1.9. Указатели

Friday, June 5, 2020 14:53

В данном уроке мы разберем один из наиболее сложных элементов языка C – указатели. Исключительная гибкость языка C при работе с указателями позволяет нам эффективно создавать сложные структуры данных, изменять аргументы передаваемые в функции, работать с динамически выделяемой памятью, а также более эффективно работать с массивами.

Так как указатели будут встречаться на практике очень часто, я постараюсь максимально широко покрыть работу с указателями:

- Объявление простых указателей
- Использование указателей в выражениях
- Создание указателей на структуры, массивы и функции
- Использование указателей для создания связных списков (*linked lists*)
- Применение к указателям ключевого слова `const`
- Передача указателей в качестве аргумента функции

Указатели и понятие косвенности (*pointers and indirection*)

Чтобы понять как работают указатели, сначала нужно понимать, что такое **косвенность**. Возьмем пример из жизни. Допустим, мы работаем в какой-то компании или учреждении и нам нужен какой-то инвентарь или оборудование. Сами мы его купить не можем, потому что нужно обращаться в отдел инвентаризации, который по нашему запросу закупит нужные нам вещи. То есть, мы не сами покупаем вещи напрямую, а получаем их **косвенно**, отправляя запрос на них.

Суть работы указателей примерно такая же. Указатели создают возможность **косвенно** получать доступ к данным.

Объявление указателя

Так же, как мы объявляем обычные переменные, например:

```
int  count = 10;
```

можно объявить переменную `int_pointer`, которую можно использовать для косвенного доступа к значению переменной count:

```
int  *int_pointer;
```

Звездочка указывает компилятору на то, что переменная `int_pointer` – указатель на тип `int`. Это означает, что `int_pointer` будет использован в программе для косвенного доступа к одному или нескольким значениями типа `int`.

Вспомним также про оператор `&`, который мы постоянно использовали при вызове `scanf()`. Он называется **адресным оператором** и используется для создания указателя на объект. Так, если `x` – переменная (любого типа), то `&x` – указатель на эту переменную. Такое выражение можно присвоить любой переменной-указателю с тем же типом данных, что и переменнная `x`.

Таким образом, выражение

```
int_pointer = &count;
```

создает косвенную ссылку между `int_pointer` и `count`. Благодаря адресному оператору, указателю `int_pointer` присваивается **не значение** переменной **`count`**, **а указатель** на эту переменную.

Для чтения значения переменной `count` через указатель `int_pointer` используется **косвенный оператор** – звездочка (`*`). Так, если `x` – переменная типа `int`, то выражение

```
x = *int_pointer;
```

присваивает значение переменной `x`, полученное **по косвенной ссылке** через `int_pointer`. Так как до этого мы установили указатель `int_pointer` на переменную count, то в `x` будет записано значение переменной `count`.

Напишем программу, иллюстрирующую работу указателей.

```
#include <stdio.h>

int main (void)
{
     int   count = 10, x;
     int   *int_pointer;

     int_pointer = &count;
     x = *int_pointer;

     printf ("count = %i, x = %i\n", count, x);

     return 0;
}
```

Программа выдаст:

```
count = 10, x = 10
```

Ничто не мешает объявлять числа и указатель одной строкой, в программе они разбиты на две для лучшей читаемости. Объявление

```
int   count = 10, x, *int_pointer;
```

вполне допустимо.

Далее программа создает указатель на переменную `count` и через этот указатель присваивает переменной `x` значение `count`.
Да, этот пример абсолютно бесполезный на практике и служит только иллюстрацией использования указателей, адресных и косвенных операторов. До практического применения дойдем чуть позже.

А пока напишем еще одну программу, но теперь будем ссылаться на символ, а не на число.
И покажем еще несколько приемов работы с указателями.

```
#include <stdio.h>

int main (void)
{
     char  c = 'Q';
     char  *char_pointer = &c;

     printf ("%c %c\n", c, *char_pointer);

     c = '/';
     printf ("%c %c\n", c, *char_pointer);

     *char_pointer = '(';
     printf ("%c %c\n", c, *char_pointer);

     return 0;
}
```

Программа выдаст:

```
Q Q
/ /
( (
```

Здесь во втором примере показано, что при изменении значения переменной `c` также изменится значение, возвращаемое указателем на эту переменную.
А в третьем примере показано, что при помощи **адресного оператора** `&` можно изменить значение переменной `c`, обратившись к ней через указатель. Обратите внимание, что так можно делать только если `c` была объявлена **до написания** этого выражения.

