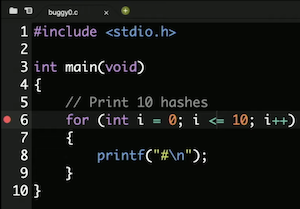
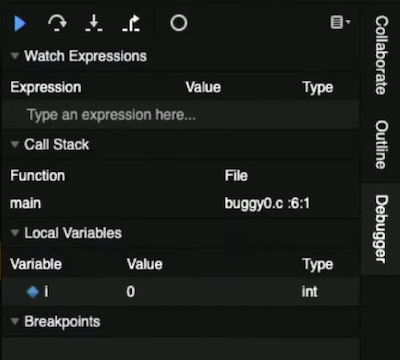
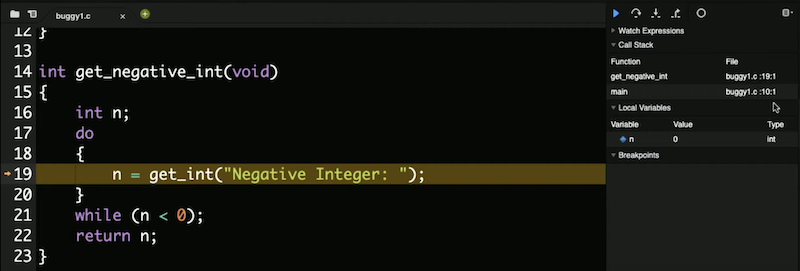
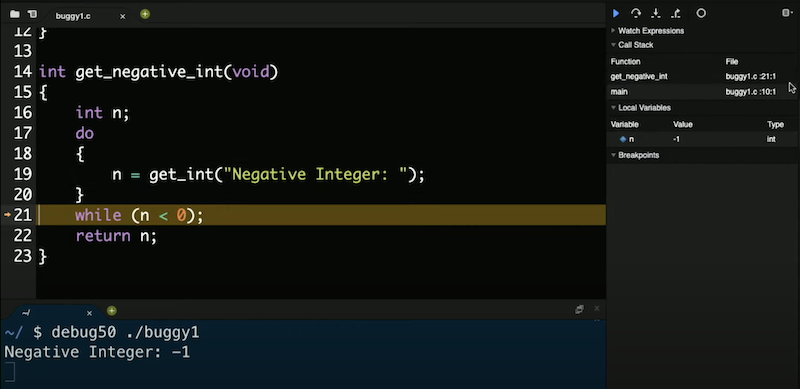
[**Чтение 2**](https://cs50.harvard.edu/x/2021/notes/2/#lecture-2)

* [Компиляция](https://cs50.harvard.edu/x/2021/notes/2/#compiling)
* [Отладка](https://cs50.harvard.edu/x/2021/notes/2/#debugging)
* [объем памяти](https://cs50.harvard.edu/x/2021/notes/2/#memory)
* [Массивы](https://cs50.harvard.edu/x/2021/notes/2/#arrays)
* [Символы](https://cs50.harvard.edu/x/2021/notes/2/#characters)
* [Струны](https://cs50.harvard.edu/x/2021/notes/2/#strings)
* [Аргументы командной строки](https://cs50.harvard.edu/x/2021/notes/2/#command-line-arguments)
* [Приложения](https://cs50.harvard.edu/x/2021/notes/2/#applications)

[**Компиляция**](https://cs50.harvard.edu/x/2021/notes/2/#compiling)

