нулем (NUL-terminated character sequences, NTCS). Область памяти, отведенная под строку С, может не заканчиваться на ячейке с нулевым символом — он служит меткой конца строки.

В С++ нет специальных языковых конструкций для работы с NTCS. Так например, сложение двух переменных типа **char*** оператором **+** приведет к сложению указателей, а не содержимого строк, а оператор `==` сравнивает указатели (адреса) массивов символов, а не массивы посимвольно. Вместо этого в стандартной библиотеке С++ определены функции для работы со строками С (файлы `<cstring>`, `<ctype>`, `<cstdlib>` и другие [1, подраздел 21.18], [2, подраздел 7.24]).

Рассмотрим пример работы со строками С при помощи функций стандартной библиотеки С++:

```
1   #include <cstring>
2   …
3   if (strlen(first) != strlen(second)) {
4       puts("Длины строк не равны.");
5   } else {
6       printf("Длины строк равны, ");
7       if (strcmp(fisrt, second) == 0)
8           puts("и сами строки равны.");
9       else
10          puts("а сами строки не равны.");
11  }
```

В примере `first` и `second` — строки С. Работа с ними ничем не отличается от работы с другими типизированными указателями, а библиотечные функции служат лишь для удобства.

4.2 Кодировки символов

Как известно, в памяти ЭВМ хранятся не символы, а их коды, а соответствие кодов и символов называется *кодировкой (encoding)*. Тип данных **`char`** определен так, чтобы переменная могла содержать код любого символа из некоего *набора символов (character set),* считающегося базовым в реализации компилятора [1, п. 3.9.1]. Как правило, это ASCII, где каждый символ представляется одним байтом. Существуют кодировки, для которых этого размера недостаточно, например, UTF-16 (два байта на любой символ строки) или UTF-8 (от 1-го до 6-и байт на символ в одной строке). Кроме того, некоторые кодировки чувствительны к порядку байт [3, 4].

Для работы со строками в кодировках, отличных от кодировки по умолчанию, могут применяться массивы иного типа (например, **`wchar_t`** — тип, вмещающий самое длинное представление символа для данной платформы), и в этом случае следует пользоваться другими функциями для их обработки (например, `wcslen()`). Имеются и функции для преобразования кодировок строк.

C++11 вводит дополнительные языковые средства для работы с кодировками — *префиксы строковых литералов (string literal prefix)* [1, п. 2.14.5]. Например, можно в исходном файле с любой кодировкой создать константу в UTF-8:

```
const char *encoded = u8"Текст в кодировке UTF-8";
```

Переменная `encoded` не содержит сведений о том, в какой кодировке строка, так как префиксы строк действуют только на этапе компиляции.

Основные особенности обработки строк с учетом кодировки:

а) Невозможен доступ к символу в произвольной позиции без просмотра всех предыдущих символов, так как у каждого символа может быть разный размер в байтах.

б) Размер строки в байтах может изменяться неявно и зависеть от взаимного расположения символов. Классический пример: «ß» (лигатура эсцет) в верхнем регистре — «SS», а если после «ß» стояло «s» — «SZ», то есть, «Straße» (6 символов, 7 байт) → «STRASSE» (7 символов, 7 байт).

в) В строке могут быть служебные байты, не относящиеся к символам. Например, строки в UTF-16 часто включают т. н. *метку порядка байт (byte-order mark)* — два байта в начале, `0xFF` и `0xEF`, сигнализирующие об используемом порядке байт.

г) Сравнение символов должно выполняться с учетом всех их байт, языка и региона. Например, строка «ß» равна «ss» для Германии и не равна

для Венгрии. Для работы с региональными настройками в стандартной библиотеке C++ предусмотрен механизм *локалей (locale)* — наборов правил форматирования для страны и региона [1, раздел 22].

При работе со строками всегда нужно знать, к какой они кодировке и с каким порядком байт. В учебном курсе работа со строками для простоты всегда ведется в кодировке по умолчанию.

5 БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. ISO/IEC 14882:2011 Programming Languages — C++. — Введен в 2011 г. — 1338 с.
2. ISO/IEC 9899:2011 Information technology — Programming languages — C. — Введен в 2011 г. — 683 с.
3. Николай «Кодт» Меркин. Многоликий Юникод. — 3 октября 2003 г. — Режим доступа: http://rsdn.ru/forum/other/400134.1, свободный.
4. Сайт Joel on Software. — Joel Spolsky — The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!). — 8 октября 2003 г. — Режим доступа: http://www.joelonsoftware.com/articles/Unicode.html, свободный.