

Ravesli [Ravesli](#)


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


Урок №85. Динамическое выделение памяти

 [Юрий](#) |

- [Уроки C++](#)

|

 Обновл. 25 Сен 2020 |

 70269

[↑](#)  22

Язык C++ поддерживает три основных типа **выделения** (или «*распределения*») **памяти**, с двумя из которых, мы уже знакомы:

- ➔ **Статическое выделение памяти** выполняется для [статических](#) и [глобальных](#) переменных. Память выделяется один раз (при запуске программы) и сохраняется на протяжении работы всей программы.
- ➔ **Автоматическое выделение памяти** выполняется для [параметров функции](#) и [локальных переменных](#). Память выделяется при входе в блок, в котором находятся эти переменные, и удаляется при выходе из него.
- ➔ **Динамическое выделение памяти** является темой этого урока.

Оглавление:

1. [Динамическое выделение переменных](#)
2. [Как работает динамическое выделение памяти?](#)
3. [Освобождение памяти](#)
4. [Висячие указатели](#)
5. [Оператор new](#)
6. [Нулевые указатели и динамическое выделение памяти](#)
7. [Утечка памяти](#)
8. [Заключение](#)

Динамическое выделение переменных

Как статическое, так и автоматическое распределение памяти имеют два общих свойства:

- Размер переменной/**массива** должен быть известен во время компиляции.
- Выделение и освобождение памяти происходит автоматически (когда переменная создается/уничтожается).

В большинстве случаев с этим всё ОК. Однако, когда дело доходит до работы с пользовательским вводом, то эти ограничения могут привести к проблемам.

Например, при использовании **строки** для хранения имени пользователя, мы не знаем наперед насколько длинным оно будет, пока пользователь его не введет. Или нам нужно создать игру с непостоянным количеством монстров (во время игры одни монстры умирают, другие появляются, пытаюсь, таким образом, убить игрока).

Если нам нужно объявить размер всех переменных во время компиляции, то самое лучшее, что мы можем сделать — это попытаться угадать их максимальный размер, надеясь, что этого будет достаточно:

```
1 char name[30]; // будем надеяться, что пользователь введет имя длиной менее 30 символов
2 Monster monster[30]; // 30 монстров максимум
3 Polygon rendering[40000]; // этому 3D-рендерингу лучше состоять из менее чем 40000 полигонов
```

Это плохое решение, по крайней мере, по трем причинам:

Во-первых, теряется память, если переменные фактически не используются или используются, но не все. Например, если мы выделим 30 символов для каждого имени, но имена в среднем будут занимать по 15 символов, то потребление памяти получится в два раза больше, чем нам нужно на самом деле. Или рассмотрим массив `rendering`: если он использует только 20 000 полигонов, то память для других 20 000 полигонов фактически тратится впустую (т.е. не используется)!

Во-вторых, память для большинства обычных переменных (включая фиксированные массивы) выделяется из специального резервуара памяти — **стека**. Объем памяти стека в программе, как правило, невелик: в Visual Studio он по умолчанию равен 1МБ. Если вы превысите это значение, то произойдет *переполнение стека*, и операционная система автоматически завершит выполнение вашей программы.

В Visual Studio это можно проверить, запустив следующий фрагмент кода:

```
1 int main()
2 {
3     int array[1000000000]; // выделяем 1 миллиард целочисленных значений
4 }
```

Лимит в 1МБ памяти может быть проблематичным для многих программ, особенно где используется графика.

В-третьих, и самое главное, это может привести к искусственным ограничениям и/или переполнению массива. Что произойдет, если пользователь попытается прочесть 500 записей с диска, но мы выделили память максимум для 400? Либо мы выведем пользователю ошибку, что максимальное количество записей — 400, либо (в худшем случае) выполнится переполнение массива и затем что-то очень нехорошее.

К счастью, эти проблемы легко устраняются с помощью динамического выделения памяти.

