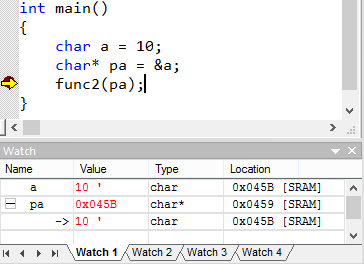
**Указатели, массивы и строки**

1. Известен ответ, приведенный пример не дает возможность изменить переменную – потому что модифицируется **копия** переменной. Как же подобраться к самой переменной, так сказать **оригиналу**? Для обращения к оригиналу, нужен адрес оригинала в памяти. Если уметь передавать и работать с этим адресом, то вопрос решен.
2. Пример, кода, когда переменная модифицируется

|  |
| --- |
| **void** func2(**char**\* pointer /\* pointer – по-английски указатель \*/)  {  // присвоим значение по адресу памяти, который содержит указатель  \*pointer = 200;  }  **int** main()  {  **char** a = 10; // выделяем память под переменную на стеке  **char**\* pa; // объявляем переменную-указатель  pa = &a; // записываем адрес переменной a в переменную-указатель  func2(pa); // передаем указатель в функцию  } |

1. Зачем это вообще нужно, если можно просто сделать не void-функцию и возвращать результат обычным способом. Ответ в виде вопроса – а что делать, если функция должна возвращать несколько результатов, например, функция поиска минимума и максимума массива.
   1. Можно сделать отступление о typedef struct. Например, функция возвращает две координаты на дисплее. Номер точка на траектории превращается в две координаты.
2. Пояснения работы кода
   1. Если посмотреть окно Watch, можно увидеть адрес переменной a в памяти (Location = 0x045B, 1-я строка на рис.).
   2. После присвоения pa = &a переменная pa будет содержать **адрес** переменной **a** (Value для **pa** совпадает с Location для **a**, 2-я строка на рис.). При этом отладчик учитывает, что переменная pa является указателем и позволяет посмотреть, что лежит по адресу, на который она указывает (3-я строка на рис.).
   3. Рисунок. Отладчик показывает состояние кода непосредственно перед выполнением func2.



* 1. Внутри функции func2 переменная pointer будет содержать адрес переменной pa.
  2. Для того, чтобы обратиться к ячейке памяти, на которую указывается адрес в pointer, используется *унарный* оператор **разыменования** «\*», не путать с *бинарным* оператором «\*», который означает умножение.
     1. Унарный оператор работает с одним операндом \*pointer (разыменовали операнд pointer).
     2. Бинарный оператор работает с двумя операторами a \* b (умножили операнд a на операнд b).
  3. Разыменованный указатель \*pointer следует рассматривать как содержимое адреса, на который указывает указатель. Выражение \*pointer = 200 означает «присвоить значение 200 ячейке памяти, на которую указывает адрес переменной pointer». В нашем примере содержимое адреса – **оригинал** переменной **a**.
  4. Вызов функции достаточно громоздкий. Разумеется, в реальном коде переменная pa не нужна, адрес &a можно сразу передать в func2:

**void** func2(**char**\* pointer) {

\*pointer = 200;

}

**int** main() {

**char** a = 10;

func2(&a);

}

* 1. Вопрос на понимание. Механизм передачи параметра через стек, при котором создается копия, все равно используется. Вопрос, копия **чего** кладется на стек в этом случае? Указатель точно так же кладется на стек. Т.е. pointer есть копия **pa**.

1. **Роль типа указателя**
   1. Как изменится код, если возникнет задача модифицировать int, а не char.
   2. По аналогии напрашивается заменить char\* на int\*

**void** func3(**int**\* pointer) {

\*pointer = 2000;

}

**int** main() {

**int** a = 10;

**int**\* pa = &a;

func3(pa);

}

* 1. Тогда немедленно возникает вопрос, в чем разница между char\* и int\*?
  2. Обратите внимание, что у самой переменной **pa** тоже есть собственный адрес. Это такая же обычная переменная
  3. Что в целом можно записать в pa – должен «влезать» весь диапазон адресов оперативной памяти микропроцессора. Объем ОЗУ в ATMega16 равен 1 килобайт или 1024 байта. Соответственно, в адрес может лежать в диапазоне 0-1023 или 0x000-0x45F. Сколько байт нужно, чтобы описать этот диапазон? Одного байта мало, двух должно быть достаточно. Проверим, sizeof(char\*) равно 2.
  4. Зависит ли размерность указателя от того, на адрес каким содержимым он указывает.
  5. Будет ли отличаться sizeof(int\*) и sizeof(char\*)? Нет.
  6. В какой момент проявится отличие int\* и char\*?

**int** main()

{

**unsigned** **int** a = 0xAABB;

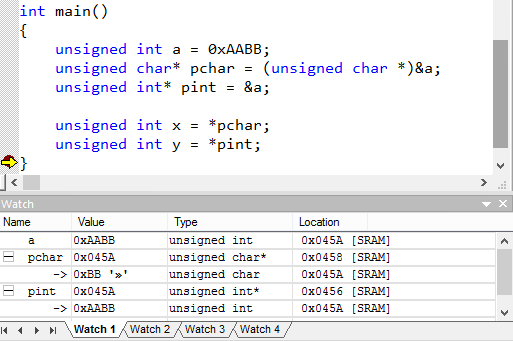
**unsigned** **char**\* pchar = (**unsigned** **char** \*)&a;

**unsigned** **int**\* pint = &a;

**unsigned** **int** x = \*pchar;

**unsigned** **int** y = \*pint;

}

* 1. 
  2. Чтобы объяснить пример, нужна карта памяти.
  3. Итак, типизированность указателя проявляется при его разыменовании. Поскольку кроме адреса, который указывает на содержимое, необходимо задать правила интерпретации битового набора этого содержимого, чтобы получить значения.