Также помните о правильном синтаксисе при присваивании начальных значений указателю.

Вот так правильно:

```
    char_pointer = &c;
```

А вот так программа вылетит с ошибкой, потому что указатель изначально указывает неизвестно куда, и получается, что мы пытаемся записать значение `&c` в совершенно рандомное место:

```
    *char_pointer = &c;
```

Всегда помните о том, что пока указателю не присвоено какое-то значение, им нельзя пользоваться.

На этом мы заканчиваем рассматривать основы работы с указателями. Без понимания базовых принципов их работы мало смысла двигаться дальше, поэтому повторно проработайте непонятные места.

Использование указателей в выражениях

В программе ниже заданы два указателя: `p1` и `p2`. Обратите внимание, как значение, на которое указывает указатель используется в арифметическом выражении.

```c
// More on pointers

#include <stdio.h>

int main (void)
{
     int  i1, i2;
     int  *p1, *p2;

     i1 = 5;
     p1 = &i1;
     i2 = *p1 / 2 + 10;
     p2 = p1;

     printf ("i1 = %i, i2 = %i, *p1 = %i, *p2 = %i\n", i1, i2, *p1, *p2);

     return 0;
}
```

Программа выдаст:

```
i1 = 5, i2 = 12, *p1 = 5, *p2 = 5
```

В программе создается указатель `p1`, указывающий на значение `i1`, затем в формуле происходит обращение к `i1` через этот указатель. После этого указателю `p2` присваивается значение `p1` и теперь они оба указывают на переменную `i1`.

Указатели и структуры

Мы уже умеем создавать указатели на простые типы данных вроде `int` или `char`. Сейчас мы рассмотрим, как создавать указатели на структуры. Зададим структуру `date`, как мы это делали в уроке 1.7 (*структуры данных*):

```
struct date
{
     int  month;
     int  day;
     int  year;
};
```

Точно так же, как объявляются переменные типа struct date:

```
struct date   todaysDate;
```

мы можем объявить указатель на структуру типа struct date:

```
struct date  *datePtr;
```

Переменная `datePtr` может использоваться точно так же, как мы использовали переменные указателей в предыдущих программах. Например, можно сделать, чтоб она указывала на `todaysDate`:

```
datePtr = &todaysDate;
```

После этой операции мы можем косвенно обращаться к любым элементам этой структуры:

```
(*datePtr).day = 21;
```

Выражение задает элементу `day` значение 21. Скобки здесь обязательны, потому что оператор элемента структуры (.) имеет **более высокий приоритет**, чем косвенный оператор (*).

Для проверки значения `month`, хранимого в структуре типа `date`, можно использовать выражение

```
if  ( (*datePtr).month == 12  )
                ...
```

Указатели на структуры настолько часто встречаются в программах на языке С, что был создан специальный указатель на структуру (`->`). Таким образом, выражение

```
(*x).y
```

можно более явно записать как

```
x->y
```

То есть, предыдущее выражение с `if` можно переписать как

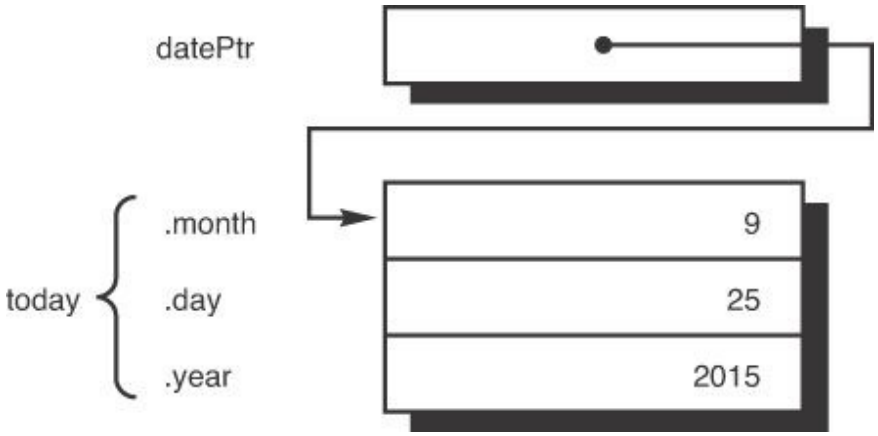
```
if  ( datePtr->month == 12 )
            ...
```