Динамическое выделение памяти — это способ запроса памяти из операционной системы запущенными программами по мере необходимости. Эта память не выделяется из ограниченной памяти стека программы, а выделяется из гораздо большего хранилища, управляемого операционной системой — **кучи**. На современных компьютерах размер кучи может составлять гигабайты памяти.

Для динамического выделения памяти одной переменной используется **оператор new**:

```
1 | new int; // динамически выделяем целочисленную переменную и сразу же отбрасываем резуль
```

В примере, приведенном выше, мы запрашиваем выделение памяти для целочисленной переменной из операционной системы. Оператор new возвращает **указатель**, содержащий адрес выделенной памяти.

Для доступа к выделенной памяти создается указатель:

```
1 | int *ptr = new int; // динамически выделяем целочисленную переменную и присваиваем её а
```

Затем мы можем разыменовать указатель для получения значения:

```
1 | *ptr = 8; // присваиваем значение 8 только что выделенной памяти
```

Вот один из случаев, когда указатели полезны. Без указателя с адресом на только что выделенную память у нас не было бы способа получить доступ к ней.

Как работает динамическое выделение памяти?

На вашем компьютере имеется память (возможно, большая её часть), которая доступна для использования программами. При запуске программы ваша операционная система загружает эту программу в некоторую часть этой памяти. И эта память, используемая вашей программой, разделена на несколько частей, каждая из которых выполняет определенную задачу. Одна часть содержит ваш код, другая используется для выполнения обычных операций (отслеживание вызываемых функций, создание и уничтожение глобальных и локальных переменных и т.д.). Мы поговорим об этом чуть позже. Тем не менее, большая часть доступной памяти компьютера просто находится в ожидании запросов на выделение от программ.

Когда вы динамически выделяете память, то вы просите операционную систему зарезервировать часть этой памяти для использования вашей программой. Если ОС может выполнить этот запрос, то возвращается адрес этой памяти обратно в вашу программу. С этого момента и в дальнейшем ваша программа сможет использовать эту память, как только пожелает. Когда вы уже выполнили с этой памятью всё, что было необходимо, то её нужно вернуть обратно в операционную систему, для распределения между другими запросами.

В отличие от статического или автоматического выделения памяти, программа самостоятельно отвечает за запрос и обратный возврат динамически выделенной памяти.

Освобождение памяти

Когда вы динамически выделяете переменную, то вы также можете её инициализировать посредством прямой инициализации или `uniform`-инициализации (в C++11):

```
1 int *ptr1 = new int (7); // используем прямую инициализацию
2 int *ptr2 = new int { 8 }; // используем uniform-инициализацию
```

Когда уже всё, что требовалось, выполнено с динамически выделенной переменной — нужно явно указать для C++ освободить эту память. Для переменных это выполняется с помощью **оператора delete**:

```
1 // Предположим, что ptr ранее уже был выделен с помощью оператора new
2 delete ptr; // возвращаем память, на которую указывал ptr, обратно в операционную систему
3 ptr = 0; // делаем ptr нулевым указателем (используйте nullptr вместо 0 в C++11)
```

Оператор `delete` на самом деле ничего не удаляет. Он просто возвращает память, которая была выделена ранее, обратно в операционную систему. Затем операционная система может переназначить эту память другому приложению (или этому же снова).

Хотя может показаться, что мы удаляем *переменную*, но это не так! Переменная-указатель по-прежнему имеет ту же область видимости, что и раньше, и ей можно присвоить новое значение, как и любой другой переменной.

Обратите внимание, удаление указателя, не указывающего на динамически выделенную память, может привести к проблемам.

Висячие указатели

Язык C++ не предоставляет никаких гарантий относительно того, что произойдет с содержимым освобожденной памяти или со значением удаляемого указателя. В большинстве случаев, память, возвращаемая операционной системе, будет содержать те же значения, которые были у нее до *освобождения*, а указатель так и останется указывать на только что освобожденную (удаленную) память.