1. **Как использовать указатели с массивами. Как передать массив (в частности, строку) в функцию.**
   1. Не получится передать массив целиком, поскольку он переменной длины. Представим себе, что мы можем ограничиться передачей только указателя на первый элемент. А дальше, если по указателю на нулевой элемент посчитать указатель на следующий элемент и так далее, то указателя на нулевой элемент достаточно.

**int** array\_min(**int**\* p, **int** size) {

// что означает эта операция применительно к указателю p?

**int** result = p[0];

**for** (**int** index = 1; index < size; ++index) {

// что означает эта операция применительно к указателю p?

**int** value = p[index];

**if** (value < result)

result = p[index];

}

**return** result;

}

**int** main() {

**int** values[] = { 6, 3, 4, 5 };

**int** min = array\_min(&values[0], 4);

}

* 1. Использование указателя на массив int\* не отличается от использования самого массива int[]. Другими словами, если пошагово в отладчике выполнить программу, то станет ясно, что p[0] = 6, p[1] = 3 и т.д.
  2. Отметим, что указатель на нулевой элемент можно взять более удобно, вместо &values[0] можно написать просто values. Это еще один из немногих кусочков синтаксического сахара в язык C.

**int** min = array\_min(values, 4);

* 1. Кроме того, можно вместо указателя int\* для p указать тип массив int[]:

**int** array\_min(**int** p[], **int** size)

Разницы с точки зрения использования p в коде не будет.

* 1. Указатель дает возможность по начальному элементу пройти к следующим элементам массив. Однако информация о количестве этих элементов, т.е. длине массива в указатели отсутствует.
     1. Поэтому длину массива приходится передавать дополнительно (параметр size).
     2. Альтернативный способ передачи информации о длине массива состоит в добавлении в конец строки элемента, который заведомо больше нигде в массиве встретиться не может, так называемого **сентинела**. Этот способ используется в строках языка C, в которых сентинелом является символ с кодом 0x00.

1. **Адресная арифметика**.
   1. Рассмотрим более подробно оператор «квадратные скобки» [ ] применительно к указателю, поскольку имеются нюансы.
   2. Забудем на время об операторе []. Как в этом случае решить задачу перехода к соседнему элементу по указателю?

**char** values[] = { 10, 20, 30, 40 };

**char**\* p = values;

p = p + 1;

**char** value = \*p; // чему здесь равно value?

* 1. Адрес в p на единицу увеличился, т.е. p указывает не на нулевой, а на первый элемент массива values. Тогда разыменование \*p дает values[1], равный 20. Обратим внимание, что несмотря на то, что сам указатель p = p + 1 модифицируется, это никак не влияет на массив.
  2. Следующий пример. Изменим размерность элементов массива.

**int** values[] = { 10, 20, 30, 40 };

**int**\* p = values;

p = p + 1; // насколько должен измениться адрес p, чтобы указывать на 20?

**int** value = \*p;

Адресная арифметика учитывает размерность данных. Карта памяти для массивов. При выполнении операции p = p + 1 адрес в p увеличивается на столько, сколько байт занимает тип данных, на который указывает p. Для char это 1 (байт), а для int это 2, для long и float это 4.

* 1. Карта памяти для двух массивов.
  2. Еще пример

**char** values[] = { 10, 20, 30, 40 };

**char**\* p = values;

**char** value1 = \*p + 3; // чему равно value1?

**char** value2 = \*(p + 3); // чему равно value2?

Чему равно value1, value2, p и массив values в конце программы?

* 1. Теперь можно очень легко сформулировать, что такое оператор квадратные скобки. Для указателя p это еще один способ разыменования с целым смещением k:

p[*k*] = \*(p + *k*)

1. **Работа со строками**

Поиск количества вхождений заданного символа в строке.

sprintf()

1. **Пока не размещено.**
   1. Преобразование типов (int \*). Можно пример как обращаться к unsigned long как к строке. 0x00303132
   2. Функция malloc.
   3. Нетипизированный указатель. Нельзя разыменовать.
   4. Пример 2. Поиск длины строки. Или поиск вхождений символа в строке.
   5. Рассмотрим два массива int values\_int[4] и char values\_char[4]. sizeof(values\_int) = 8, sizeof(values\_char)=4.

**int** main()

{

**int** values\_int[] = { 10, 20, 30, 40 };

**char** values\_char[] = { 1, 2, 3, 4 };

**int**\* pi1 = values\_int; // чему здесь равно pi1?

**int**\* pi2 = pi1 + 1; // насколько отличаются pi1 и pi2?

**char**\* pс1 = values\_char; // чему здесь равно pc1?

**char**\* pс2 = pс1 + 1; // насколько отличаются pc1 и pc2?

}

* 1. **int** min = array\_min(values, **sizeof**(values)/**sizeof**(values[0]));

1. Итог, зачем нужны указатели
   1. Возможность модификации переданных значений
   2. Передача большого объема данных (массив, строка, структура) в функцию. Экономится место на стеке.