Перепишем программу из урока 1.7 (структуры данных) с использованием указателей.

```
1    //  Program to illustrate structure pointers
2
3    #include <stdio.h>
4
5    int main (void)
6    {
```

```
7           struct date
8           {
9                int  month;
10               int  day;
11               int  year;
12          };
13
14          struct date  today, *datePtr;
15
16          datePtr = &today;
17
18          datePtr->month = 9;
19          datePtr->day = 25;
20          datePtr->year = 2015;
21
22          printf ("Today's date is %i/%i/%.2i.\n",
23                    datePtr->month, datePtr->day, datePtr->year % 100);
24
25          return 0;
26      }
```

На рисунке ниже показано, как будут выглядеть переменные `today` и `datePtr` после операций присваивания в программе выше.



Опять же, никакого практического смысла в использовании указателей здесь нет и без них можно легко обойтись. До практического применения скоро дойдем.

Структуры из указателей

Указателю ничто не мешает быть частью структуры:

```
struct  intPtrs
{
     int  *p1;
     int  *p2;
};
```

Здесь мы создали структуру inPtrs, состоящую из двух указателей, – `p1` и `p2`. Переменные типа `struct intPtrs` создаются как обычно:

```
struct intPtrs  pointers;
```

Переменную `pointers` также можно использовать как обычно, главное не забывать, что это переменная, хранящая **структуру** из двух указателей, а не просто указатель.

Напишем программу, иллюстрирующую работу со структурами из указателей.

```c
// Function to use structures containing pointers

#include <stdio.h>

int main (void)
{
     struct  intPtrs
     {
          int  *p1;
          int  *p2;
     };

     struct intPtrs  pointers;
     int  i1 = 100, i2;

     pointers.p1 = &i1;
     pointers.p2 = &i2;
     *pointers.p2 = -97;

     printf ("i1 = %i, *pointers.p1 = %i\n", i1, *pointers.p1);
     printf ("i2 = %i, *pointers.p2 = %i\n", i2, *pointers.p2);
     return 0;
}
```

Программа выдаст:

```
i1 = 100, *pointers.p1 = 100
i2 = -97, *pointers.p2 = -97
```

После объявления переменных, выражение

```
pointers.p1 = &i1;
```

устанавливает элемент `p1` структуры `pointers` на переменную `i1`.

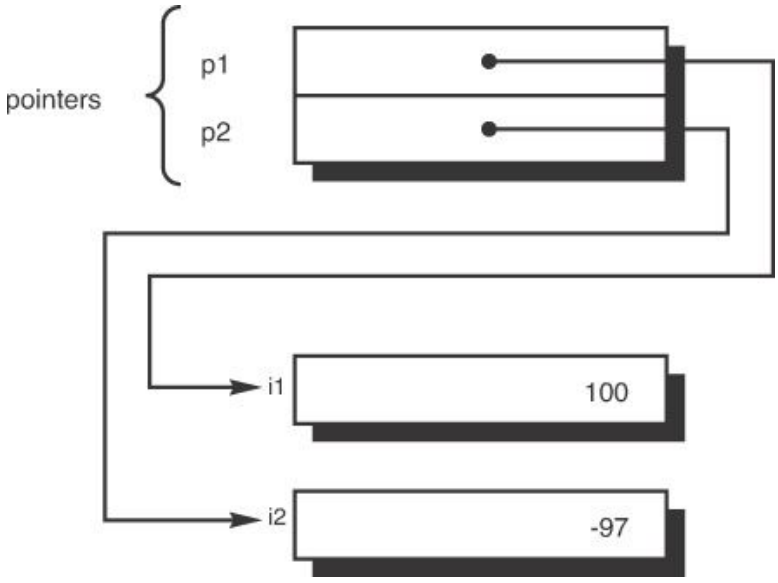
А выражение

```
pointers.p2 = &i2;
```

элемент `p2` на переменную `i2`.

После этого программа через указатель `p2` обращается к переменной `i2` и задает ей значение -97.

