Смысл программ

Что делает моя программа?

Когда мы программируем, важно иметь кристально ясное понимание того, что именно делает та или иная конструкция языка. Что именно произойдёт в компьютере, когда он будет выполнять ту или иную строчку кода?

Как достичь этого понимания? Задачу усложняет то, что компьютер не умеет напрямую выполнять строчки кода на C: он выполняет только программы в машинных кодах. Специальная программа, называемая *компилятор*, переводит текст программы на C в набор машинных кодов. Такой код очень сложно прочитать и понять: он описывает функционирование программы в терминах базовой арифметики и операций с памятью. В машинном коде даже нет переменных или функций.

Компиляторы тоже усложняют задачу понимания смысла программы:

1. Компиляторы генерируют машинный код из программы на C очень сложным и неочевидным образом. Они оптимизируют машинный код так, чтобы он выполнялся быстрее, но при этом он становится менее понятным.
2. В компиляторе может быть ошибка и он неправильно скомпилирует программу. В ситуации, когда программа делает не то, что мы ожидаем, стоит ли искать ошибку в компиляторе, или мы просто неправильно понимаем, что на самом деле должна делать программа?
3. На процессорах с разной архитектурой машинные коды могут быть совсем разные и несравнимые друг с другом (например, AMD64 в персональных компьютерах и ARM в телефонах и планшетах)

По этим причинам мы не можем на вопрос "Что должна делать моя программа?" отвечать: "Скомпилируем, запустим, и как она будет работать – то она и должна делать". На разных процессорах, с разными компиляторами, программа будет работать немного по-разному, да и прочитать и понять машинный код – долгая, нудная и сложная задача.

Абстрактный вычислитель

Чтобы точно описать, чего мы ожидаем от всех конструкций языка, мы вводим модель воображаемого, упрощённого компьютера, который умеет выполнять программы на этом языке "как родные". Это *абстрактный вычислитель*. Мы говорили, что язык C низкоуровневый – это означает, что абстрактный вычислитель C похож на реальный компьютер.

На схеме ниже мы видим:

- Память, хранящую код программы и необходимые ей данные. Эта память (которую также называют *memory)* примерно соответствует оперативной памяти компьютера, не жёсткому диску/SSD накопителю (которые для различия называют *storage,* хранилище).
- Процессор, выполняющий программу. Он читает из памяти программу, необходимые ей данные и выполняет её statement'ы.

