

Ravesli [Ravesli](#)


- [Уроки по C++](#)
- [OpenGL](#)
- [SFML](#)
- [Qt5](#)
- [RegExр](#)
- [Ассемблер](#)
- [Купить .PDF](#)

Урок №80. Указатели

 [Юрий](#) |

- [Уроки C++](#)

|

 Обновл. 15 Сен 2020 |

 69143

[↑](#)  18

На [уроке №10](#) мы узнали, что переменная — это название кусочка памяти, который содержит значение.

Оглавление:

1. [Оператор адреса &](#)
2. [Оператор разыменования *](#)
3. [Указатели](#)
4. [Присваивание значений указателю](#)
5. [Оператор адреса возвращает указатель](#)
6. [Разыменование указателей](#)
7. [Разыменование некорректных указателей](#)
8. [Размер указателей](#)
9. [В чём польза указателей?](#)
10. [Заключение](#)
11. [Тест](#)

Оператор адреса &

При выполнении инициализации переменной, ей автоматически присваивается свободный адрес памяти, и, любое значение, которое мы присваиваем переменной, сохраняется по этому адресу в памяти. Например:

```
1 | int b;
```

При выполнении этого стейтмента процессором, выделяется часть оперативной памяти. В качестве примера предположим, что переменной `b` присваивается ячейка памяти под номером 150. Всякий раз, когда программа встречает переменную `b` в выражении или в стейтменте, она понимает, что для того, чтобы получить значение — ей нужно заглянуть в ячейку памяти под номером 150.

Хорошая новость: 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. Однако этот подход имеет некоторые ограничения, которые мы обсудим на этом и следующих уроках.

Оператор адреса `&` позволяет узнать, какой адрес памяти присвоен определенной переменной. Всё довольно просто:

```
1 #include <iostream>
2
3 int main()
4 {
5     int a = 7;
6     std::cout << a << '\n'; // выводим значение переменной a
7     std::cout << &a << '\n'; // выводим адрес памяти переменной a
8
9     return 0;
10 }
```

Результат на моем компьютере:

```
7
0046FCF0
```

Примечание: Хотя оператор адреса выглядит так же, как [оператор побитового И](#), отличить их можно по тому, что оператор адреса является [унарным оператором](#), а оператор побитового И — бинарным оператором.

Оператор разыменования `*`

Оператор разыменования `*` позволяет получить значение по указанному адресу:

```
1 #include <iostream>
2
3 int main()
4 {
5     int a = 7;
6     std::cout << a << '\n'; // выводим значение переменной a
7     std::cout << &a << '\n'; // выводим адрес переменной a
8     std::cout << *a << '\n'; /// выводим значение ячейки памяти переменной a
9
10    return 0;
11 }
```

```
| }
```

Результат на моем компьютере:

```
7
0046FCF0
7
```

Примечание: Хотя оператор разыменования выглядит так же, как и оператор умножения, отличить их можно по тому, что оператор разыменования — унарный, а оператор умножения — бинарный.

Указатели

Теперь, когда мы уже знаем об операторах адреса и разыменования, мы можем поговорить об указателях.

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:

```
1 int *iPtr; // указатель на значение типа int
2 double *dPtr; // указатель на значение типа double
3
4 int* iPtr3; // корректный синтаксис (допустимый, но не желательный)
5 int * iPtr4; // корректный синтаксис (не делайте так)
6
7 int *iPtr5, *iPtr6; // объявляем два указателя для переменных типа int
```

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.

Однако, при объявлении нескольких указателей, звёздочка должна находиться возле каждого идентификатора. Это легко забыть, если вы привыкли указывать звёздочку возле типа данных, а не возле имени переменной. Например:

```
1 int* iPtr3, iPtr4; // iPtr3 - это указатель на значение типа int, а iPtr4 - это обычная
```

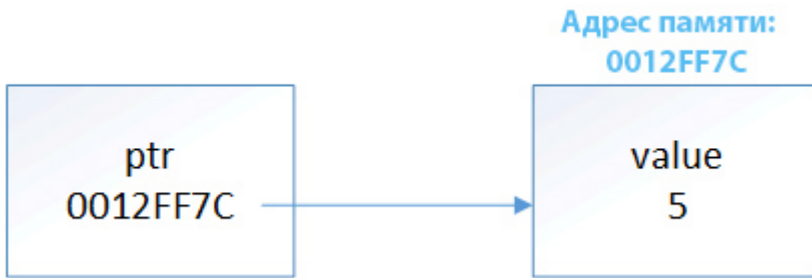
По этой причине, при объявлении указателя, рекомендуется указывать звёздочку возле имени переменной. Как и обычные переменные, указатели не инициализируются при объявлении. Содержимым неинициализированного указателя является обычный мусор.

