

Ravesli [Ravesli](#)


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


Урок №105. Стек и Куча

 [Юрий](#) |

- [Уроки C++](#)

|

 Обновл. 2 Дек 2020 |

 56259

[↑](#)  3

На этом уроке мы рассмотрим стек и кучу в языке C++.

Оглавление:

1. [Сегменты](#)
2. [Куча](#)
3. [Стек вызовов](#)
4. [Стек как структура данных](#)
5. [Сегмент стека вызовов](#)
6. [Стек вызовов на практике](#)
7. [Пример стека вызовов](#)
8. [Переполнение стека](#)

Сегменты

Память, которую используют программы, состоит из нескольких частей — **сегментов**:

- ➔ **Сегмент кода** (или «*текстовый сегмент*»), где находится скомпилированная программа. Обычно доступен только для чтения.
- ➔ **Сегмент bss** (или «*неинициализированный сегмент данных*»), где хранятся глобальные и **статические переменные**, инициализированные нулем.
- ➔ **Сегмент данных** (или «*сегмент инициализированных данных*»), где хранятся инициализированные глобальные и статические переменные.

- **Куча**, откуда выделяются динамические переменные.
- **Стек вызовов**, где хранятся параметры функции, локальные переменные и другая информация, связанная с функциями.

Куча

Сегмент кучи (или просто «*куча*») отслеживает память, используемую для динамического выделения. Мы уже немного поговорили о куче на [уроке о динамическом выделении памяти в языке C++](#).

В языке C++ при использовании оператора `new` динамическая память выделяется из сегмента кучи самой программы:

```
1 int *ptr = new int; // для ptr выделяется 4 байта из кучи
2 int *array = new int[10]; // для array выделяется 40 байт из кучи
```

Адрес выделяемой памяти передается обратно оператором `new` и затем он может быть сохранен в [указателе](#). О механизме хранения и выделения свободной памяти нам сейчас беспокоиться не за чем. Однако стоит знать, что последовательные запросы памяти не всегда приводят к выделению последовательных адресов памяти!

```
1 int *ptr1 = new int;
2 int *ptr2 = new int;
3 // ptr1 и ptr2 могут не иметь последовательных адресов памяти
```

При удалении динамически выделенной переменной, память возвращается обратно в кучу и затем может быть переназначена (исходя из последующих запросов). Помните, что удаление указателя не удаляет переменную, а просто приводит к возврату памяти по этому адресу обратно в операционную систему.

Куча имеет свои преимущества и недостатки:

- Выделение памяти в куче сравнительно медленное.
- Выделенная память остается выделенной до тех пор, пока не будет освобождена (остерегайтесь утечек памяти) или пока программа не завершит свое выполнение.
- Доступ к динамически выделенной памяти осуществляется только через указатель. Разыменование указателя происходит медленнее, чем доступ к переменной напрямую.
- Поскольку куча представляет собой большой резервуар памяти, то именно она используется для выделения больших [массивов](#), [структур](#) или классов.

Стек вызовов

Стек вызовов (или просто «*стек*») отслеживает все активные функции (те, которые были вызваны, но еще не завершены) от начала программы и до текущей точки выполнения, и обрабатывает выделение всех параметров функции и локальных переменных.

Стек вызовов реализуется как структура данных «Стек». Поэтому, прежде чем мы поговорим о том, как работает стек вызовов, нам нужно понять, что такое стек как структура данных.

Стек как структура данных

Структура данных в программировании — это механизм организации данных для их эффективного использования. Вы уже видели несколько типов структур данных, например, массивы или структуры. Существует множество других структур данных, которые используются в программировании. Некоторые из них реализованы в Стандартной библиотеке C++, и стек как раз является одним из таковых.

Например, рассмотрим стопку (аналогия стеку) тарелок на столе. Поскольку каждая тарелка тяжелая, а они еще и сложены друг на друге, то вы можете сделать лишь что-то одно из следующего:

- ➔ Посмотреть на поверхность первой тарелки (которая находится на самом верху).
- ➔ Взять верхнюю тарелку из стопки (обнажая таким образом следующую тарелку, которая находится под верхней, если она вообще существует).
- ➔ Положить новую тарелку поверх стопки (спрятав под ней самую верхнюю тарелку, если она вообще была).

В компьютерном программировании стек представляет собой контейнер (как структуру данных), который содержит несколько переменных (подобно массиву). Однако, в то время как массив позволяет получить доступ и изменять элементы в любом порядке (так называемый «*произвольный доступ*»), стек более ограничен.

В стеке вы можете:

- ➔ Посмотреть на верхний элемент стека (используя функцию `top()` или `peek()`).
- ➔ Вытянуть верхний элемент стека (используя функцию `pop()`).
- ➔ Добавить новый элемент поверх стека (используя функцию `push()`).