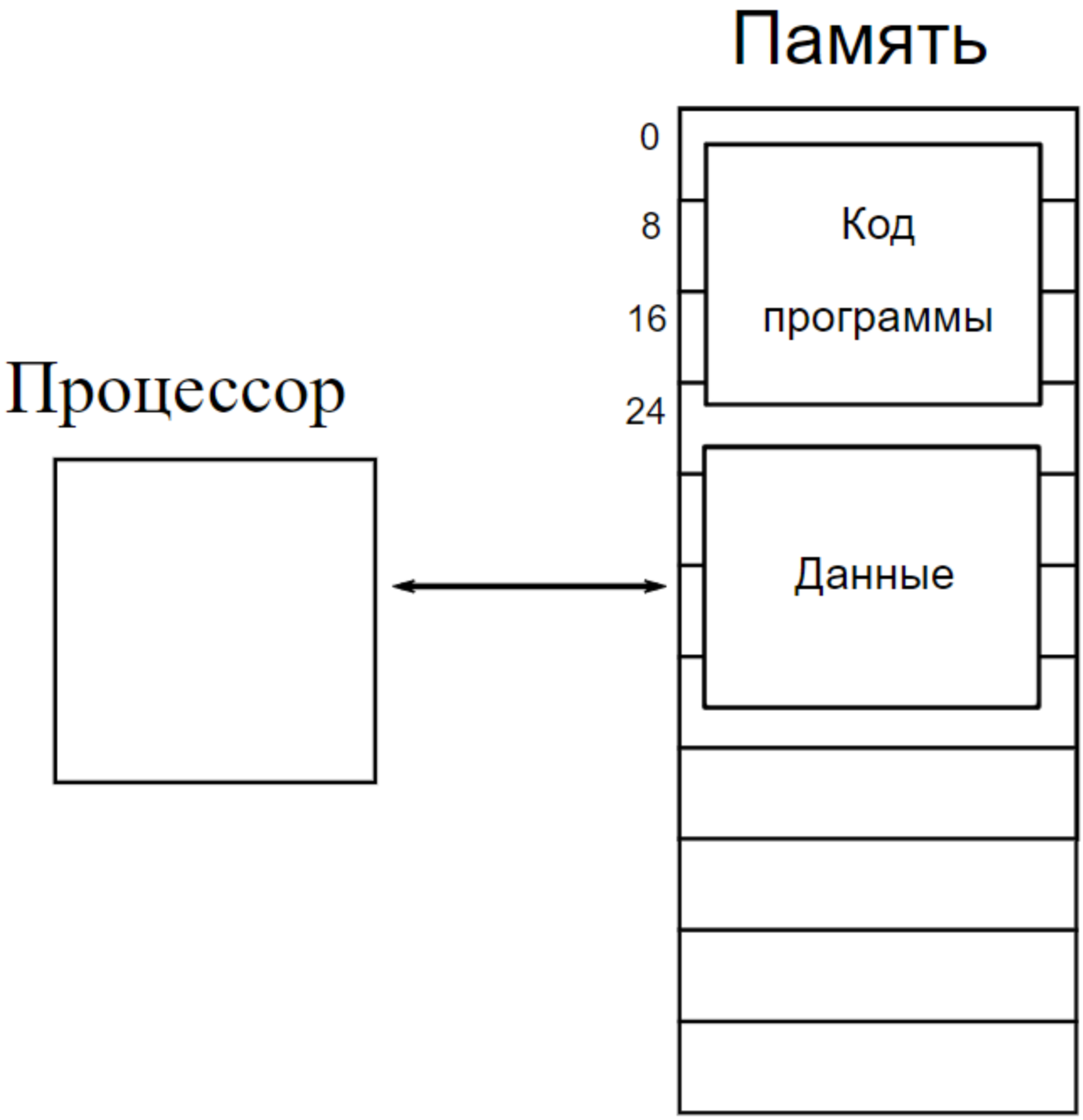
Память хранит нули и единицы (биты) в пачках по восемь (одна пачка называется *байт* и хранит 8 бит). Каждый байт имеет свой *адрес* -- число от 0 до некоторого максимального адреса памяти. Это называется *линейной организацией памяти*, когда все её ячейки перенумерованы, а номера идут один за другим без пропусков. Сама программа тоже состоит из нулей и единиц, т.е. закодирована.

Перечисленные нами характеристики процессора и памяти называются принципами фон Неймана, а сам абстрактный вычислитель языка С имеет архитектуру фон Неймана.

Для нашего описания абстрактного вычислителя С мы также будем использовать стек -- специальную область памяти для организации вызовов функций и хранения их данных; об этом мы узнаем чуть позже в этом уроке. При этом память одна, а стек является её частью.

Хотя память организована линейно, нам будет удобно изображать её стопкой из кусочков по 8 байт (ячеек) каждая. Первый такой кусочек начинается с адреса 0, второй -- с адреса 8, и так далее. В одном байте 8 бит, поэтому в каждом кусочке в совокупности по 8*8 = 64 бита.

Но помните: мы по прежнему можем прочитать 8 байт начиная с любого адреса не кратного восьми, например, с адресов 4, 5 или 29.



Глобальные переменные

Переменные это именованные кусочки памяти. Можно читать их содержимое или перезаписывать его. В исходном коде программы имя переменной означает обращение к её памяти.

Каждая переменная имеет тип; пока мы ограничиваемся переменными типа `int`. На современных 64-разрядных компьютерах одна такая переменная занимает 4 байт памяти[1].

Проще всего обращаться с *глобальными* переменными. Вот пример глобальной переменной `g` и её использования; функция `main` выведет `41 42`

```
int g;                      // объявление переменной с именем g и типом int


void f1() {
   g = 41;                  // написав g = <expr> мы записываем в g новое значение.
   printf( "%d ", g );      // просто написав имя переменной мы прочитаем её
значение.
   g = g + 1;
}


void f2() {
   printf( "%d ", g );
}


void main() {
   f1();
   f2();                    // здесь g == 41 + 1
}
```

В исходном коде глобальные переменные объявляются вне функций. По умолчанию значение глобальных переменных равно нулю.