То есть, перед завершением программы ее переменные будут выглядеть вот так:



Указатель `pointers.p1` указывает на переменную `i1`, которая хранит значение 100, а `pointers.p2` указывает на переменную `i2 = -97`.

Связные списки (*linked lists*)

Указатели на структуры и структуры из указателей – очень мощные инструменты в языке C и с их помощью можно создавать сложные структуры данных, такие как одинарные и двойные **связные списки** (*linked lists / double linked lists*), а также **деревья** (*tree*).

Допустим, у нас есть вот такая структура под именем `entry`:

```
struct entry
{
     int            value;
     struct entry   *next;
};
```

Первый элемент структуры – обычное число `int`. Второй элемент – указатель на тип `struct entry`. Да, это допустимо, чтоб в структуре был указатель на саму структуру. Теперь зададим две переменных типа `struct entry`:

```
struct entry  n1, n2;
```

И сделаем, чтоб указатель из `n1` указывал на структуру `n2`:

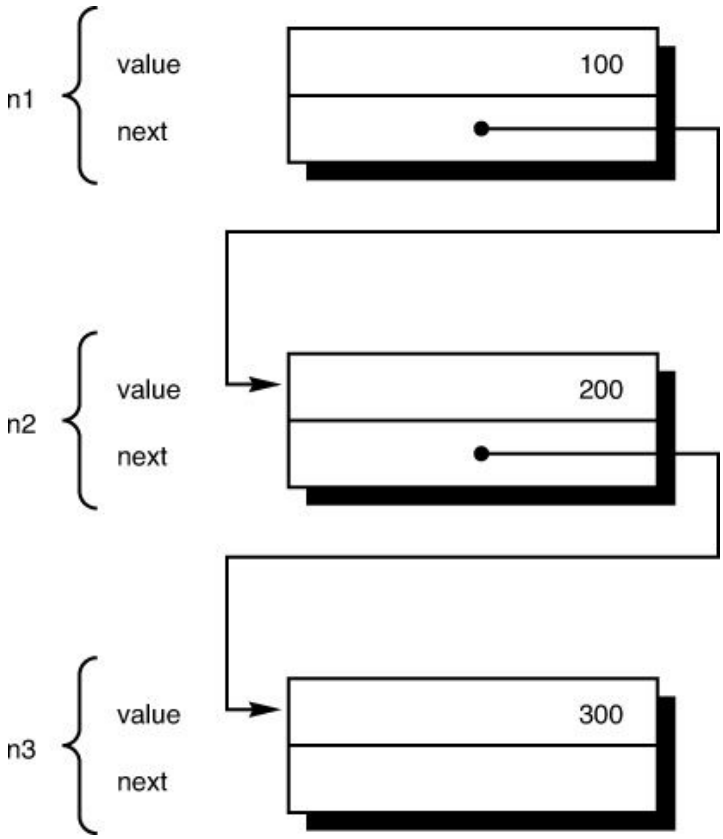
```
n1.next = &n2;
```

Такая операция создаст связь между `n1` и `n2`.

Если у нас будет третья переменная `n3`, тоже типа `struct entry`, можно создать еще одну связь:

```
n2.next = &n3;
```

Получившаяся цепочка связанных структур называется **связным списком**, который будет выглядеть следующим образом:



Напишем программу, иллюстрирующую работу связных списков.

```c
// Function to use linked lists

#include <stdio.h>

int main (void)
{
     struct  entry
     {
          int           value;
          struct entry  *next;
     };

     struct entry n1, n2, n3;
     int          i;

     n1.value = 100;
     n2.value = 200;
     n3.value = 300;

     n1.next = &n2;
     n2.next = &n3;

     i = n1.next->value;
     printf ("%i  ", i);

     printf ("%i\n", n2.next->value);

     return 0;
}
```

Программа выдаст:

```
200  300
```

Здесь мы создали структуры `n1`, `n2` и `n3` типа struct `entry`, который содержит число `value` и указатель на структуру `entry` – `next`.

Затем программа присваивает значения 100, 200 и 300 элементам `value` в структурах `n1`, `n2` и `n3` соответственно. Далее выражения

```c
n1.next = &n2;
n2.next = &n3;
```

создают **связный список**, где указатель `next` из структуры `n1` указывает на структуру `n2`, а в `n2` `next` указывает на `n3`.