Присваивание значений указателю

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

```
1 int value = 5;
2 int *ptr = &value; // инициализируем ptr адресом значения переменной
```

Приведенное выше можно проиллюстрировать следующим образом:



Вот почему указатели имеют такое имя: `ptr` содержит адрес значения переменной `value`, и, можно сказать, `ptr` *указывает* на это значение.

Еще очень часто можно увидеть следующее:

```

1  #include <iostream>
2
3  int main()
4  {
5      int value = 5;
6      int *ptr = &value; // инициализируем ptr адресом значения переменной
7
8      std::cout << &value << '\n'; // выводим адрес значения переменной value
9      std::cout << ptr << '\n'; // выводим адрес, который хранит ptr
10
11     return 0;
12 }
  
```

Результат на моем компьютере:

```

003AFCD4
003AFCD4
  
```

Тип указателя должен соответствовать типу переменной, на которую он указывает:

```

1  int iValue = 7;
2  double dValue = 9.0;
3
4  int *iPtr = &iValue; // ок
5  double *dPtr = &dValue; // ок
6  iPtr = &dValue; // неправильно: указатель типа int не может указывать на адрес переменной double
7  dPtr = &iValue; // неправильно: указатель типа double не может указывать на адрес переменной int
  
```

Следующее не является допустимым:

```

1  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:

```
1 | double *dPtr = 0x0012FF7C; // не ок: рассматривается как присваивание целочисленного ли
```

Оператор адреса возвращает указатель

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`):

```
1 | #include <iostream>
2 | #include <typeinfo>
3 |
4 | int main()
5 | {
6 |     int x(4);
7 |     std::cout << typeid(&x).name();
8 |
9 |     return 0;
10 | }
```

Результат выполнения программы:

`int *`

Разыменование указателей

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

```
1 | #include <iostream>
2 |
3 | int main()
4 | {
5 |     int value = 5;
6 |     std::cout << &value << std::endl; // выводим адрес value
7 |     std::cout << value << std::endl; // выводим содержимое value
8 |
9 |     int *ptr = &value; // ptr указывает на value
10 |    std::cout << ptr << std::endl; // выводим адрес, который хранится в ptr, т.е. &value
11 |    std::cout << *ptr << std::endl; // разыменовываем ptr (получаем значение на которое
12 |
13 |    return 0;
14 | }
```

| }

Результат:

```
0034FD90
5
0034FD90
5
```

Вот почему указатели должны иметь тип данных. Без типа указатель не знал бы, как интерпретировать содержимое, на которое он указывает (при разыменовании). Также, поэтому и должны совпадать тип указателя с типом переменной. Если они не совпадают, то указатель при разыменовании может неправильно интерпретировать биты (например, вместо типа `double` использовать тип `int`).

Одному указателю можно присваивать разные значения:

```
1 int value1 = 5;
2 int value2 = 7;
3
4 int *ptr;
5
6 ptr = &value1; // ptr указывает на value1
7 std::cout << *ptr; // выведется 5
8
9 ptr = &value2; // ptr теперь указывает на value2
10 std::cout << *ptr; // выведется 7
```

Когда адрес значения переменной присвоен указателю, то выполняется следующее:

- `ptr` — это то же самое, что и `&value`;
- `*ptr` обрабатывается так же, как и `value`.

Поскольку `*ptr` обрабатывается так же, как и `value`, то мы можем присваивать ему значения так, как если бы это была обычная переменная. Например:

```
1 int value = 5;
2 int *ptr = &value; // ptr указывает на value
3
4 *ptr = 7; // *ptr - это то же самое, что и value, которому мы присвоили значение 7
5 std::cout << value; // выведется 7
```

Разыменование некорректных указателей

Указатели в языке C++ по своей природе являются небезопасными, а их неправильное использование — один из лучших способов получить сбой программы.

При разыменовании указателя, программа пытается перейти в ячейку памяти, которая хранится в указателе и извлечь содержимое этой ячейки. По соображениям безопасности современные операционные системы (ОС) запускают программы в песочнице для предотвращения их неправильного

взаимодействия с другими программами и для защиты стабильности самой операционной системы. Если программа попытается получить доступ к ячейке памяти, не выделенной для нее операционной системой, то ОС сразу завершит выполнение этой программы.

Следующая программа хорошо иллюстрирует вышесказанное. При запуске вы получите сбой (попробуйте, ничего страшного с вашим компьютером не произойдет):