При запуске программы глобальные переменные создаются один раз и на всю жизнь программы. Разные функции могут изменять их, и изменения будут видны из любого места программы. Так, в примере выше функция `f1` изменила значение `g`, и другая функция прочитала именно это изменённое значение.

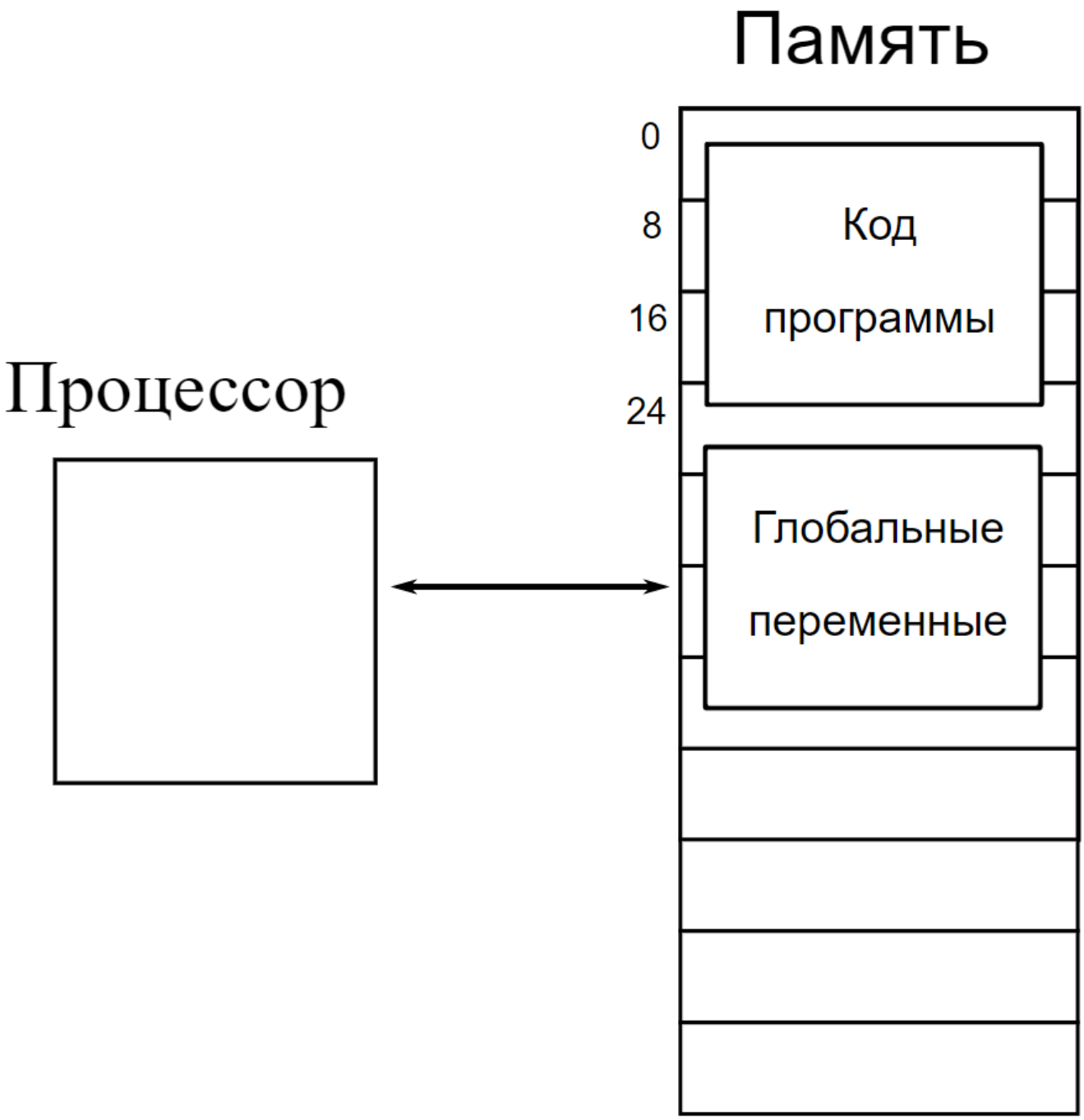
Тип переменной обозначен в исходном коде программы, но в памяти никак не указан, то есть во время выполнения программа не может определить тип данных, которые лежат в памяти по тому или иному адресу.

