МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ
УНИВЕРСИТЕТ

ИНСТИТУТ ВЫЧИСЛИТЕЛЬНОЙ МАТЕМАТИКИ И
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ
Кафедра системного анализа и информационных технологий

Долгов Дмитрий Александрович

Основы программирования. Работа с памятью в языке программирования C++.

Методическое пособие

КАЗАНЬ, 2025

УДК 004.42
ББК 32.973

Рекомендовано к изданию
Учебно-методической комиссией ИВМиИТ КФУ
(Протокол № 8 от 18 апреля 2025 года)

Рецензенты:
заведующий кафедры системного анализа и информационных технологий КФУ, доцент, кандидат физико-математических наук **А.В. Васильев**;
кандидат физико-математических наук, доцент кафедры алгебры и математической логики КФУ **М.М. Ямалеев**

Долгов Д.А.

Основы программирования. Работа с памятью в языке программирования C++. / Д.А. Долгов. – Казань: Изд-во Казан. ун-та, 2025. – 19 с.

Методическое пособие предназначено для проведения практических занятий по курсу «Основы программирования» для студентов, обучающихся по направлениям «Фундаментальная информатика и информационные технологии», «Информационная безопасность». В первой части методического пособия даются необходимые основы работы с памятью в языке программирования C++: указатели и ссылки. Представлено большое количество примеров на языке программирования C++.

УДК 004.42
ББК 32.973

Содержание

1. Введение

C++ – компилируемый, статически типизированный язык программирования общего назначения. Язык C++ многим обязан языку C, который сохраняется как его подмножество. Сохранены и все свойственные C средства низкого уровня, предназначенные для решения самых насущных задач системного программирования. Выбор C в качестве базового языка для C++ объясняется следующими его достоинствами:

1) универсальность, краткость и относительно низкий уровень;

2) адекватность большинству задач системного программирования;

3) он идет в любой системе и на любой машине;

4) полностью подходит для программной среды UNIX.

Первоначально язык C задумывался как конкурент ассемблера, способный вытеснить его из основных и наиболее требовательных к ресурсам задач системного программирования. В проектах на C++ были приняты меры, чтобы успехи C в этой области не оказались под угрозой. Различие между двумя языками прежде все состоит в степени внимания, уделяемого типам и структурам. Язык C выразителен и в то же время снисходителен по отношению к типам. Язык C++ еще более выразителен, но такой выразительности можно достичь лишь тогда, когда типам уделяют большое внимание. Когда типы объектов известны, транслятор правильно распознает такие выражения, в которых иначе программисту пришлось бы записывать операции с утомительными подробностями. Кроме того, знание типов позволяет транслятору обнаруживать такие ошибки, которые в противном случае были бы выявлены только при тестировании. Отметим, что само по себе использование строгой типизации языка для контроля параметров функции, защиты данных от незаконного доступа, определения новых типов и операций не влечет дополнительных расходов памяти и увеличения времени выполнения программы.

В проектах на C++ особое внимание уделяется структурированию программы. Если у вашей программы в 10 000 строк плохая структура, то вы обнаружите, что новые ошибки появляются в ней так же быстро, как удаляются старые. C++ создавался с целью, чтобы большую программу можно было структурировать таким образом, чтобы одному человеку не пришлось работать с текстом в 25000 строк. В настоящее время можно считать, что эта цель полностью достигнута.

В C++ можно использовать различные типы объектов, которые различаются по использованию памяти. Так, глобальные объекты создаются при запуске программы и освобождаются при ее завершении. Локальные автоматические объекты создаются в блоке кода и удаляются, когда этот блок кода завершает работу. Локальные статические объекты создаются перед их первым использованием и освобождаются при завершении программы.

В C++ можно создавать динамические объекты. Продолжительность их жизни не зависит от того, где они созданы. Динамические объекты существуют, пока не будут удалены явным образом. Динамические объекты размещаются в динамической памяти. Это область памяти, не занятая операционной системой или другими загруженными в данный момент программами.