Указатель, указывающий на освобожденную память, называется **висячим указателем**. Разыменование или удаление висячего указателя приведет к неожиданным результатам. Рассмотрим следующую программу:

```
1 #include <iostream>
2
3 int main()
4 {
5     int *ptr = new int; // динамически выделяем целочисленную переменную
6     *ptr = 8; // помещаем значение в выделенную ячейку памяти
7
8     delete ptr; // возвращаем память обратно в операционную систему, ptr теперь является висячим указателем
9
10    std::cout << *ptr; // разыменование висячего указателя приведет к неожиданным результатам
11    delete ptr; // попытка освободить память снова приведет к неожиданным результатам
12 }
```

```
13 |     return 0;  
14 | }
```

В программе, приведенной выше, значение 8, которое ранее было присвоено динамической переменной, после освобождения может и далее находиться там, а может и нет. Также возможно, что освобожденная память уже могла быть выделена другому приложению (или для собственного использования операционной системы), и попытка доступа к ней приведет к тому, что операционная система автоматически прекратит выполнение вашей программы.

Процесс освобождения памяти может также привести и к созданию *нескольких* висячих указателей. Рассмотрим следующий пример:

```
1 | #include <iostream>  
2 |  
3 | int main()  
4 | {  
5 |     int *ptr = new int; // динамически выделяем целочисленную переменную  
6 |     int *otherPtr = ptr; // otherPtr теперь указывает на ту же самую выделенную память  
7 |  
8 |     delete ptr; // возвращаем память обратно в операционную систему. ptr и otherPtr те  
9 |     ptr = 0; // ptr теперь уже nullptr  
10 |  
11 |     // Однако, otherPtr по-прежнему является висячим указателем!  
12 |  
13 |     return 0;  
14 | }
```

Есть несколько рекомендаций, которые могут здесь помочь:

- ➔ Во-первых, старайтесь избегать ситуаций, когда несколько указателей указывают на одну и ту же часть выделенной памяти. Если это невозможно, то выясните, какой указатель из всех «владеет» памятью (и отвечает за её удаление), а какие указатели просто получают доступ к ней.
- ➔ Во-вторых, когда вы удаляете указатель, и, если он не выходит из области видимости сразу же после удаления, то его нужно сделать нулевым, т.е. присвоить значение 0 (или `nullptr` в C++11). Под «выходом из области видимости сразу же после удаления» имеется в виду, что вы удаляете указатель в самом конце блока, в котором он объявлен.

Правило: Присваивайте удаленным указателям значение 0 (или `nullptr` в C++11), если они не выходят из области видимости сразу же после удаления.

Оператор new

При запросе памяти из операционной системы в редких случаях она может быть не выделена (т.е. её может и не быть в наличии).

По умолчанию, если оператор `new` не сработал, память не выделилась, то генерируется *исключение* `bad_alloc`. Если это исключение будет неправильно обработано (а именно так и будет, поскольку мы

еще не рассматривали исключения и их обработку), то программа просто прекратит свое выполнение (произойдет сбой) с ошибкой необработанного исключения.

Во многих случаях процесс генерации исключения оператором `new` (как и сбой программы) нежелателен, поэтому есть альтернативная форма оператора `new`, которая возвращает нулевой указатель, если память не может быть выделена. Нужно просто добавить константу `std::nothrow` между ключевым словом `new` и типом данных:

```
1 | int *value = new (std::nothrow) int; // указатель value станет нулевым, если динамическ
```

В примере, приведенном выше, если оператор `new` не возвратит указатель с динамически выделенной памятью, то возвратится нулевой указатель.