Выражение

```c
i = (n1.next)->value;
```

присваивает переменной `i` значение элемента `value` той структуры, на которую указывает `n1.next`, то есть `n2`. Таким образом, `i` будет присвоено `n2.value = 200`.

Оператор структурного элемента (`.`) имеет тот же приоритет, что и указатель на структуру (`->`), поэтому скобки в выражении выше не нужны.

```c
printf ("%i\n", n2.next->value);
```

выводит на экран значение элемента `value` той структуры, на которую указывает `n2.next`, то есть `n3`.

Связные списки также упрощают **добавление и удаление** элементов в больших списках. Например, в предыдущей программе можно легко удалить из списка элемент `n2` просто приравняв значение указателя `n1.next` к указателю `n2.next`:

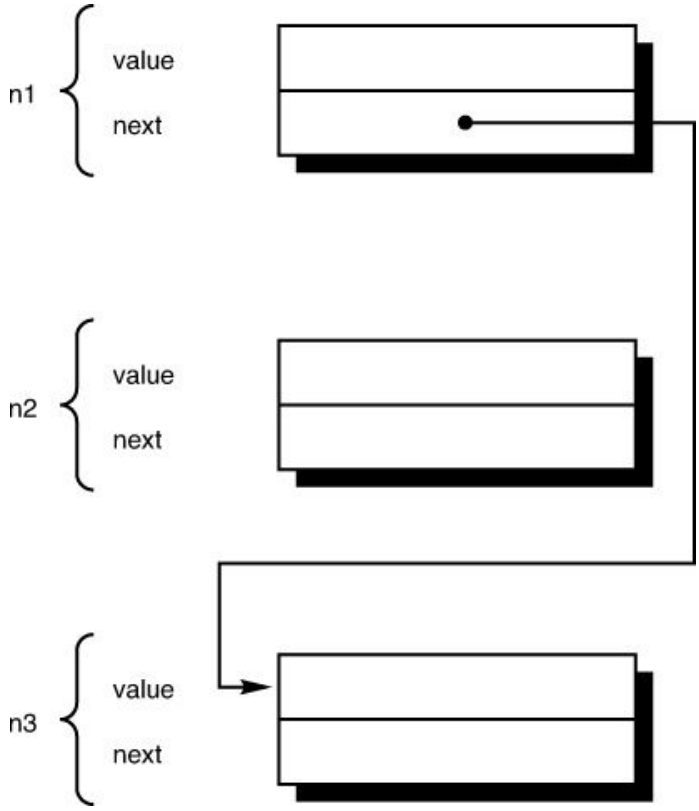
```
n1.next = n2.next;
```

Так как `n2.next` указывал на `n3`, `n1.next` теперь тоже на него указывает, а на `n2` уже указать некому и `n2` перестает существовать в связном списке. Еще это можно было сделать вот так:

```
n1.next = &n3;
```

но это часто не так удобно, потому что нужно точно знать, куда указывает `n2`. При работе с большими списками тяжело понять, куда будет указывать каждый указатель.

Продемонстрируем удаление элемента более наглядно, картинкой.