Использование динамических объектов имеет ряд преимуществ. Во-первых, более эффективное использование памяти - выделяется имеенно столько места, сколько необходимо, а после использования сразу освобождается. Во-вторых, мы можем использовать гораздо больший объем памяти, который в ином случае был бы не доступен. Но это и накладывает ограничения: мы должны следить, чтобы все динамические объекты были удалены.

В данном методическом пособие описывается работа с памятью в языке программирования C++.

2. Объекты и переменные

Программируя на языке C++, мы создаем, обрабатываем и уничтожаем объекты. Объект — это часть памяти, которая может хранить значение. В качестве аналогии мы можем использовать почтовый ящик, куда мы помещаем информацию и откуда её извлекаем. Все компьютеры имеют оперативную память, которую используют программы. При создании объекта, часть оперативной памяти выделяется для этого объекта.

```
a=8;
```

Эта строка выглядит довольно просто: мы присваиваем значение 8 переменной a. Но что такое a? a — это переменная, объект с именем.

```
int a;
```

При выполнении этой инструкции центральным процессором часть

оперативной памяти выделяется под этот объект. Например, предположим, что переменной a присваивается ячейка памяти под номером 150. Когда программа видит переменную a в выражении или в операторе присваивания, то она понимает, что для того, чтобы получить значение этой переменной, нужно заглянуть в ячейку памяти под номером 150.

Итак, переменная — это название кусочка памяти, который содержит значение. Нам не нужно беспокоиться о том, какие конкретно адреса памяти выделены для определенных переменных. Мы просто ссылаемся на переменную через присвоенный ей идентификатор, а компилятор конвертирует это имя в соответствующий адрес памяти.

2.1. Оператор адреса &

Оператор адреса & позволяет узнать, какой адрес памяти присвоен определенной переменной.

```
#include <iostream>
int main(){
   int a = 7;
   std::cout << a << '\n';
   std::cout << &a << '\n';
   return 0;}
```

2.2. Оператор разыменования *

Оператор разыменования * позволяет получить значение по указанному адресу.

```
#include <iostream>
int main(){
  int a = 7;
  std::cout << a << '\n';
  std::cout << &a << '\n';
  std::cout << *&a << '\n';
  return 0;}
```

Первый cout выводит значение переменной a, второй выводит адрес переменной a, а третий выводит значегие ячейки памяти переменной a.

2.3. Указатели

Указатель — это переменная, значением которой является адрес ячейки памяти. Указатели объявляются точно так же, как и обычные переменные, только со звёздочкой между типом данных и идентификатором. Синтаксически язык C++ принимает объявление указателя, когда звёздочка находится рядом с типом данных, с идентификатором или даже посередине. Обратите внимание, эта звёздочка не является оператором разыменования. Это всего лишь часть синтаксиса объявления указателя. Тип указателя должен соответствовать типу переменной, на которую он указывает!

```
int *iPtr;
double *dPtr;
int* iPtr3;
int * iPtr4;
int *iPtr5, *iPtr6;
int* iPtr3, iPtr4;
```

iPtr – указатель на значение типа int. dPtr – указатель на значение типа double. Указатель iPtr3 использует корректный синтаксис (допустимый, но не желательный). iPtr5, iPtr6 – два указателя для переменных типа int. iPtr3 - это указатель на значение типа int, а iPtr4 - это обычная переменная типа int.