* В прошлый раз мы научились писать нашу первую программу на C, выводя на экран «привет, мир».
* make helloСначала мы скомпилировали его , превратив исходный код в машинный код, прежде чем мы смогли запустить скомпилированную программу с помощью ./hello.
* makeна самом деле это просто программа, которая вызывает clangкомпилятор с параметрами. Мы могли бы hello.cсами скомпилировать наш файл исходного кода , выполнив команду clang hello.c. Кажется, что ничего не произошло, значит, ошибок не было. И если мы запустим ls, то теперь мы видим a.outфайл в нашем каталоге. Имя файла по - прежнему по умолчанию, так что мы можем работать с более конкретной команды: clang -o hello hello.c.
* Мы добавили еще один **аргумент командной строки** или ввод в программу в командной строке в виде дополнительных слов после имени программы. clangэто название программы, а также -o, helloи hello.cдополнительные аргументы. Мы говорим clangиспользовать helloв качестве *выходного* имени файла и использовать hello.cв качестве исходного кода. Теперь мы видим, helloчто создается как результат.
* Если мы хотим использовать библиотеку CS50 #include <cs50.h>для get\_stringфункции, мы также должны добавить флаг clang -o hello hello.c -lcs50:
* #include <cs50.h>
* #include <stdio.h>
* int main(void)
* {
* string name = get\_string("What's your name? ");
* printf("hello, %s\n", name);
* }
  + -lФлаг связывает cs50файл, который уже установлен в CS50 IDE, и включает в себя машинный код get\_string(наряду с другими функциями) , что наша программа может обратиться и использования , а также.
* С make, эти аргументы генерируются для нас, поскольку персонал уже настроил makeв CS50 IDE.
* Компиляция исходного кода в машинный код фактически состоит из более мелких шагов:
  + предварительная обработка
  + составление
  + сборка
  + связывание
* **В предварительной обработке** обычно используются строки, начинающиеся с символа #вроде #include. Например, #include <cs50.h>он скажет clangискать этот файл заголовка, поскольку он содержит контент, который мы хотим включить в нашу программу. Затем, по clangсути, заменит содержимое этих файлов заголовков в нашей программе.
* Например …
* #include <cs50.h>
* #include <stdio.h>
* int main(void)
* {
* string name = get\_string("What's your name? ");
* printf("hello, %s\n", name);
* }
* … Будет предварительно обработан в:
* ...
* string get\_string(string prompt);
* int printf(string format, ...);
* ...
* int main(void)
* {
* string name = get\_string("Name: ");
* printf("hello, %s\n", name);
* }
* Сюда входят прототипы всех функций из тех библиотек, которые мы включили, поэтому мы можем затем использовать их в нашем коде.
* **Компиляция** берет наш исходный код на C и преобразует его в другой тип исходного кода, называемый **кодом сборки** , который выглядит следующим образом:
* ...
* main: # @main
* .cfi\_startproc
* # BB#0:
* pushq %rbp
* .Ltmp0:
* .cfi\_def\_cfa\_offset 16
* .Ltmp1:
* .cfi\_offset %rbp, -16
* movq %rsp, %rbp
* .Ltmp2:
* .cfi\_def\_cfa\_register %rbp
* subq $16, %rsp
* xorl %eax, %eax
* movl %eax, %edi
* movabsq $.L.str, %rsi
* movb $0, %al
* callq get\_string
* movabsq $.L.str.1, %rdi
* movq %rax, -8(%rbp)
* movq -8(%rbp), %rsi
* movb $0, %al
* callq printf
* ...
  + Эти инструкции являются низкоуровневыми и ближе к двоичным инструкциям, которые процессор компьютера может понять напрямую. Обычно они работают с самими байтами, а не с абстракциями, такими как имена переменных.
* Следующий шаг - взять ассемблерный код и преобразовать его в инструкции в двоичном формате путем его **сборки** . Инструкции в двоичном формате называются **машинным кодом** , который процессор компьютера может запускать напрямую.
* Последний шаг - это **компоновка** , когда ранее скомпилированные версии библиотек, которые мы включили ранее cs50.c, фактически объединяются с двоичным кодом нашей программы. Таким образом , мы в конечном итоге с одним бинарным файлом, a.outили hello, что является комбинированной машиной кода hello.c, cs50.cи stdio.c. (В среде IDE CS50 предварительно скомпилированный машинный код для cs50.cи stdio.cуже установлен и clangнастроен для их поиска и использования.)
* Эти четыре шага были абстрагированы или упрощены, makeпоэтому все, что нам нужно реализовать, - это код для наших программ.