Разыменовывать его также не рекомендуется, так как это приведет к неожиданным результатам (скорее всего, к сбою в программе). Поэтому наилучшей практикой является проверка всех запросов на выделение памяти для обеспечения того, что эти запросы будут выполнены успешно и память выделится:

```
1 | int *value = new (std::nothrow) int; // запрос на выделение динамической памяти для цел
2 | if (!value) // обрабатываем случай, когда new возвращает null (т.е. память не выделяется
3 | {
4 |     // Обработка этого случая
5 |     std::cout << "Could not allocate memory";
6 | }
```

Поскольку не выделение памяти оператором `new` происходит крайне редко, то обычно программисты забывают выполнять эту проверку!

Нулевые указатели и динамическое выделение памяти

Нулевые указатели (указатели со значением `0` или `nullptr`) особенно полезны в процессе динамического выделения памяти. Их наличие как бы сообщаем нам: «Этому указателю не выделено никакой памяти». А это, в свою очередь, можно использовать для выполнения условного выделения памяти:

```
1 | // Если для ptr до сих пор не выделено памяти, то выделяем её
2 | if (!ptr)
3 |     ptr = new int;
```

Удаление нулевого указателя ни на что не влияет. Таким образом, в следующем нет необходимости:

```
1 | if (ptr)
2 |     delete ptr;
```

Вместо этого вы можете просто написать:

```
1 | delete ptr;
```

Если `ptr` не является нулевым, то динамически выделенная переменная будет удалена. Если значением указателя является ноль, то ничего не произойдет.

Утечка памяти

Динамически выделенная память не имеет области видимости, т.е. она остается выделенной до тех пор, пока не будет явно освобождена или пока ваша программа не завершит свое выполнение (и операционная система очистит все буфера памяти самостоятельно). Однако указатели, используемые для хранения динамически выделенных адресов памяти, следуют правилам области видимости обычных переменных. Это несоответствие может вызвать интересное поведение, например:

```
1 void doSomething()  
2 {  
3     int *ptr = new int;  
4 }
```

Здесь мы динамически выделяем целочисленную переменную, но никогда не освобождаем память через использование оператора `delete`. Поскольку указатели следуют всем тем же правилам, что и обычные переменные, то, когда функция завершит свое выполнение, `ptr` выйдет из области видимости. Поскольку `ptr` — это единственная переменная, хранящая адрес динамически выделенной целочисленной переменной, то, когда `ptr` уничтожится, больше не останется указателей на динамически выделенную память. Это означает, что программа «потеряет» адрес динамически выделенной памяти. И в результате эту динамически выделенную целочисленную переменную нельзя будет удалить.

Это называется **утечкой памяти**. Утечка памяти происходит, когда ваша программа теряет адрес некоторой динамически выделенной части памяти (например, переменной или массива), прежде чем вернуть её обратно в операционную систему. Когда это происходит, то программа уже не может удалить эту динамически выделенную память, поскольку она больше не знает, где она находится. Операционная система также не может использовать эту память, поскольку считается, что она по-прежнему используется вашей программой.

Утечки памяти «съедают» свободную память во время выполнения программы, уменьшая количество доступной памяти не только для этой программы, но и для других программ также. Программы с серьезными проблемами с утечкой памяти могут «съесть» всю доступную память, в результате чего ваш компьютер будет медленнее работать или даже произойдет сбой. Только после того, как выполнение вашей программы завершится, операционная система сможет очистить и *вернуть* всю память, которая «утекла».

Хотя утечка памяти может возникнуть и из-за того, что указатель выходит из области видимости, возможны и другие способы, которые могут привести к утечкам памяти. Например, если указателю, хранящему адрес динамически выделенной памяти, присвоить другое значение:

```
1 int value = 7;  
2 int *ptr = new int; // выделяем память  
3 ptr = &value; // старый адрес утерян - произойдет утечка памяти
```

Это легко решается удалением указателя перед операцией переприсваивания:

```
1 int value = 7;
```

```
2 | int *ptr = new int; // выделяем память
3 | delete ptr; // возвращаем память обратно в операционную систему
4 | ptr = &value; // переприсваиваем указателю адрес value
```

Кроме того, утечка памяти также может произойти и через двойное выделение памяти:

```
1 | int *ptr = new int;
2 | ptr = new int; // старый адрес утерян - произойдет утечка памяти
```

Адрес, возвращаемый из второго выделения памяти, перезаписывает адрес из первого выделения. Следовательно, первое динамическое выделение становится утечкой памяти!