Поскольку указатели содержат только адреса, то при присваивании указателю значения — это значение должно быть адресом. Для получения адреса переменной используется оператор адреса:

```
int value = 5;
int *ptr = &value;
```

В последней строке инициализируем ptr адресом значения переменной. В следующем примере инициализируем указатель ptr адресом значения переменной value.

```
#include <iostream>
int main(){
   int value = 5;
   int *ptr = &value;
   std::cout << &value << '\n';
   std::cout << ptr << '\n';
   return 0;}
```

Следующее не является допустимым:

```
int *ptr = 7;
```

Это связано с тем, что указатели могут содержать только адреса, а целочисленный литерал 7 не имеет адреса памяти. Если вы все же сделаете это, то компилятор сообщит вам, что он не может преобразовать целочисленное значение в целочисленный указатель. Язык C++ также не позволит вам напрямую присваивать адреса памяти указателю:

```
double *dPtr = 0x0012FF7C;
```

Стоит отметить, что оператор адреса & не возвращает адрес своего операнда в качестве литерала. Вместо этого он возвращает указатель, содержащий адрес операнда, тип которого получен из аргумента (например, адрес переменной типа int передается как адрес указателя на значение типа int):

```
#include <iostream>
#include <typeinfo>
int main(){
   int x(4);
   std::cout << typeid(&x).name();
   return 0;}
```

2.4. Разыменовывание указателей

Как только у нас есть указатель, указывающий на что-либо, мы можем его разыменовать, чтобы получить значение, на которое он указывает. Разыменованный указатель — это содержимое ячейки памяти, на которую он указывает:

```
#include <iostream>
int main(){
   int value = 5;
   std::cout << &value << std::endl;
   std::cout << value << std::endl;
   int *ptr = &value;
   std::cout << ptr << std::endl;
   std::cout << *ptr << std::endl;
   return 0;}
```

Размер указателя зависит от архитектуры, на которой скомпилиро-

ван исполняемый файл: 32-битный исполняемый файл использует 32-битные адреса памяти. Следовательно, указатель на 32-битном устройстве занимает 32 бита (4 байта). С 64-битным исполняемым файлом указатель будет занимать 64 бита (8 байт). И это вне зависимости от того, на что указывает указатель. Например, тип char занимает 1 байт, тип int занимает 4 байта.

Помимо адресов памяти, есть еще одно значение, которое указатель может хранить: значение null. Нулевое значение (или «значение null») — это специальное значение, которое означает, что указатель ни на что не указывает. Указатель, содержащий значение null, называется нулевым указателем. В языке C++ мы можем присвоить указателю нулевое значение, инициализируя его/присваивая ему литерал 0.

```
int *ptr(0);
int *ptr1;
ptr1 = 0;
```

ptr теперь – нулевой указатель, ptr1 в конце стал нулевым указателем.

Поскольку значением нулевого указателя является нуль, то это можно использовать внутри условного ветвления для проверки того, является ли указатель нулевым или нет:

```
double *ptr(0);
if (ptr)
std::cout << "ptr is pointing to a double value.";
else
std::cout << "ptr is a null pointer.";
```

2.5. Макрос NULL

В языке Си (но не в C++) есть специальный макрос препроцессора с именем NULL, который определен как значение 0. Хоть он и не является частью языка C++, его использование достаточно распространено, и должно работать в каждом компиляторе C++:

```
int *ptr(NULL);
```

Здесь мы присваиваем адрес 0 указателю ptr. NULL является макросом препроцессора и, технически, не является частью C++.

2.6. nullptr

Значение 0 не является типом указателя, и присваивание указателю значения 0 для обозначения того, что он является нулевым — немного противоречиво, вам не кажется? В редких случаях, использование 0 в качестве аргумента-литерала может привести к проблемам, так как компилятор не сможет определить, используется ли нулевой указатель или целое число 0: Для решения этой проблемы в C++11 ввели новое ключевое слово nullptr.

```
int *ptr = nullptr;
```

Указатель ptr по-прежнему остается указателем типа int, просто со значением null (0).

Язык C++ обычно неявно преобразует nullptr в соответствующий тип указателя. Таким образом, в вышеприведенном примере, nullptr неявно преобразуется в указатель типа int, а затем значение nullptr присваивается ptr.

2.7. Применение указателей

Указатели полезны в следующих случаях:

1) Массивы реализованы с помощью указателей. Указатели могут использоваться для итерации по массиву.