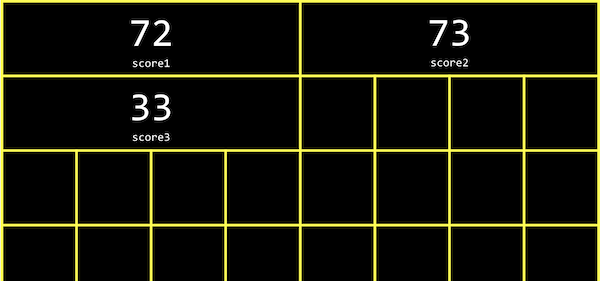
[**Отладка**](https://cs50.harvard.edu/x/2021/notes/2/#debugging)

* **Ошибки** - это ошибки или проблемы в программах, из-за которых они ведут себя не так, как предполагалось. А отладка - это процесс поиска и исправления этих ошибок.
* На прошлой неделе мы узнали о нескольких инструментах, которые помогают нам писать код, который компилируется, имеет хороший стиль и правильный:
  + help50
  + style50
  + check50
* Мы можем использовать другой «инструмент», printfфункцию, для вывода сообщений и переменных, которые помогут нам отлаживать.
* Давайте посмотрим на buggy0.c:
* #include <stdio.h>
* int main(void)
* {
* // Print 10 hashes
* for (int i = 0; i <= 10; i++)
* {
* printf("#\n");
* }
* }
  + Хм, мы хотим напечатать только 10 #секунд, а их 11. Если бы мы не знали, в чем проблема (поскольку наша программа компилируется без каких-либо ошибок, и теперь у нас есть логическая ошибка), мы могли бы printfвременно добавить еще одну :
  + #include <stdio.h>
  + int main(void)
  + {
  + for (int i = 0; i <= 10; i++)
  + {
  + printf("i is now %i\n", i);
  + printf("#\n");
  + }
  + }
  + Теперь мы можем видеть, что это iначалось с 0 и продолжалось до тех пор, пока оно не стало 10, но мы должны forостановить наш цикл, как только он достигнет 10, с i < 10вместо i <= 10.
* В IDE CS50 у нас есть еще один инструмент **debug50**, который помогает нам отлаживать программы. Это инструмент, написанный сотрудниками, который построен на стандартном инструменте под названием gdb. Оба этих **отладчика** - это программы, которые будут запускать наши собственные программы шаг за шагом и позволяют нам просматривать переменные и другую информацию во время работы нашей программы.
* Мы запустим команду debug50 ./buggy0, и она скажет нам перекомпилировать нашу программу, поскольку мы ее изменили. Затем он скажет нам добавить **точку останова** или индикатор для строки кода, где отладчик должен приостановить нашу программу.
  + Используя клавиши «вверх» и «вниз» в терминале, мы можем повторно использовать команды из прошлого, не вводя их снова.
* Мы щелкнем слева от строки 6 в нашем коде, и появится красный кружок:  
  
* Теперь, если мы debug50 ./buggy0снова запустим , мы увидим, что панель отладчика открыта справа:  
  