Точно так же этого можно избежать удалением указателя перед операцией переприсваивания.

Заключение

С помощью операторов `new` и `delete` можно динамически выделять отдельные переменные в программе. Динамически выделенная память не имеет области видимости и остается выделенной до тех пор, пока не произойдет её освобождение или пока программа не завершит свое выполнение. Будьте осторожны, не разыменовывайте висячие или нулевые указатели.

На следующем уроке мы рассмотрим использование операторов `new` и `delete` для выделения и удаления динамически выделенных массивов.

Оценить статью:

★★★★★ (347 оценок, среднее: 4,91 из 5)



← [Урок №84. Символьные константы строк C-style](#)

[Урок №86. Динамические массивы](#) →



Комментариев: 22



1. Сергей:
[29 августа 2020 в 21:32](#)

"Объем памяти стека в программе, как правило, невелик: в Visual Studio он по умолчанию равен 1МБ."

Вопрос а как я могу управлять памятью стека. Или все программы должны укладываться в этот мегабайт?

[Ответить](#)1. *Владимир:*[30 августа 2020 в 12:08](#)

Вопрос, а зачем вам управлять памятью стека? Насколько я понимаю это во первых управляется автоматически, а во вторых это тема уже низкоуровневого программирования (assembler там).

Размер стека и их кол-во формируется автоматически, в зависимости от кол-ва обычных переменных, всевозможных функций в программе, статических и динамических библиотек (факторов много если коротко). Вам это может понадобится только если вы разрабатываете какие либо Операционные Системы или другие среды выполнения, ну или возможно программирование контроллеров.

Я конечно и близко не специалист, если где то сказал откровенную чушь, поправте.

[Ответить](#)2. *Дмитрий:*[26 июля 2020 в 10:00](#)

Добрый день. Пытаюсь сделать модуль C++ для питона. Модуль запускается однократно, во время его работы переполняется память. Модуль создает оператором new большое количество объектов. $\text{Sizeof(obj)} = 1056$. При создании около 60 000 объектов процесс занимает 2Г памяти и выдает прерывание bad_alloc.

Количество выделяемой на объекты памяти я контролировал в функции, которая их порождает, сделал специальный счетчик для этого. Общая использованная память $1056 * 60\,000 = 64\text{М}$, т.е. в 30 раз меньше памяти процесса. В каком режиме запускать — с отладкой или без, не влияет. В чем проблема, откуда такой расход памяти?

[Ответить](#)3. *Хабибулло:*[20 мая 2020 в 19:27](#)

Спасибо большое!
Коротко и понятно)

[Ответить](#)4. *Виталий:*[23 ноября 2019 в 06:16](#)

Объясните, пожалуйста, целесообразность создания указателя для одной переменной.

[Ответить](#)



1. Евгений:

[15 февраля 2020 в 22:02](#)

Это может быть большая переменная, например, структура, которая будет содержать несколько переменных или большие массивы.

```

1 void test_85_dyn_struct() {
2     std::cout << "# test_85_dyn_struct()\n\n";
3
4     const int size = 100'000'000;
5
6     struct Big {
7         int arr[size];
8         int brr[size];
9         int crr[size];
10        int drr[size];
11    };
12
13    // если объявить переменную так, то программа вылетит из-за нехватки стека
14    // Big a;
15    // поэтому будем выделять динамическую память для такой структуры
16    Big *parr = new Big;
17    int step = size / 10;
18    for(int i=0; i<size; ++i) {
19        (*parr).arr[i] = i;
20        (*parr).brr[i] = i + 1;
21        (*parr).crr[i] = i + 2;
22        (*parr).drr[i] = i + 3;
23        // в консоли будем выводить статистику заполнения массивов
24        if (i % step == 0) {
25            std::cout << i << " elements filled" << std::endl;
26        }
27    }
28
29    // теперь можно проверить сколько RAM занимает приложение
30    std::cout << "check RAM then type an integer: ";
31    int c;
32    std::cin >> c;
33
34    std::cout << std::endl;
35 }
```