2) Они являются единственным способом динамического выделения памяти в C++. Это, безусловно, самый распространенный вариант использования указателей.

3) Они могут использоваться для передачи большого количества данных в функцию без копирования этих данных.

4) Они могут использоваться для передачи одной функции в качестве параметра другой функции.

5) Они используются для достижения полиморфизма при работе с наследованием.

6) Они могут использоваться для представления одной структуры-/класса в другой структуре/классе, формируя, таким образом, целые цепочки.

3. Указатели и массивы

Рассмотрим следующий массив:

```
int array[4] = { 5, 8, 6, 4 };
```

Определяем фиксированный массив из 4 целых чисел Для нас это массив из 4 целых чисел, но для компилятора array является переменной типа int[4]. Мы знаем что array[0] = 5, array[1] = 8, array[2] = 6 и array[3] = 4. Но какое значение имеет сам array? Переменная array содержит адрес первого элемента массива, как если бы это был указатель!

В следующем примере возвращается всегда адрес первого элемента массива!

```
#include <iostream>
int main(){
   int array[4] = { 5, 8, 6, 4 };
   std::cout << "The array has address: " << array << '\n';
   std::cout << "Element 0 has address: " << &array << '\n';
   std::cout << "Element 0 has address: " << &array[0] << '\n';
   return 0;}
```

Распространенная ошибка думать, что переменная array и указатель на array являются одним и тем же объектом. Это не так. Хотя оба указывают на первый элемент массива, информация о типе данных у них разная. В вышеприведенном примере типом переменной array является int[4], тогда как типом указателя на массив является int *.

Разыменование массива (переменной array) приведет к возврату первого элемента массива (элемента под номером 0) и выведется 5!

```
int array[4] = { 5, 8, 6, 4 };
std::cout << *array;
```

Мы не разыменовываем фактический массив. Массив (типа int[4]) неявно конвертируется в указатель (типа int *), и мы разыменовываем указатель, который указывает на значение первого элемента массива.

Есть случаи, когда разница между фиксированными массивами и указателями имеет значение. Основное различие возникает при использовании оператора sizeof. При использовании в фиксированном массиве, оператор sizeof возвращает размер всего массива (длина массива * размер элемента). При использовании с указателем, оператор sizeof возвращает размер адреса памяти (в байтах).

```
#include <iostream>
int main(){
   int array[4] = { 5, 8, 6, 4 };
   char arr[3]={'a','b','c'};
   std::cout << sizeof(array) << '\n';
   std::cout << sizeof(arr) << '\n';
   std::cout << sizeof(ptr) << '\n';
   return 0;}
```

Фиксированный массив знает свою длину, а указатель на массив — нет.

3.1. Адресная арифметика

Язык C++ позволяет выполнять целочисленные операции сложения/вычитания с указателями. Если ptr указывает на целое число, то ptr + 1 является адресом следующего целочисленного значения в памяти после ptr. ptr - 1 — это адрес предыдущего целочисленного значения (перед ptr). Обратите внимание, ptr + 1 не возвращает следующий любой адрес памяти, который находится сразу после ptr, но он возвращает адрес памяти следующего объекта, тип которого совпадает с типом значения, на которое указывает ptr. Если ptr указывает на адрес памяти целочисленного значения (размер которого 4 байта), то ptr + 3 будет возвращать адрес памяти третьего целочисленного значения после ptr.