Стек — это структура данных типа **LIFO** (англ. «*Last In, First Out*» = «*Последним пришел, первым ушел*»). Последний элемент, который находится на вершине стека, первым и уйдет из него. Если положить новую тарелку поверх других тарелок, то именно эту тарелку вы первой и возьмете. По мере того, как элементы помещаются в стек — стек растет, по мере того, как элементы удаляются из стека — стек уменьшается.

Например, рассмотрим короткую последовательность, показывающую, как работает добавление и удаление в стеке:

```
Stack: empty
Push 1
Stack: 1
Push 2
Stack: 1 2
Push 3
```

```
Stack: 1 2 3
Push 4
Stack: 1 2 3 4
Pop
Stack: 1 2 3
Pop
Stack: 1 2
Pop
Stack: 1
```

Стопка тарелок довольно-таки хорошая аналогия работы стека, но есть лучшая аналогия. Например, рассмотрим несколько почтовых ящиков, которые расположены друг на друге. Каждый почтовый ящик может содержать только один элемент, и все почтовые ящики изначально пустые. Кроме того, каждый почтовый ящик прибивается гвоздем к почтовому ящику снизу, поэтому количество почтовых ящиков не может быть изменено. Если мы не можем изменить количество почтовых ящиков, то как мы получим поведение, подобное стеку?

Во-первых, мы используем наклейку для обозначения того, где находится самый нижний пустой почтовый ящик. Вначале это будет первый почтовый ящик, который находится на полу. Когда мы добавим элемент в наш стек почтовых ящиков, то мы поместим этот элемент в почтовый ящик, на котором будет наклейка (т.е. в самый первый пустой почтовый ящик на полу), а затем переместим наклейку на один почтовый ящик выше. Когда мы вытаскиваем элемент из стека, то мы перемещаем наклейку на один почтовый ящик ниже и удаляем элемент из почтового ящика. Всё, что находится ниже наклейки — находится в стеке. Всё, что находится в ящике с наклейкой и выше — находится вне стека.

Сегмент стека вызовов

Сегмент стека вызовов содержит память, используемую для стека вызовов. При запуске программы, функция `main()` помещается в стек вызовов операционной системой. Затем программа начинает свое выполнение.

Когда программа встречает вызов функции, то эта функция помещается в стек вызовов. При завершении выполнения функции, она удаляется из стека вызовов. Таким образом, просматривая функции, добавленные в стек, мы можем видеть все функции, которые были вызваны до текущей точки выполнения.

Наша аналогия с почтовыми ящиками — это действительно то, как работает стек вызовов. Стек вызовов имеет фиксированное количество адресов памяти (фиксированный размер). Почтовые ящики являются адресами памяти, а «элементы», которые мы добавляем или вытягиваем из стека, называются **фреймами** (или **«кадрами»**) **стека**. Кадр стека отслеживает все данные, связанные с одним вызовом функции. «Наклейка» — это регистр (небольшая часть памяти в ЦП), который является указателем стека. **Указатель стека** отслеживает вершину стека вызовов.

Единственное отличие фактического стека вызовов от нашего гипотетического стека почтовых ящиков заключается в том, что, когда мы вытягиваем элемент из стека вызовов, нам не нужно очищать память (т.е. вынимать всё содержимое из почтового ящика). Мы можем просто оставить эту память для следующего элемента, который и перезапишет её. Поскольку указатель стека будет ниже этого адреса памяти, то, как мы уже знаем, эта ячейка памяти не будет находиться в стеке.

Стек вызовов на практике

Давайте рассмотрим детально, как работает стек вызовов. Ниже приведена **последовательность шагов, выполняемых при вызове функции**:

- Программа сталкивается с вызовом функции.
- Создается фрейм стека, который помещается в стек. Он состоит из:
 - ➔ адреса инструкции, который находится за вызовом функции (так называемый **«обратный адрес»**). Так процессор запоминает, куда ему возвращаться после выполнения функции;
 - ➔ аргументов функции;
 - ➔ памяти для локальных переменных;
 - ➔ сохраненных копий всех регистров, модифицированных функцией, которые необходимо будет восстановить после того, как функция завершит свое выполнение.
- Процессор переходит к точке начала выполнения функции.
- Инструкции внутри функции начинают выполняться.

После завершения функции, выполняются следующие шаги:

- Регистры восстанавливаются из стека вызовов.
- Фрейм стека вытягивается из стека. Освобождается память, которая была выделена для всех локальных переменных и аргументов.
- Обрабатывается возвращаемое значение.
- ЦП возобновляет выполнение кода (исходя из обратного адреса).

Возвращаемые значения могут обрабатываться разными способами, в зависимости от архитектуры компьютера. Некоторые архитектуры считают возвращаемое значение частью фрейма стека, другие используют регистры процессора.