```
1  #include <iostream>
2
3  void foo(int *&p)
4  {
5  }
6
7  int main()
8  {
9      int *p; // создаем неинициализированный указатель (содержимым которого является мусор)
10     foo(p); // вводим компилятор в заблуждение, будто бы собираемся присвоить указателю значение
11
12     std::cout << *p; // разыменовываем указатель с мусором
13
14     return 0;
15 }
```

Размер указателей

Размер указателя зависит от архитектуры, на которой скомпилирован 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:

```
1  char *chPtr; // тип char занимает 1 байт
2  int *iPtr; // тип int занимает 4 байта
3
4  struct Something
5  {
6      int nX, nY, nZ;
7  };
8
9  Something *somethingPtr;
10
11 std::cout << sizeof(chPtr) << '\n'; // выведется 4
12 std::cout << sizeof(iPtr) << '\n'; // выведется 4
13 std::cout << sizeof(somethingPtr) << '\n'; // выведется 4
```

Как вы можете видеть, размер указателя всегда один и тот же. Это связано с тем, что указатель — это всего лишь адрес памяти, а количество бит, необходимое для доступа к адресу памяти на определенном

устройстве, — всегда постоянное.

В чём польза указателей?

Сейчас вы можете подумать, что указатели являются непрактичными и вообще ненужными. Зачем использовать указатель, если мы можем использовать исходную переменную?

Однако, оказывается, указатели полезны в следующих случаях:

- ➔ **Случай №1:** Массивы реализованы с помощью указателей. Указатели могут использоваться для итерации по массиву.
- ➔ **Случай №2:** Они являются единственным способом динамического выделения памяти в C++. Это, безусловно, самый распространенный вариант использования указателей.
- ➔ **Случай №3:** Они могут использоваться для передачи большого количества данных в функцию без копирования этих данных.
- ➔ **Случай №4:** Они могут использоваться для передачи одной функции в качестве параметра другой функции.
- ➔ **Случай №5:** Они используются для достижения полиморфизма при работе с наследованием.
- ➔ **Случай №6:** Они могут использоваться для представления одной структуры/класса в другой структуре/классе, формируя, таким образом, целые цепочки.

Указатели применяются во многих случаях. Не волнуйтесь, если вы многого не понимаете из вышесказанного. Теперь, когда мы разобрались с указателями на базовом уровне, мы можем начать углубляться в отдельные случаи, в которых они полезны, что мы и сделаем на последующих уроках.

Заключение

Указатели — это переменные, которые содержат адреса памяти. Их можно разыменовывать с помощью оператора разыменовывания * для извлечения значений, хранимых по адресу памяти. Разыменование указателя, значением которого является мусор, приведет к сбою в вашей программе.

Совет: При объявлении указателя указывайте звёздочку возле имени переменной.

Тест

Задание №1

Какие значения мы получим в результате выполнения следующей программы (предположим, что это 32-битное устройство, и тип short занимает 2 байта):

```
1 short value = 7; // &value = 0012FF60
2 short otherValue = 3; // &otherValue = 0012FF54
3
4 short *ptr = &value;
```



```

5
6 std::cout << &value << '\n';
7 std::cout << value << '\n';
8 std::cout << ptr << '\n';
9 std::cout << *ptr << '\n';
10 std::cout << '\n';
11
12 *ptr = 9;
13
14 std::cout << &value << '\n';
15 std::cout << value << '\n';
16 std::cout << ptr << '\n';
17 std::cout << *ptr << '\n';
18 std::cout << '\n';
19
20 ptr = &otherValue;
21
22 std::cout << &otherValue << '\n';
23 std::cout << otherValue << '\n';
24 std::cout << ptr << '\n';
25 std::cout << *ptr << '\n';
26 std::cout << '\n';
27
28 std::cout << sizeof(ptr) << '\n';
29 std::cout << sizeof(*ptr) << '\n';

```

Ответ №1

Значения:

0012FF60

7

0012FF60

7

0012FF60

9

0012FF60

9

0012FF54

3

0012FF54

3

4

2

Краткое объяснение по поводу последней пары: 4 и 2. 32-битное устройство означает, что размер указателя составляет 32 бита, но [оператор sizeof](#) всегда выводит размер в байтах: 32 бита = 4 байта.

Таким образом, `sizeof(ptr)` равен 4. Поскольку `ptr` является указателем на значение типа `short`, то `*ptr` является типа `short`. Размер `short` в этом примере составляет 2 байта. Таким образом, `sizeof(*ptr)` равен 2.

Задание №2

Что не так со следующим фрагментом кода:

```
1 int value = 45;  
2 int *ptr = &value; // объявляем указатель и инициализируем его адресом переменной value  
3 *ptr = &value; // присваиваем адрес value для ptr
```

Ответ №2

Последняя строка не скомпилируется. Рассмотрим эту программу детально.

В первой строке находится стандартное определение переменной вместе с инициализируемым значением. Здесь ничего особенного.

Во второй строке мы определяем новый указатель с именем `ptr` и присваиваем ему адрес переменной `value`. Помним, что в этом контексте звёздочка является частью синтаксиса объявления указателя, а не оператором разыменования. Так что и в этой строке всё нормально.

В третьей строке звёздочка уже является оператором разыменования, и используется для вытаскивания значения, на которое указывает указатель. Таким образом, эта строка говорит: «Вытаскиваем значение, на которое указывает `ptr` (целочисленное значение), и переписываем его на адрес этого же значения». А это уже какая-то чепуха — вы не можете присвоить адрес целочисленному значению!

Третья строка должна быть:

```
1 ptr = &value;
```

В вышеприведенной строке мы корректно присваиваем указателю адрес значения переменной.

Оценить статью:

★★★★★ (343 оценок, среднее: 4,89 из 5)



[← Введение в класс `std::string_view` в C++](#)

[Урок №81. Нулевые указатели](#)



Комментариев: 18



1. Евгений:

[13 февраля 2020 в 19:35](#)

Спасибо, информация хорошо разжевана. Впервые разобрался с указателями.
Для любителей поскрипеть мозгами и залезть глубже:

```
1 int *pa = new int{9};  
2 std::cout << *pa;
```

По сути, нам не нужна переменная, в которой будет храниться значение, мы лишь храним адрес на это значение.

Но в этом случае переменная не удалится автоматически, а останется в памяти приложения.

[Ответить](#)



2. Malkin:

[30 сентября 2019 в 17:38](#)

Чуть добавлю, хотя автор и так всё это знает. Вся эта фигня — указатели, ссылки — так или иначе связана с косвенной адресацией. Зачем она нужна? Затем, чтобы увеличить производительность. Так как при косвенной адресации, у нас не меняется сама команда: [код операции + указатель на адрес]. А указатель — например, FSR, как в PIC контроллерах. Адрес то FSR — 04h не меняется, а меняем мы лишь его содержимое, загоняя туда адреса операндов из портов МК. В итоге, удлиняется время выполнения программы, но производительность увеличивается, так как не надо каждый раз формировать новую команду с новым адресом нового операнда, как при прямой адресации. p.s. мож что соврал или недопонял)

[Ответить](#)



3. Алексей:

[30 июля 2019 в 16:39](#)

Указатели, интересная тема на самом деле.
Сейчас пока в школе, думаю после 100 урока будет в универе.
После 200 рабочие проекты.

[Ответить](#)



4. Анастасия:

[23 июня 2019 в 20:36](#)

1) Кто-нибудь, напомните, пожалуйста, где встречалось пояснение вот этого:

```
1 #include <typeinfo>  
2 std::cout << typeid(&x).name();
```

А то я уже или забыла или пропустила, похоже.

2) В примере, где иллюстрируется сбой программы при обращении к неинициализированному указателю, поясните, пожалуйста, почему так сложно записан параметр для функции `foo: void foo(int *&p)`. Что это значит? Если мы хотим показать, что её параметр — указатель, то почему нельзя было написать `void foo(int *p)`?

[Ответить](#)



1. *Alex_1988:*

[30 января 2020 в 19:56](#)

1) Урок №55. Неявное преобразование типов данных

2) Мне кажется новичкам правильнее было бы показать так:

```
1 int main()
2 {
3     int *p{}; /*указателю нельзя присваивать литерал, но трюк с пустыми унифо
4
5     std::cout << *p; /*если вывести значение по адресу 0 которое явно не выде
6
7     return 0;
8 }
```

[Ответить](#)



2. *Анатолий:*

[19 июля 2020 в 10:54](#)

Нет, не пропустили. Означает: вывести тип переменной

[Ответить](#)



5. *Алексей:*

[31 мая 2019 в 00:48](#)

Немного не понял по поводу как правильно `int* ptr` или `int *ptr`. Проблема в том, что я написал в своем коде второй вариант, поставил точку с запятой, и VS автоматически исправил на первый вариант. Но почему?

[Ответить](#)



1. *Анастасия:*

[23 июня 2019 в 20:26](#)

Правильны оба варианта, пока Вы не начнёте объявлять сразу несколько указателей. В этом случае только `*ptr`.

[Ответить](#)



6. *Татьяна:*

[17 апреля 2019 в 14:58](#)

Здравствуйте !

В тексте написано: "Как и обычные переменные, указатели не инициализируются при объявлении."

, но, если я правильно понимаю,

```
1 | int *ptr = &value; // инициализируем ptr адресом значения переменной
```

это и есть объявление указателя и его инициализация.

[Ответить](#)



1. *Анастасия:*

[23 июня 2019 в 20:24](#)

Вы правильно понимаете. В тексте написано "Как и обычные переменные, указатели не инициализируются при объявлении.". Это означает, что если только объявить указатель и не инициализировать его, то там будет что попало, поэтому и надо их сразу инициализировать. При объявлении.

[Ответить](#)



7. *Рустам:*

[23 января 2019 в 20:35](#)

Лол! Попробовал я значит вывести указатель с мусором, в итоге упал visual studio)

[Ответить](#)



1. *somebox:*

[30 мая 2019 в 19:48](#)

А в Xcode мусорные значения всегда забиваются нулями.

[Ответить](#)



8. *Владимир:*

[30 декабря 2018 в 21:10](#)

Юрий спасибо! C++ начал осваивать недавно и с указателями — ступор. Теперь появился свет к конце туннеля.

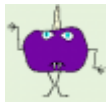
[Ответить](#)



1. *Юрий:*
[30 декабря 2018 в 21:46](#)

Пожалуйста 😊

[Ответить](#)



9. *Vado:*
[17 июля 2018 в 10:16](#)

Очень классные уроки! Спасибо за труд! Все по делу, ничего лишнего!

[Ответить](#)



10. *Виталий:*
[14 апреля 2018 в 12:12](#)

Очень странный термин "разыменование". Никак не клеится со смыслом. Это общепринятая терминология или вольный перевод автора?

[Ответить](#)



1. *Юрий:*
[14 апреля 2018 в 19:12](#)

Как вы определяете общепринятую терминологию? Разве у нас есть орган, который однозначно указывает перевод слов? Слово "разыменование" используется во многих ресурсах о программировании, но ручаться за все источники и что это общепринятая терминология — я не могу.

В этих уроках используется термин "разыменование".

[Ответить](#)



2. *Анастасия:*
[23 июня 2019 в 20:18](#)

Это не вольный перевод автора. В других книгах этот оператор так же называют.

[Ответить](#)

Добавить комментарий

Ваш E-mail не будет опубликован. Обязательные поля помечены *

Имя *






Email *

Комментарий

- ☐ Сохранить моё Имя и E-mail. Видеть комментарии, отправленные на модерацию
- ☐ Получать уведомления о новых комментариях по электронной почте. Вы можете [подписаться](#) без комментирования.

[TELEGRAM](#)  [КАНАЛ](#)
[ПАБЛИК](#) 

ТОП СТАТЬИ

-  [Словарь программиста. Сленг, который должен знать каждый кодер](#)
-  [Урок №1. Введение в программирование](#)
-  [70+ бесплатных ресурсов для изучения программирования](#)
-  [Урок №1: Введение в создание игры «SameGame» на C++/MFC](#)
-  [Урок №4. Установка IDE \(Интегрированной Среды Разработки\)](#)

- [Ravesli](#)
- - [О проекте/Контакты](#) -
- - [Пользовательское Соглашение](#) -
- - [Все статьи](#) -
- Copyright © 2015 - 2020