Если ptr указывает на адрес памяти значения типа char, то ptr + 3 будет возвращать адрес памяти третьего значения типа char после ptr. При вычислении результата выражения адресной арифметики (или «арифметики с указателями») компилятор всегда умножает целочисленный операнд на размер объекта, на который указывает указатель.

```
int value = 8;
int *ptr = &value;
std::cout << ptr << '\n';
std::cout << ptr+1 << '\n';
std::cout << ptr+2 << '\n';
std::cout << ptr+3 << '\n';
std::cout << *(ptr+3) << '\n';
std::cout << *(ptr+2) << '\n';
```

Тем не менее изменять работу сторонних программ, обращаясь к выделенной для них памяти мы не можем. Здесь будут действовать различные механизмы защиты оперативной памяти. Используя оператор адре-

са &, мы можем легко определить, что элементы массива расположены в памяти последовательно. То есть, элементы 0, 1, 2 и т.д. размещены рядом (друг за другом).

Обратите внимание, каждый из этих адресов по отдельности занимает 4 байта, как и размер типа int:

```
#include <iostream>
int main(){
   int array[] = { 7, 8, 2, 4, 5 };
   std::cout << "Element 0 is at address: " << &array[0] <<
       '\n';
   std::cout << "Element 1 is at address: " << &array[1] <<
       '\n';
   std::cout << "Element 2 is at address: " << &array[2] <<
       '\n';
   std::cout << "Element 3 is at address: " << &array[3] <<
       '\n';
   return 0;}
```

Когда компилятор видит оператор индекса [], он, на самом деле, конвертирует его в указатель с операцией сложения и разыменования! То есть, array[n] — это то же самое, что и *(array + n), где n является целочисленным значением. Оператор индекса [] используется в целях удобства, чтобы не нужно было всегда помнить о скобках.

Мы можем использовать указатели и адресную арифметику для выполнения итераций по массиву. Хотя обычно это не делается (использование оператора индекса, как правило, читабельнее и менее подвержено ошибкам).

```
#include <iostream>
int main(){
   int arrayLength = 9;
   char name[arrayLength] = ``Kazan'';
   int numVowels(0);
   for (char *ptr = name; ptr < name + arrayLength; ++ptr){
      switch (*ptr)
      {
         case 'A':
         case 'a':
         ++numVowels;  }}
   std::cout << name << " has " << numVowels << " search
       letters.\n";
   return 0;}
```

Как это работает? Программа использует указатель для прогона каждого элемента массива поочередно. Присвоив name для ptr, сам ptr стал указывать на первый элемент массива. Каждый элемент разыменовывается с помощью выражения switch, и, если текущий элемент массива является гласной буквой, то numVowels увеличивается. Для перемещения указателя на следующий символ (элемент) массива в цикле for используется оператор ++. Работа цикла завершится, когда все символы будут проверены.

Какое условие нужно было поставить в цикле, чтобы сократить обход строки?

```
for (char *ptr = name; (ptr < name +
    arrayLength)&&(*ptr!='\0'); ++ptr)
```

Да, действительно, цикл должен выглядеть именно так.

4. Операторы new и delete

Операции new и delete в C++ нужны для создания и удаления динамических объектов.

Основная особенность динамических объектов в том, что временем их жизни нужно управлять вручную. Противоположность им с этой точки зрения составляют автоматические объекты, временем жизни которых управляет компилятор.

Для управления динамическими объектами применяются операторы *new* и *delete*.

Оператор new выделяет место в динамической памяти для объекта и возвращает указатель на этот объект.

Оператор delete получает указатель на динамический объект и удаляет его из памяти.

Пример использования динамических переменных.

```
#include <iostream>
using namespace std;

int main()
{
   int *a = new int;
   int *b = new int(5);

   *a = 10;
   *b = *a + *b;
```

```
    cout << "b is " << *b << endl;

    delete b;
    delete a;

    return 0;
}
```

int *a = new int; // Объявление указателя для переменной типа int
int *b = new int(5); // Инициализация указателя

Динамический массив — это массив с элементами, выделенными в динамической памяти. Он необходим в случае, если размер неизвестен на этапе компиляции, или если размер достаточно большой, и мы не хотим выделять массив на стеке, размер которого обычно сильно ограничен.