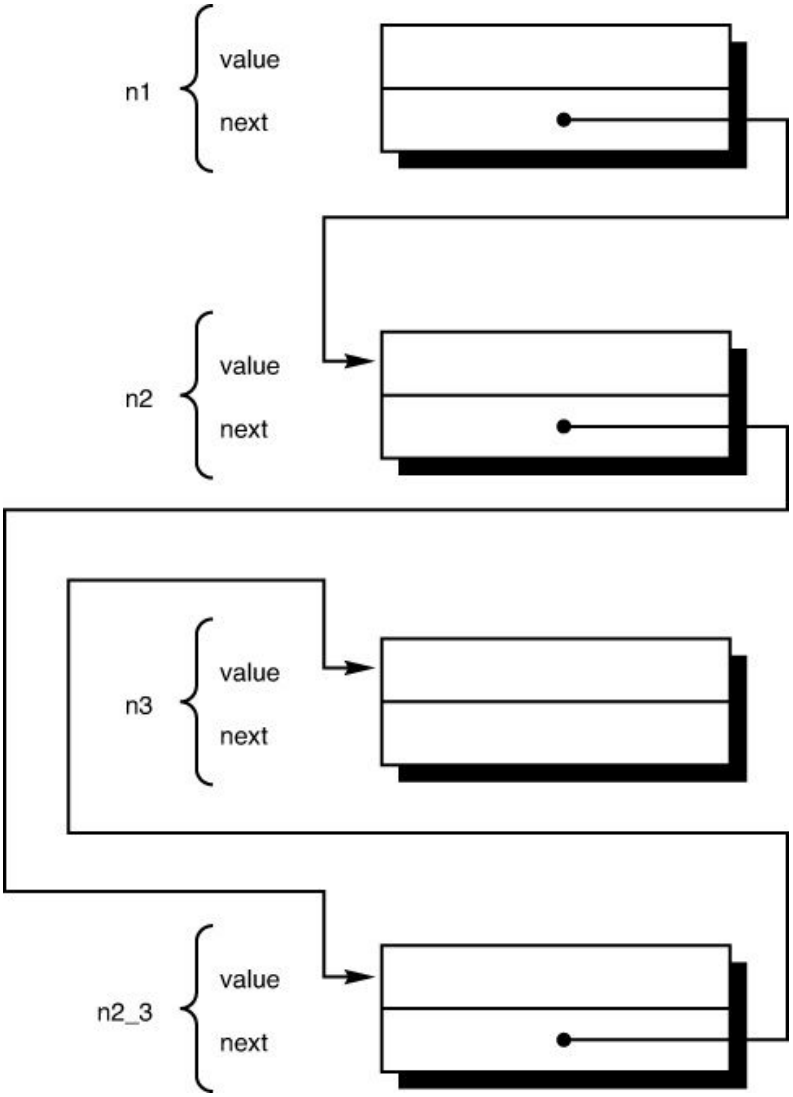


Удаление элемента из связного списка

Добавление элемента в список тоже простое и интуитивно понятное. Допустим, мы хотим между `n2` и `n3` вставить элемент `n2_3`. Для этого нужно установить указатель в `n2_3` туда, куда указывал `n2.next`, а потом задать указатель `n2.next` на `n2_3`. Это будет выглядеть вот так:

```
n2_3.next = n2.next;
n2.next = &n2_3;
```

Обратите внимание, что действия должны выполняться именно в таком порядке, потому что если сначала выполнить вторую операцию, то мы потеряем информацию о том, куда указывал `n2.next` изначально. Снова нарисуем картинку, как это выглядит.



Добавление элемента в связный список

Тут `n2_3` специально не нарисован между `n2` и `n3` чтобы показать, что `n2_3` может находиться где угодно в памяти и ему не обязательно физически находиться между `n2` и `n3`.

Перед тем, как мы начнем работать со связными списками, надо обговорить еще один момент – необходимость задания **начала и конца связного списка**.

Указатель на начало списка полезен для последовательного перебора элементов, что мы скоро увидим на примере. Приведем пример указателя на начало списка:

```
struct entry *list_pointer = &n1;
```

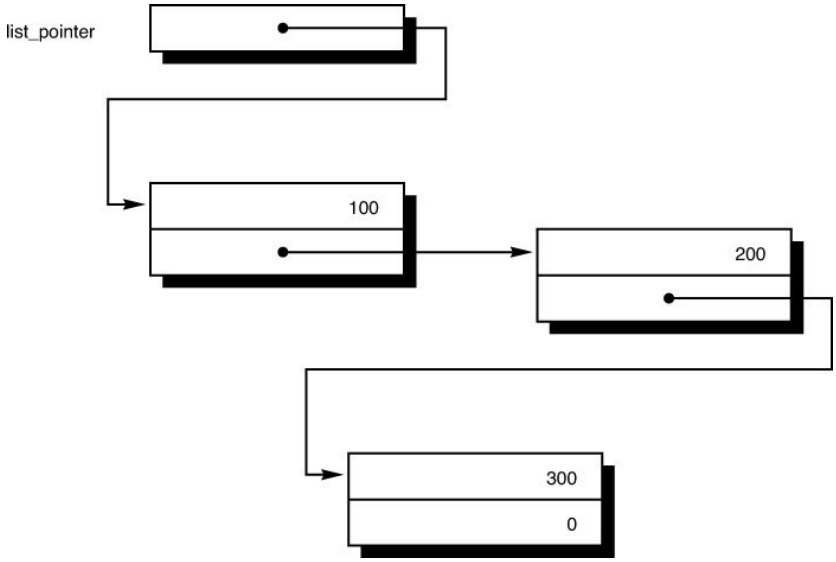
Теперь рассмотрим обозначение конца списка. Это нужно, например, для определения, что цикл дошел до конца списка. Обычно конец списка обозначается через **нулевой указатель** (*null pointer*).
Это указатель, значение которого равно 0.
прим.: нулевой указатель не всегда может после компиляции иметь нулевое значение. Но если мы попытаемся присвоить указателю нулевое значение, компилятор поймет, что мы хотим сделать нулевой указатель, но может сделать его значение отличным от нуля. То же самое происходит при сравнении указателя с нулем.