Знать все детали работы стека вызовов не так уж и важно. Однако понимание того, что функции при вызове добавляются в стек, а при завершении выполнения — удаляются из стека, дает основы, необходимые для понимания рекурсии, а также некоторых других концепций, которые полезны при [отладке программ](#).

Пример стека вызовов

Рассмотрим следующий фрагмент кода:

```
1 int boo(int b)
2 {
```

```
3 | // b
4 | return b;
5 | } // функция boo() вытягивается из стека вызовов здесь
6 |
7 | int main()
8 | {
9 |     // a
10 |    boo(7); // функция boo() добавляется в стек вызовов здесь
11 |    // c
12 |
13 |    return 0;
14 | }
```

Стек вызовов этой программы выглядит следующим образом:

a:

main()

b:

boo() (включая параметр b)

main()

c:

main()

Переполнение стека

Стек имеет ограниченный размер и, следовательно, может содержать только ограниченный объем информации. В операционной системе Windows размер стека по умолчанию составляет 1МБ. На некоторых Unix-системах этот размер может достигать и 8МБ. Если программа пытается поместить в стек слишком много информации, то это приведет к переполнению стека. **Переполнение стека** (англ. «*stack overflow*») происходит, когда запрашиваемой памяти нет в наличии (вся память уже занята).

Переполнение стека является результатом добавления слишком большого количества переменных в стек и/или создания слишком большого количества вложенных вызовов функций (например, когда функция A() вызывает функцию B(), которая вызывает функцию C(), а та, в свою очередь, вызывает функцию D() и т.д.). Переполнение стека обычно приводит к сбою в программе, например:

```
1 | int main()
2 | {
3 |     int stack[1000000000];
4 |     return 0;
5 | }
```

Эта программа пытается добавить огромный массив в стек вызовов. Поскольку размера стека недостаточно для обработки такого массива, то операция его добавления переходит и на другие части памяти, которые программа использовать не может. Следовательно, получаем сбой.

Вот еще одна программа, которая вызовет переполнение стека, но уже по другой причине:

```
1 void boo()  
2 {  
3     boo();  
4 }  
5  
6 int main()  
7 {  
8     boo();  
9  
10    return 0;  
11 }
```

В программе, приведенной выше, фрейм стека добавляется в стек каждый раз, когда вызывается функция `boo()`. Поскольку функция `boo()` вызывает сама себя бесконечное количество раз, то в конечном итоге в стеке не хватит памяти, что приведет к переполнению стека.

Стек имеет свои преимущества и недостатки:

- ➔ Выделение памяти в стеке происходит сравнительно быстро.
- ➔ Память, выделенная в стеке, остается в области видимости до тех пор, пока находится в стеке. Она уничтожается при выходе из стека.
- ➔ Вся память, выделенная в стеке, обрабатывается во время компиляции, следовательно, доступ к этой памяти осуществляется напрямую через переменные.
- ➔ Поскольку размер стека является относительно небольшим, то не рекомендуется делать что-либо, что съест много памяти стека (например, [передача по значению](#) или создание локальных переменных больших массивов или других *затратных* структур данных).

Оценить статью:

★★★★★ (212 оценок, среднее: 4,92 из 5)



[← Урок №104. Указатели на функции](#)

[Урок №106. Ёмкость вектора](#)



Комментариев: 3



1. *Spardoks:*

[3 декабря 2020 в 21:14](#)

В стеке можно разместить исполняемый код, если постараться. Есть даже такая уязвимость, связанная с переполнением стека. Тем не менее, в адекватной ситуации код там не хранится.

[Ответить](#)



2. *bazik210:*

[22 февраля 2019 в 23:07](#)

По поводу вот этого утверждения: "Когда программа встречает вызов функции, то эта функция помещается в стек вызовов." От того преподавателя с ITVDN: "Исполняемый код НИКОГДА!!! не может оказаться в стеке, это статическая область памяти, instruction pointer (регистр процессора) никогда не перейдет туда, чтобы её выполнять, процессор никогда не зайдет в стек и не начнет там выполнять программные коды. Это просто коробка для хранения адресов и локальных переменных."

[Ответить](#)



1. *Uraut:*

[24 июня 2019 в 05:24](#)

Разумеется в стеке исполняемого кода быть не может. При вызове функции все ее аргументы помещаются в стек в обратном порядке, а следом происходит вызов функции, который изымет из стека эти аргументы и поместит в стек свой вызов.

На примере автора: когда функция `foo` рекурсивно вызывает саму себя, переполнение стека произойдет и это факт! При каждом вызове в стек будет помещаться по 4 байта (32 разрядные системы), но изъятия из стека происходить не будет так как функция не доходит до завершения.

[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.

Отправить комментарий

[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