* Мы видим, что созданная нами переменная iнаходится под Local Variablesразделом, и видим, что там есть значение 0.
* Наша точка останова приостановила нашу программу в строке 6, выделив эту строку желтым цветом. Чтобы продолжить, у нас есть несколько элементов управления на панели отладчика. Синий треугольник будет продолжать нашу программу до тех пор, пока мы не достигнем другой точки останова или до конца нашей программы. Изогнутая стрелка справа, Step Over, «перешагнет» линию, запустит ее и сразу после этого снова приостановит нашу программу.
* Итак, мы будем использовать изогнутую стрелку, чтобы перейти к следующей строке, и посмотрим, что изменится после. Мы находимся на printfлинии, и, снова нажав на изогнутую стрелку, мы видим единственную, #напечатанную в окне нашего терминала. Еще одним щелчком стрелки мы видим значение iизменения на 1. Мы можем продолжать щелкать стрелку, чтобы наблюдать за работой нашей программы, по одной строке за раз.
* Чтобы выйти из отладчика, мы можем нажать, control + Cчтобы остановить запущенную программу.
* Давайте посмотрим на другой пример buggy1.c:
* #include <cs50.h>
* #include <stdio.h>
* // Prototype
* int get\_negative\_int(void);
* int main(void)
* {
* // Get negative integer from user
* int i = get\_negative\_int();
* printf("%i\n", i);
* }
* int get\_negative\_int(void)
* {
* int n;
* do
* {
* n = get\_int("Negative Integer: ");
* }
* while (n < 0);
* return n;
* }
  + Мы реализовали другую функцию, get\_negative\_intчтобы получить от пользователя отрицательное целое число. Нам нужно запомнить прототип перед нашей mainфункцией, а затем наш код компилируется.
* Но когда мы запускаем нашу программу, она продолжает запрашивать отрицательное целое число даже после того, как мы его предоставили. Мы установим точку останова в строке 10, int i = get\_negative\_int();поскольку это первая интересная строка кода. Мы запустим debug50 ./buggy1и увидим в разделе «Стек вызовов» панели отладки, что мы находимся в mainфункции. («Стек вызовов» относится ко всем функциям, которые были вызваны в нашей программе в то время, но еще не вернулись из mainнее . До сих пор была вызвана только функция.)
* Мы щелкнем стрелку, указывающую вниз, «Шаг с заходом», и отладчик перенесет нас *в* функцию, вызываемую в этой строке get\_negative\_int. Мы видим, как в стек вызовов добавлено имя функции и переменная nсо значением 0:  
  
* Мы можем снова щелкнуть стрелку Step Over и посмотреть, nбудет ли обновлено -1, что мы и в самом деле ввели:  
  
* Мы снова нажимаем Step Over, и мы видим, что наша программа возвращается в цикл. Наш whileцикл все еще работает, поэтому условие, которое он проверяет, должно оставаться trueнеизменным. И мы видим, что n < 0это правда, даже если мы ввели отрицательное целое число, поэтому мы должны исправить нашу ошибку, изменив ее на n >= 0.
* Мы можем сэкономить много времени в будущем, вложив немного сейчас, чтобы научиться пользоваться debug50!
* Мы также можем использовать ddb, сокращенно от «duck debugger», [реальную технику, в](https://en.wikipedia.org/wiki/Rubber_duck_debugging) которой мы объясняем, что мы пытаемся сделать с резиновой уткой, и часто мы осознаем нашу собственную ошибку в логике или реализации, когда мы объясняем это.

[**объем памяти**](https://cs50.harvard.edu/x/2021/notes/2/#memory)

* В C у нас есть разные типы переменных, которые мы можем использовать для хранения данных, и каждая из них занимает фиксированное количество места. Различные компьютерные системы на самом деле различаются по объему фактически используемого пространства для каждого типа, но мы будем работать с объемами здесь, которые используются в CS50 IDE:
  + bool 1 byte
  + char 1 byte
  + double 8 bytes
  + float 4 bytes
  + int 4 bytes
  + long 8 bytes
  + string ? bytes
  + …
* Внутри наших компьютеров, у нас есть чипы , называемых RAM, произвольный доступ **памятью** , которая хранит данные для краткосрочного использования, как кода программы в то время как он работает, или файл , пока он открыт. Мы можем сохранить программу или файл на нашем жестком диске (или SSD, твердотельном накопителе) для длительного хранения, но использовать оперативную память, потому что это намного быстрее. Однако ОЗУ энергозависимо или требует питания для хранения данных.
* Мы можем представить байты, хранящиеся в ОЗУ, как если бы они были в сетке:  
  
  + На самом деле в каждом чипе миллионы или миллиарды байтов.
  + Каждый байт будет иметь место на микросхеме, например, первый байт, второй байт и т. Д.
* В C, когда мы создаем переменную типа char, размер которой будет равен одному байту, она будет физически храниться в одном из этих ящиков в ОЗУ. Целое число с 4 байтами займет четыре из этих ящиков.