Тип влияет на выбор машинных инструкций, которые сгенерирует компилятор. Например, если мы читаем из памяти два числа типа `char` (в отличие от `int`, они занимают один байт) и складываем их, компилятор сгенерирует инструкции, читающие два числа размером по 1

байт и складывающие их. А для чисел типа int он выберет инструкции, читающие пачки из четырех байт и складывающие их.

Еще раз о присваивании. В конструкции `<var> = <expr>` сначала будет подсчитано значение выражения `<expr>`, затем это значение будет записано в переменную `<var>`. Обратите внимание: конструкции вроде `x+1 = 34` не имеют смысла: слева от знака равенства должно стоять "место в памяти", иначе куда мы будем записывать значение?

Картинка ниже показывает, в каком месте памяти находятся глобальные переменные.



Еще немного о printf

Строка, которую принимает `printf`, на самом деле является "шаблоном" для вывода. В ней расставлены *спецификаторы вывода*, такие, как `%d`. Каждому спецификатору вывода по порядку соответствует один из аргументов, которые передаются после строки. Вот несколько примеров:

```
printf("%d", 42);                    // "42"

printf("%d %d", 42, 44);              // "42 44"

printf("%d %d", 42 );                // ошибка, спецификаторов 2 но недостаточно аргументов

printf("Hello %d world %d", 42, 44); // "Hello 42 world 44"
```

Коды специальных символов, таких, как перевод строки, можно включать прямо в строку.

```
printf("%d\n%d\n%d", 42, 43, 44);

/* Выведет:

42

43

44

*/
```

Локальные переменные

С глобальными переменными связано много проблем, и в больших программах их практически никогда не используют. Сложно понять, какие именно глобальные переменные использует, скажем, функция `f`. Для этого нужно просмотреть код `f`, и код всех функций, которые вызывает `f`, и код тех функций, которые вызывают функции, вызванные `f`, и так далее. Кроме того, функции можно вызывать по их адресам, а не по именам, что дополнительно усложняет анализ.

Вместо глобальных переменных, существующих отдельно от функций, мы почти всегда используем *локальные* переменные, привязанные к конкретной запущенной функции. В коде они объявляются внутри тела функции:

```
void f() {

   int i = 42;

   printf("%d", i);

}
```

Во время выполнения программы каждый раз при вызове функции `f` программа зарезервирует ячейки памяти для всех её локальных переменных, в данном случае переменной `i`. По завершению функции `f` эти ячейки становятся вновь доступны для резервирования. Когда-нибудь в будущем в этих ячейках могут оказаться другие данные.

Изменим пример так, чтобы переменная `i` не инициализировалась значением 42. Что выведет на экран такая функция?

```
void f() {

   int i;

   printf("%d", i);

}
```

Неинициализированные локальные переменные функций хранят "мусорные" значения, поэтому такая программа выведет какое-то число, и мы заранее не знаем, какое. Может быть, 532123. Программа резервирует ячейки памяти, но в них уже лежали какие-то значения – ячейки памяти не бывают пустыми. Мы заранее ничего не знаем о том, какие это будут значения, поэтому использовать их в программе бессмысленно, даже для "генерации случайных значений". Никакой автоматической записи нулей в эти переменные не происходит.

Одна из наиболее распространённых ошибок при использовании локальных переменных -- когда программисты забывают их инициализировать и действуют как если бы в них хранились нули.

Можно сказать, что *время жизни* локальной переменной – от вызова её функции до возврата из этой функции. А время жизни глобальных переменных совпадает со временем выполнения самой программы.

Кстати, аргументы функций ведут себя как локальные переменные с той разницей, что значения аргументов в момент запуска функции явно определены.

```
void print_int(int var1) {

   printf("%d", var1);

}
```

```
...
```

```
/* При запуске print int в её "переменную" var1 запишется значение 42, затем её тело начнёт выполняться */

print_int(42);
```

Для следующих заданий мы предоставляем функцию `read_int`, которая считывает с клавиатуры целое число и возвращает его. Мы уже упоминали о ней в одном из прошлых уроков.