[Ответить](#)



1. Сергей:

[29 августа 2020 в 21:34](#)

```
1 | const int size = 100'000'000;
```

что значит правая часть, я имею ввиду скобки одинарные?

[Ответить](#)



1. *Антон:*

[9 сентября 2020 в 00:36](#)

ничего особенного, с помощью одинарных кавычек можно разделять числа для лучшей читабельности



5. *Игорь:*

[2 ноября 2019 в 16:45](#)

Я так понял, что 99% всех переменных и массивов я буду использовать именно через динамическую память? И красивое и лаконичное `int value(26);` Превращается в нагромождение в пол экрана из следующего?

```
1 | int *value = new int (26);
```

И при этом ещё и следить за тем, чтобы освобождалась память при помощи `delete`?

[Ответить](#)



1. *Евгений:*

[15 февраля 2020 в 22:20](#)

Игорь, `int` это ещё самая короткая запись для типа данных, который ты будешь использовать))

[Ответить](#)



6. *Oleg:*

[12 февраля 2019 в 03:16](#)

Здравствуйте. Вопрос следующего характера. Мне необходимо написать несколько функций таким образом что бы всё взаимодействие было с одним динамическим массивом.

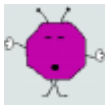
— в первой функции создаем динамический массив.

— во второй функции удаляем массив созданный первой функцией.

Подскажите, плз, как это реализовать.

проблема именно с функцией удаления массива так как не совсем понимаю как передать этот массив в функцию удаления.

[Ответить](#)

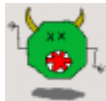


1. *Александр:*

[22 февраля 2019 в 11:57](#)

```
1 void create(int *arr, int n) {  
2     arr = new int[n];  
3 }  
4 void delArr(int *arr) {  
5     delete []arr;  
6 }
```

[Ответить](#)



7. *Тимофей:*

[29 октября 2018 в 20:38](#)

Доброго времени суток!

Вопрос у меня такой...

Как, если память возвратилась обратно в ОС, переменную можно потом использовать?

"Хотя может показаться, что мы удаляем переменную, но это не так! Переменная-указатель по-прежнему имеет ту же область видимости, что и раньше, ВОТ С ЭТОГО МЕСТА и ей можно присвоить новое значение, как и любой другой переменной."

Буду рад услышать ваш ответ!

[Ответить](#)



1. *Владимир:*

[3 декабря 2018 в 23:58](#)

В вашем вопросе уже кроется половина ответа. Как и сказано в статье, `delete` возвращает в ОС память, на которую указывает указатель, и далее ОС может делать с ней всё, что пожелает. Но `delete` НЕ УДАЛЯЕТ переменную, то есть после выполнения этого стейтмента тому же самому указателю можно повторно выделить память в любое время. Для наглядности:

```
1     int *pointer_1;  
2     pointer_1 = new int{ 5 };  
3     cout << "Adresses: " << pointer_1 << ' ' << "Value: " << *pointer_1 <<  
4  
5     delete pointer_1;  
6  
7     pointer_1 = new int{ 20 };  
8     cout << "Adresses: " << pointer_1 << ' ' << "Value: " << *pointer_1 << '  
9  
10    delete pointer_1;
```

```
11 |  
12 |     pointer_1 = new int{ 12 };  
13 |     cout << "Adresses: " << pointer_1 << ' ' << "Value: " << *pointer_1 << '  
14 |  
15 |     delete pointer_1;  
16 |     pointer_1 = nullptr;  
17 |     cout << "Adresses: " << pointer_1 << '\n';
```

Указатель после каждого оператора new будет указывать на один и тот же адрес памяти (не знаю, почему), однако содержимое этой области памяти будет разным. Если мы в 8 строке уберём { 20 }, то в консоль после адреса выведется случайное значение.