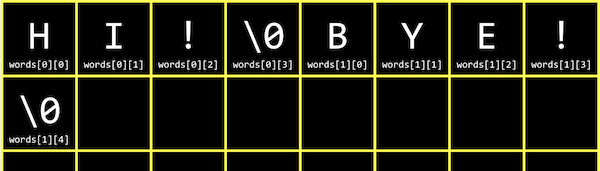
[**Массивы**](https://cs50.harvard.edu/x/2021/notes/2/#arrays)

* Допустим, мы хотели взять среднее значение трех переменных:
* #include <stdio.h>
* int main(void)
* {
* int score1 = 72;
* int score2 = 73;
* int score3 = 33;
* printf("Average: %f\n", (score1 + score2 + score3) / 3.0);
* }
  + Мы делим на «нет» 3, но 3.0результат тоже будет с плавающей запятой.
  + Мы можем скомпилировать и запустить нашу программу и увидеть среднее значение напечатанного.
* Пока наша программа работает, три intпеременные хранятся в памяти:  
  
  + Каждый intзанимает четыре блока, представляющих четыре байта, и каждый байт, в свою очередь, состоит из восьми битов, нулей и единиц, хранящихся в электрических компонентах.
* Оказывается, в памяти мы можем хранить переменные одну за другой, спина к спине, и получать к ним доступ с помощью циклов. В C список значений, хранящихся друг за другом, называется **массивом** .
* Для нашей программы выше мы можем int scores[3];вместо этого объявить массив из трех целых чисел.
* И мы можем назначать и использовать переменные в массиве с помощью scores[0] = 72. С помощью скобок мы индексируем или переходим к «0-й» позиции в массиве. Массивы имеют нулевой индекс, что означает, что первое значение имеет индекс 0, а второе значение имеет индекс 1 и т. Д.
* Давайте обновим нашу программу, чтобы использовать массив:
* #include <cs50.h>
* #include <stdio.h>
* int main(void)
* {
* int scores[3];
* scores[0] = get\_int("Score: ");
* scores[1] = get\_int("Score: ");
* scores[2] = get\_int("Score: ");
* // Print average
* printf("Average: %f\n", (scores[0] + scores[1] + scores[2]) / 3.0);
* }
  + Теперь мы запрашиваем у пользователя три значения и печатаем среднее значение, как и раньше, но с использованием значений, хранящихся в массиве.
* Поскольку мы можем устанавливать элементы в массиве и получать к ним доступ в зависимости от их положения, и эта позиция *также* может быть значением некоторой переменной, мы можем использовать цикл:
* #include <cs50.h>
* #include <stdio.h>
* int main(void)
* {
* int scores[3];
* for (int i = 0; i < 3; i++)
* {
* scores[i] = get\_int("Score: ");
* }
* // Print average
* printf("Average: %f\n", (scores[0] + scores[1] + scores[2]) / 3.0);
* }
  + Теперь вместо жесткого кодирования или трехкратного указания каждого элемента вручную мы используем forцикл и iв качестве индекса каждого элемента в массиве.
* И мы повторили значение 3, представляющее длину нашего массива, в двух разных местах. Таким образом, мы можем использовать в нашей программе **константу** или переменную с фиксированным значением:
* #include <cs50.h>
* #include <stdio.h>
* const int TOTAL = 3;
* int main(void)
* {
* int scores[TOTAL];
* for (int i = 0; i < TOTAL; i++)
* {
* scores[i] = get\_int("Score: ");
* }
* printf("Average: %f\n", (scores[0] + scores[1] + scores[2]) / TOTAL);
* }
  + Мы можем использовать constключевое слово, чтобы сообщить компилятору, что значение TOTALникогда не должно изменяться нашей программой. И по соглашению мы разместим наше объявление переменной вне mainфункции и сделаем ее имя заглавными, что не обязательно для компилятора, но показывает другим людям, что эта переменная является константой, и позволяет легко увидеть ее с самого начала.
  + Но теперь наше среднее значение будет неверным или неверным, если у нас нет ровно трех значений.
* Добавим функцию для вычисления среднего:
* float average(int length, int array[])
* {
* int sum = 0;
* for (int i = 0; i < length; i++)
* {
* sum += array[i];
* }
* return sum / (float) length;
* }
  + Мы передадим длину и массив ints (который может быть любого размера) и воспользуемся другим циклом внутри нашей вспомогательной функции, чтобы сложить значения в sumпеременную. Мы используем (float)для литья lengthв поплавок, так что результат мы получаем от деления двух также поплавок.
  + Теперь в нашей mainфункции мы можем вызвать нашу новую averageфункцию с помощью printf("Average: %f\n", average(TOTAL, scores);. Обратите внимание, что имена переменных в mainне обязательно должны совпадать с тем, что averageих вызывает, поскольку передаются только *значения* .
  + Нам нужно передать в averageфункцию длину массива , чтобы она знала, сколько там значений.