Напомним, что на Stepik система тестирования заданий автоматически подаёт текстовое представление чисел на вход так, что вызвав `read_int` можно "откусить" первое из этих чисел. При каждом следующем вызове `read_int` прочитает ещё одно число. Через несколько уроков мы объясним вам, как устроена эта функция.

Распространённая ошибка. Код ниже выглядит корректным но на самом деле не выполняет задачу "считать два числа, передать их в функцию `f`":

```
f( read_int() , read_int() )
```

В C порядок вычисления аргументов функции не определён. Это означает, что возможны две ситуации:

- с клавиатуры считывается второй аргумент функции `f`, затем первый; или
- с клавиатуры считывается первый аргумент функции `f`, затем второй.

Для нас важен порядок подсчёта аргументов, поэтому в этом случае мы считываем аргументы в отдельные локальные переменные по порядку:

```
int arg1 = read_int();

int arg2 = read_int();

f( arg1, arg2 );
```

В этом задании мы посчитаем и выведем количество корней квадратного уравнения.

Под квадратным уравнением мы имеем в виду уравнение ax^2+bx+c=0$ax^2+bx+c=0$, где a, b, c -- целые числа (в общем случае они не обязаны быть целыми), и a \neq 0$a \neq 0$. Чтобы понять, сколько корней у квадратного уравнения, мы считаем так называемый *дискриминант.* Дискриминант обозначается DD и равен D = b^2-4ac$D = b^2-4ac$.

В зависимости от значения DD возможны три ситуации:

- если дискриминант отрицательный, то корней у уравнения нет;
- если дискриминант равен нулю, то корень ровно один;
- если дискриминант положителен, то у уравнения два различные корня.

Вам необходимо определить три новые функции:

- Функция `discriminant` принимает числа a,b,ca, b, c через аргументы и возвращает значение дискриминанта, подсчитанное по формуле выше.
- Функция `root_count`, принимает числа a,b,ca, b, c через свои аргументы и возвращает количество корней квадратного уравнения ax^2 + bx + c = 0$ax^2+bx+c=0$. Она не должна печатать ничего на экран, только вернуть 0, 1 или 2.

Используйте локальную переменную чтобы не подсчитывать дискриминант больше одного раза.
- Функция `main` считывает числа a,b,ca, b, c от пользователя (см. предыдущий слайд), с помощью `root_count` считает количество корней и выводит его на экран.

Заполните пропуски в шаблоне в соответствии с комментариями.

Про ошибку control reaches the end of non-void function.

Обратите внимание: в C нет оператора "возведения в степень". Оператор ^ означает побитовое исключающее ИЛИ, поэтому запись `x^2` возможна, но означает не возведение в степень, а битовую операцию.

Sample Input:

```
6 11 -35
```

Sample Output:

```
2
```

1task – program

Как локальные переменные хранятся в памяти?

Мы уже детально рассмотрели процесс вызова функций и возврата из них. Мы знаем, что последняя вызванная функция первой завершает свою работу; а первая вызванная функция (это main) живёт дольше всех – вообще всё время работы программы.