[Ответить](#)



1. Евгений:

[15 февраля 2020 в 22:23](#)

> Указатель после каждого оператора new будет указывать на один и тот же адрес памяти

Потому что вы освобождаете память и ОС еще не успевает ее отдать другому приложению. Вы сразу же занимаете эту область памяти снова.

[Ответить](#)



8. Максим:

[6 августа 2018 в 09:16](#)

Автор! Спасибо за прекраснейшую статью! Как много полезной информации!

[Ответить](#)



1. Юрий:

[6 августа 2018 в 19:37](#)

Спасибо, что читаешь 😊

[Ответить](#)



9. Владимир:

[3 августа 2018 в 23:24](#)

Здравствуйте) У меня тут небольшой вопросик общего характера по выделению памяти. Правильно ли я понимаю, что ,когда идёт компиляция программы, компилятор в каком-то из этапов проходит по программе и предоставляет для использования(но пока не заполняет) какие-то определённые адреса в памяти для переменных? А сама память выделяется, только тогда, когда я запускаю программу (в начале программы для глобальных переменных/массивов и в определенных блоках для локальных переменных/массивов). Так всё это работает??

"Как статическое, так и автоматическое распределение памяти имеют две общие черты:

Размер переменной/массива должен быть известен во время компиляции.

Выделение и освобождение памяти происходит автоматически (когда переменная создается или уничтожается)."

Ну со вторым пунктом всё понятно, а с первым не очень) То есть получается, (как я писал выше), под переменные или массивы память предоставляется ещё во время компиляции? Или тогда для чего нужно знать размер массива во время компиляции? Если да, то возникает ещё вопрос, если у меня в программе много функций в которых создаются переменные (именно создаются, а не передаются), компилятор проходит ещё и по всем функциям, резервируя место и для этих переменных? в независимости, буду я вызывать эту функцию или нет? Заранее спасибо)))

[Ответить](#)

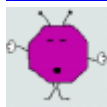
R

1. *Юрий:*

[21 августа 2018 в 21:51](#)

Привет. Как уже было сказано в уроке — память выделяется во время компиляции автоматически, она не резервируется перед выполнением программы (до того как ты её запустишь), она сразу же выделяется, как компилятор встречает переменную. Компилятор перемещается построчно и если есть вызов функции, то он переходит в эту функцию и если встречает новую переменную, то выделяет сразу же ей память. Неважно, будешь ли ты использовать переменную в дальнейшем, если она есть в коде, то память для неё выделяется.

[Ответить](#)



2. *Александр:*

[22 февраля 2019 в 12:03](#)

тут скорее путанность в терминологии... человеческий язык не может отразить весь тот вынос мозга, который происходит в плюсах 😊

на этапе компиляции происходит что-то вроде предварительного распределения памяти... можно представить, что создается некоторый условный шаблон карты памяти для конкретной функции.

А фактическое распределение/выделение памяти происходит при фактическом вызове функции на этапе работы программы.

В этом легко убедиться, создавая рекурсивные вызовы функций с автоматическими переменными — чем больше вызовов выполнит рекурсия, тем больше памяти сожрет программа. Если бы полная карта распределения памяти происходила на этапе компиляции, то объем съедаемой памяти не зависел бы от глубины рекурсии

[Ответить](#)



10. *Nanes:*

[9 октября 2017 в 15:48](#)

Когда до объектно-ориентированной части дойдем?

[Ответить](#)

1.  Юрий:

[9 октября 2017 в 16:45](#)

Этот материал будет немного позже.

[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