[**Символы**](https://cs50.harvard.edu/x/2021/notes/2/#characters)

* Мы можем распечатать один символ с помощью простой программы:
* #include <stdio.h>
* int main(void)
* {
* char c = '#';
* printf("%c\n", c);
* }
  + Когда мы запускаем эту программу, мы получаем #печать в терминале.
* Посмотрим, что произойдет, если мы изменим нашу программу на печать cцелым числом:
* #include <stdio.h>
* int main(void)
* {
* char c = '#';
* printf("%i\n", (int) c);
* }
  + Когда мы запускаем эту программу, мы получаем 35распечатку. Оказывается, это 35действительно код ASCII для символа #.
  + Фактически, нам не нужно явно cприводить int; в этом случае компилятор может сделать это за нас.
* A char- это один байт, поэтому мы можем представить его хранящимся в одном блоке в сетке памяти выше.

[**Струны**](https://cs50.harvard.edu/x/2021/notes/2/#strings)

* Мы можем распечатать строку или некоторый текст, создав переменную для каждого символа и распечатав их:
* #include <stdio.h>
* int main(void)
* {
* char c1 = 'H';
* char c2 = 'I';
* char c3 = '!';
* printf("%c%c%c\n", c1, c2, c3);
* }
  + Здесь мы увидим HI!распечатку.
* Теперь распечатаем целые значения каждого символа:
* #include <stdio.h>
* int main(void)
* {
* char c1 = 'H';
* char c2 = 'I';
* char c3 = '!';
* printf("%i %i %i\n", c1, c2, c3);
* }
  + Мы увидим 72 73 33распечатку и поймем, что эти символы хранятся в памяти вот так:  
    
* **Строки** на самом деле представляют собой просто массивы символов и определены не в C, а библиотекой CS50. Если бы мы имели массив называется s, каждый символ может быть доступен с s[0], s[1]и так далее.
* И оказывается, что строка заканчивается специальным символом, '\0'или байтом, все биты которого установлены в 0. Этот символ называется **нулевым символом** или NUL. Таким образом, нам действительно нужно четыре байта для хранения нашей строки с тремя символами:  
  