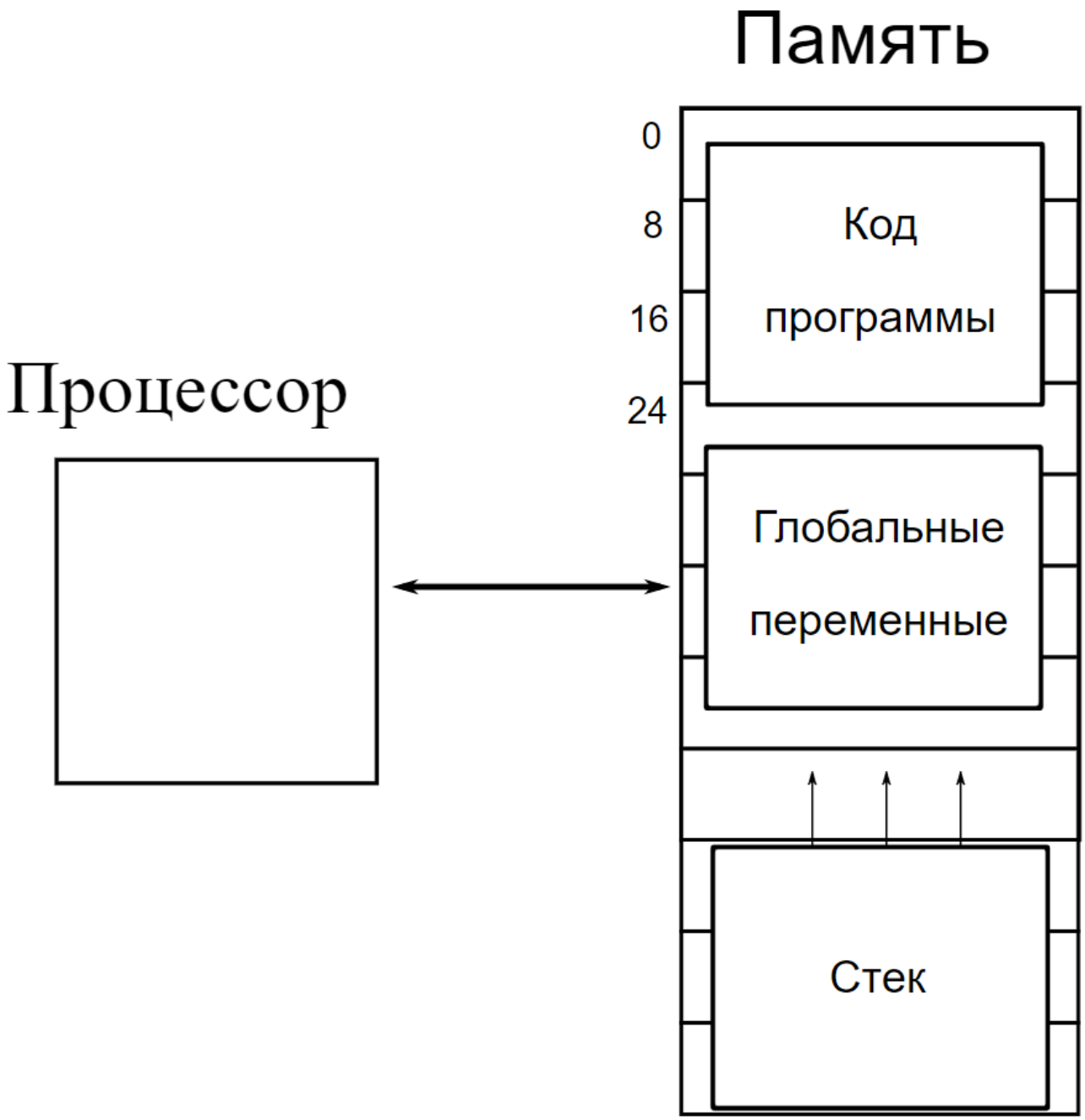
Существует структура данных, которая ведёт себя похожим образом. Она называется *стек* и часто сравнивается со стопкой тарелок: мы можем класть тарелку только сверху, и забирать только самую верхнюю тарелку. Собственно, только эти две операции стек и поддерживает: *push* (положить элемент на вершину стека) и *pop* (забрать элемент с вершины стека).

На компьютерах существует аппаратный стек – выделенная часть общей памяти, внутри которой выделена *вершина стека*. С помощью специальных машинных инструкций можно:

- записывать элемент на вершину стека, одновременно двигая её к нулевому адресу (*push*)
- считывать элемент с вершины стека, одновременно двигая её к максимальному адресу (pop)

При этом наш вариант аппаратного стека – это не отдельное устройство, работающее по своим законам. В настоящей "стековой" памяти нет возможности обратиться к каким-то элементам, кроме самой верхушки стека. В нашем же случае все элементы, лежащие в стеке, имеют адреса, и можно обратиться к элементу глубоко внутри стека если мы знаем его точный адрес.

Аппаратный стек используется для хранения служебной информации о функциях, в том числе и их локальных переменных.



В этой секции мы рассказали удобное приближение к той ситуации, которая наблюдается в реальных программах. Это не точное описание абстрактного вычислителя C и не точное описание реальности. Детали этих отличий мы узнаем в более поздних модулях.