Для выделения памяти под динамический массив также используется оператор new, после которого в квадратных скобках указывается, сколько массив будет содержать объектов:

Два способа создания динамического массива:

```
  int *numbers {new int[4]};
// int *numbers = new int[4];
```

В данном случае определяется массив из четырех элементов типа int, но каждый из них имеет неопределенное значение. Однако мы также можем инициализировать массив значениями:

```
int *numbers1 {new int[4]{}};
int *numbers2 {new int[4]{ 1, 2, 3, 4 }};
int *numbers3 {new int[4]{ 1, 2 }};
// int *numbers1 = new int[4]{};
// int *numbers1 = new int[4]();
// int *numbers2 = new int[4]{ 1, 2, 3, 4 };
// int *numbers3 = new int[4]{ 1, 2 };
```

При инициализации массива конкретными значениями следует учитывать, что если значений в фигурных скобках больше чем длина массива, то оператор new потерпит неудачу и не сможет создать массив. Если переданных значений, наоборот, меньше, то элементы, для которых не предоставлены значения, инициализируются значением по умолчанию.

Стоит отметить, что в стандарт C++20 добавлена возможность выведения размера массива, поэтому, если применяется стандарт C++20, то можно не указывать длину массива:

```
int *numbers {new int[]{ 1, 2, 3, 4 }};
```

После создания динамического массива мы сможем с ним работать по полученному указателю, получать и изменять его элементы:

```
int *numbers {new int[4]{ 1, 2, 3, 4 }};
std::cout << numbers[0] << std::endl;   std::cout << numbers[1]
    << std::endl;   std::cout << *numbers << std::endl;
    std::cout << *(numbers+1) << std::endl;
```

Причем для доступа к элементам динамического массива можно использовать как синтаксис массивов (numbers[0]), так и операцию разыменования (*numbers)

Соответственно для перебора такого массива можно использовать различные способы:

```
unsigned n{ 5 };
int* p{ new int[n] { 1, 2, 3, 4, 5 } };
for (unsigned i{}; i < n; i++)
{
  std::cout << p[i] << "\t";
}
std::cout << std::endl;

for (unsigned i{}; i < n; i++)
{
  std::cout << *(p+i)<< "\t";
}
std::cout << std::endl;

for (int* q{ p }; q != p + n; q++)
{
  std::cout << *q << "\t";
}
std::cout << std::endl;
```

Обратите внимание, что для задания размера динамического массива мы можем применять обычную переменную, а не константу, как в случае со стандартными массивами.

Для удаления динамического массива и освобождения его памяти применяется специальная форма оператора delete: delete [] указатель на динамический массив.

```
#include <iostream>

int main()
{
  unsigned n{ 5 };
  int* p{ new int[n] { 1, 2, 3, 4, 5 } };


  for (unsigned i{}; i < n; i++)
  {
     std::cout << p[i] << "\t";
  }
  std::cout << std::endl;

  delete [] p;
}
```

Чтобы после освобождения памяти указатель не хранил старый адрес, также рекомендуется обнулить его:

```
delete [] p;
p = nullptr;
```

Список литературы

[1] Страуступ Б. Программирование: принципы и практика использования C++. / Б. Страуступ. – М.: Вильямс, 2011. – 1328 с.

[2] Прата С. Язык программирования C++. Лекции и упражнения. / С. Прата. – М.: Диалектика-Вильямс, 2018. – 928 с.

[3] Павловская Т.А. C/C++. Программирование на языке высокого уровня. Учебник для вузов / Т.А. Павловская. – М.: Питер, 2010. – 461 с.

[4] Шилдт Г. C++. Базовый курс. / Г. Шилдт. – М.: Диалектика-Вильямс, 2018. – 624 с.

[5] Мейерс С. Эффективный и современный C++: 42 рекомендации по использованию C++11 и C++14. / С. Мейерс. – М.: Вильямс, 2019. – 304 с.

Методическое пособие

Долгов Дмитрий Александрович

Основы программирования. Работа с памятью в языке программирования C++