* Мы можем использовать строку в качестве массива в нашей программе и распечатать коды ASCII или целочисленные значения каждого символа в строке:
* #include <cs50.h>
* #include <stdio.h>
* int main(void)
* {
* string s = "HI!";
* printf("%i %i %i %i\n", s[0], s[1], s[2], s[3]);
* }
  + И как мы могли ожидать, мы видим 72 73 33 0напечатанные.
  + Фактически, мы могли бы попытаться получить доступ s[4]и увидеть напечатанный неожиданный символ. С помощью C наш код имеет возможность обращаться к памяти или изменять ее, чего в противном случае он не должен был бы делать, что одновременно мощно и опасно.
* Мы можем использовать цикл для вывода каждого символа в строке:
* #include <cs50.h>
* #include <stdio.h>
* int main(void)
* {
* string s = get\_string("Input: ");
* printf("Output: ");
* for (int i = 0; s[i] != '\0'; i++)
* {
* printf("%c", s[i]);
* }
* printf("\n");
* }
* Мы можем изменить условие нашего цикла для продолжения независимо от того i, что есть, а только тогда s[i] != '\0', когда или когда символ в текущей позиции в s *не* является нулевым символом.
* Мы можем использовать функцию, которая поставляется с stringбиблиотекой C , strlenчтобы получить длину строки для нашего цикла:
* #include <cs50.h>
* #include <stdio.h>
* #include <string.h>
* int main(void)
* {
* string s = get\_string("Input: ");
* printf("Output: ");
* for (int i = 0; i < strlen(s); i++)
* {
* printf("%c", s[i]);
* }
* printf("\n");
* }
* У нас есть возможность улучшить дизайн нашей программы. Наш цикл был немного неэффективным, поскольку мы проверяли длину строки после того, как каждый символ напечатан в нашем условии. Но поскольку длина строки не меняется, мы можем проверить длину строки один раз:
* #include <cs50.h>
* #include <stdio.h>
* #include <string.h>
* int main(void)
* {
* string s = get\_string("Input: ");
* printf("Output:\n");
* for (int i = 0, n = strlen(s); i < n; i++)
* {
* printf("%c\n", s[i]);
* }
* }
  + Теперь, в начале нашего цикла, мы инициализируем iи nпеременную, и переменную и запоминаем длину нашей строки в n. Затем мы можем проверять значения, не strlenвызывая каждый раз для вычисления длины строки.
  + И нам действительно нужно было использовать немного больше памяти для хранения n, но это сэкономит нам время, поскольку нам не нужно каждый раз проверять длину строки.
* Мы могли бы объявить массив из двух строк:
* string words[2];
* words[0] = "HI!";
* words[1] = "BYE!";
* А в памяти массив строк может быть сохранен и доступен с помощью:  
  
  + words[0]относится к первому элементу или значению wordsмассива, которое является строкой, и, таким образом, words[0][0]относится к первому элементу в этой строке, который является символом.
  + Итак, массив строк - это просто массив массивов символов.
* Теперь мы можем объединить то, что мы видели, чтобы написать программу, которая может использовать заглавные буквы:
* #include <cs50.h>
* #include <stdio.h>
* #include <string.h>
* int main(void)
* {
* string s = get\_string("Before: ");
* printf("After: ");
* for (int i = 0, n = strlen(s); i < n; i++)
* {
* if (s[i] >= 'a' && s[i] <= 'z')
* {
* printf("%c", s[i] - 32);
* }
* else
* {
* printf("%c", s[i]);
* }
* }
* printf("\n");
* }
  + Сначала мы получаем строку sот пользователя. Затем для каждого символа в строке, если он строчный (что означает, что он имеет значение между значением aи z), мы преобразуем его в верхний регистр. В противном случае мы просто печатаем его.
  + Мы можем преобразовать строчную букву в ее эквивалент в верхнем регистре, вычтя разницу между их значениями ASCII. (Мы знаем, что строчные буквы имеют более высокое значение ASCII, чем прописные буквы, и разница между теми же буквами такая же, поэтому мы можем вычесть, чтобы получить прописную букву из строчной буквы.)
* Оказывается, есть еще одна библиотека ctype.h, которую мы можем использовать:
* #include <cs50.h>
* #include <ctype.h>
* #include <stdio.h>
* #include <string.h>
* int main(void)
* {
* string s = get\_string("Before: ");
* printf("After: ");
* for (int i = 0, n = strlen(s); i < n; i++)
* {
* if (islower(s[i]))
* {
* printf("%c", toupper(s[i]));
* }
* else
* {
* printf("%c", s[i]);
* }
* }
* printf("\n");
* }
  + Теперь наш код более читабелен и, вероятно, будет правильным, поскольку другие написали и протестировали эти функции для нас.
* Мы можем еще больше упростить и просто передать каждый символ в toupper, поскольку он не меняет символы, отличные от нижнего регистра:
* #include <cs50.h>
* #include <ctype.h>
* #include <stdio.h>
* #include <string.h>
* int main(void)
* {
* string s = get\_string("Before: ");
* printf("After: ");
* for (int i = 0, n = strlen(s); i < n; i++)
* {
* printf("%c", toupper(s[i]));
* }
* printf("\n");
* }
* Мы можем использовать [**справочные страницы**](https://man.cs50.io/) CS50, чтобы найти и узнать об общих библиотечных функциях. При поиске по страницам руководства мы видим, что toupper()есть функция, среди прочего, из библиотеки ctype, которую мы можем использовать.