В нашем списке из трех элементов нулевой указатель можно хранить в переменной `n3.next`:

```
n3.next =  (struct entry *) 0;
```

На самом деле, так не пишут и в следующих уроках мы рассмотрим нормальный вариант записи этого выражения. **Оператор приведения типа** `(struct entry *)` здесь не обязателен, но упрощает чтение

кода. На рисунке ниже показан связный список в структуре `struct entry` с указателем `list_pointer`, указывающим на начало списка, а поле `n3.next` содержит нулевой указатель.



Связный список с начальным и конечным (нулевым) указателями

Напишем программу с циклом, который будет последовательно перебирать элементы связного списка и завершится при достижении нулевого указателя.

```c
// Program to traverse a linked list

#include <stdio.h>

int main (void)
{
     struct entry
     {
          int            value;
          struct entry   *next;
     };

     struct entry   n1, n2, n3;
     struct entry   *list_pointer = &n1;

     n1.value = 100;
     n1.next  = &n2;

     n2.value = 200;
     n2.next  = &n3;

     n3.value = 300;
     n3.next  = (struct entry *) 0;    // Mark list end with null pointer

     while ( list_pointer != (struct entry *) 0 ) {
          printf ("%i\n", list_pointer->value);
          list_pointer = list_pointer->next;
     }

     return 0;
}
```

Программа выдаст:

```
100
200
300
```

В программе мы задаем переменные `n1, n2, n3` и переменную-указатель `list_pointer`, которая изначально указывает на `n1`, – первый элемент связного списка. Далее мы при помощи указателей связываем между собой другие элементы связного списка и при помощи *type cast operator* задаем в элементе `n3` **нулевой указатель**, обозначающий окончание связного списка.

Цикл `while` выводит на экран значения элементов `value` созданного нами списка, последовательно переходя по заданным указателям и завершается по достижению нулевого указателя. Такой цикл может применяться для связных списков любых размеров, главное чтоб в этом списке последний элемент имел нулевой указатель.

При работе со связными списками на практике обычно не нужно вручную поэлементно создавать связный список, как мы это сделали в программе выше. Вместо этого программисты запрашивают у системы выделить память под каждый новый элемент списка и связывают новый элемент с уже существующим списком. Этот способ называется *динамическим выделением памяти*, который мы рассмотрим в будущих уроках.

Советуем внимательно пройтись по циклу `while` и расписать его работу на бумаге. Понимание принципа работы этого цикла является ключевым для понимания работы указателей в языке С.

Указатели и ключевое слово `const`

Мы уже знаем, что если мы объявим переменную как `const`, то это укажет компилятору на то, что переменная не будет меняться по ходу выполнения программы. В случае работы с указателями, нужно учитывать две вещи: будет ли меняться указатель и будет ли меняться значение, на которое он указывает. Допустим, у нас есть переменная и указатель на нее:

```
char c = 'X';
char *charPtr = &c;
```

Если charPtr всегда должен указывать на c, его можно объявить как **указатель-константу**:

```
char * const charPtr = &c;
```

После этого будет **нельзя** присваивать этому указателю новые значения. Выражение вроде

```
charPtr = &d;    // not valid
```

является недопустимым и компилятор выдаст предупреждение о попытке присвоения значения константе.

Если же нужно сделать **указатель на константу**, это делается вот так:

```
const char * const *charPtr = &c;
```

Так как `const` находится в начале строки, это указывает компилятору на то, что переменная, на которую указывает указатель, **не будет изменяться**. Второй раз `const` пишется уже для указания, что сам указатель не будет изменяться.