[**Аргументы командной строки**](https://cs50.harvard.edu/x/2021/notes/2/#command-line-arguments)

* Наши собственные программы также могут принимать аргументы командной строки или слова, добавленные после имени нашей программы в самой команде.
* В argv.c, мы меняем mainвнешний вид нашей функции:
* #include <cs50.h>
* #include <stdio.h>
* int main(int argc, string argv[])
* {
* if (argc == 2)
* {
* printf("hello, %s\n", argv[1]);
* }
* else
* {
* printf("hello, world\n");
* }
* }
  + argcи argvдве переменные, которые наша mainфункция теперь будет получать автоматически, когда наша программа запускается из командной строки. argcэто на *количество аргументов* , или число аргументов, и argv, *аргумент вектор* (или список аргументов), массив строк.
  + Первый аргумент argv[0]- это имя нашей программы (первое набранное слово, например ./hello). В этом примере мы проверяем, есть ли у нас два аргумента, и распечатываем второй, если да.
  + Например, если мы запустим ./argv David, нас hello, Davidнапечатают, поскольку мы ввели Davidвторое слово в нашей команде.
* Мы также можем распечатать каждый символ по отдельности:
* #include <cs50.h>
* #include <stdio.h>
* #include <string.h>
* int main(int argc, string argv[])
* {
* if (argc == 2)
* {
* for (int i = 0, n = strlen(argv[1]); i < n; i++)
* {
* printf("%c\n", argv[1][i]);
* }
* }
* }
  + Мы будем использовать argv[1][i]для доступа к каждому символу в первом аргументе нашей программы.
* Оказывается, наша mainфункция тоже возвращает целочисленное значение. По умолчанию наша mainфункция возвращается, 0чтобы указать, что ничего не случилось, но мы можем написать программу, возвращающую другое значение:
* #include <cs50.h>
* #include <stdio.h>
* int main(int argc, string argv[])
* {
* if (argc != 2)
* {
* printf("missing command-line argument\n");
* return 1;
* }
* printf("hello, %s\n", argv[1]);
* return 0;
* }
  + Возвращаемое значение mainв нашей программе называется **кодом выхода** и обычно используется для обозначения кодов ошибок. (Мы напишем здесь return 0явно в конце нашей программы, хотя технически нам это не нужно.)
* Когда мы пишем более сложные программы, такие коды ошибок могут помочь нам определить, что пошло не так, даже если это не видно и не имеет значения для пользователя.

[**Приложения**](https://cs50.harvard.edu/x/2021/notes/2/#applications)

* Теперь, когда мы знаем, как работать со строками в наших программах, а также с кодом, написанным другими в библиотеках, мы можем анализировать абзацы текста на предмет их читабельности на основе таких факторов, как длина и сложность слов и предложений.
* **Криптография** - это искусство **шифрования** или сокрытия информации. Если мы хотим отправить кому-то сообщение, мы можем захотеть **зашифровать** или каким-то образом зашифровать это сообщение, чтобы другим было трудно его прочитать. Исходное сообщение или ввод для нашего алгоритма называется **открытым текстом** , а зашифрованное сообщение или вывод называется **зашифрованным текстом** . А алгоритм, выполняющий скремблирование, называется **шифром** . Шифр обычно требует другого ввода в дополнение к открытому тексту. **Ключ** , как номер, какой - то другой вход , который держится в секрете.
* Например, если мы хотим , чтобы отправить сообщение , как I L O V E Y O Uмы можем сначала преобразовать его в ASCII: 73 76 79 86 69 89 79 85. Тогда мы можем зашифровать его с ключом просто 1и простым алгоритмом, в котором мы просто добавить ключ к каждому значению: 74 77 80 87 70 90 80 86. Тогда зашифрованный текст после преобразования значений обратно в ASCII будет J M P W F Z P V. Чтобы расшифровать это, кто-то должен знать ключ 1и вычитать его из каждого символа!
* Мы применим эти концепции в наших разделах и наборе задач!