прим.: `const` обычно не используют в каждом удобном случае, только если это очень нужно. Выражения выше довольно трудно читать, лучше по возможности избегать их использования.

Указатели и функции

Указатели можно передавать в функции в качестве аргумента, как и любые другие переменные. Также можно создать функцию, возвращающую указатель.

Передача указателя в функцию довольно проста и ничем не отличается от передачи других переменных. Так, передача указателя list_pointer в функцию print_list будет выглядеть как:

```
print_list (list_pointer);
```

Очевидно, при объявлении функции нужно указать тип указателя:

```
void  print_list  (struct entry  *pointer)
```

```
{
        ...
}
```

Следует помнить, что значение указателя, передаваемого в функцию, **копируется** в формальный параметр при вызове функции. Поэтому функция в процессе своей работы **не изменяет** переданный ей указатель. Но есть один нюанс: несмотря на то, что функция не может изменить сам указатель, данные на которые указывает указатель, **могут быть изменены** внутри функции.

Программа ниже это иллюстрирует.

```
1   // Program to illustrate using pointers and functions
2
3   #include <stdio.h>
4
5   void test (int  *int_pointer)
6   {
7       *int_pointer = 100;
8   }
9
10  int main (void)
11  {
12      void test (int  *int_pointer);
13      int  i = 50, *p = &i;
14
15      printf ("Before the call to test i = %i\n", i);
16
17      test (p);
18      printf ("After the call to test i = %i\n", i);
19
20      return 0;
21  }
```

Программа выдаст:

```
Before the call to test i = 50
After the call to test i = 100
```

В качестве аргумента функция `test()` принимает указатель на тип `int`. В самой функции происходит присваивание значение `100` той переменной, на которую указывает полученный на вход указатель.

В `main()` мы задаем переменную `i = 50` и указывающий на нее указатель `p`. Затем передаем этот указатель в функцию `test()`, которая через него меняет значение переменной `i` на `100`.

Теперь давайте рассмотрим следующую программу:

```
1   // More on pointers and functions
2
3   #include <stdio.h>
4
5   void  exchange (int * const pint1, int * const pint2)
6   {
7       int  temp;
8
9       temp = *pint1;
10      *pint1 = *pint2;
11      *pint2 = temp;
12  }
13
14  int main (void)
15  {
16      void  exchange (int * const pint1, int * const pint2);
17      int   i1 = -5, i2 = 66, *p1 = &i1, *p2 = &i2;
```

```
18
19          printf ("i1 = %i, i2 = %i\n", i1, i2);
20
21          exchange (p1, p2);
22          printf ("i1 = %i, i2 = %i\n", i1, i2);
23
24          exchange (&i1, &i2);
25          printf ("i1 = %i, i2 = %i\n", i1, i2);
26
27          return 0;
28      }
```

Программа выдаст:

```
i1 = -5, i2 = 66
i1 = 66, i2 = -5
i1 = -5, i2 = 66
```

Здесь функция `exchange()` меняет между собой значения двух переменных, на которые указывают переданные ей указатели. Заголовок функции

```
void  exchange (int * const pint1, int * const pint2)
```

говорит о том, что функция `exchange()` будет принимать на вход два указателя на `int` и что переданные указатели не будут меняться в ходе работы функции (`const`).

Обратите внимание, что не обязательно создавать переменные-указатели только для того чтоб их потом передать в функцию; можно на лету создавать указатели при помощи **адресного оператора** (`&`). Что мы и делаем в 24-й строке:

```
exchange (&i1, &i2);
```

Данная программа показывает очень важное применение указателей. Вспомните, что функции могут возвращать только **одну** переменную и поэтому **без применения указателей такую функцию реализовать невозможно**, потому что нужно возвращать **две** переменные.

Программа ниже показывает, как функция может возвращать указатель. В программе задается функция `findEntry()`, которая ищет в связном списке заданное пользователем значение. Когда значение найдено, функция возвращает указатель на элемент списка, содержащий это значение. Если же совпадение не было найдено, будет возвращен **нулевой указатель**.

```
1   // Returning a Pointer from a Function
2
3   #include <stdio.h>
4
5   struct entry
6   {
7       int  value;
8       struct entry   *next;
9   };
10
11  struct entry  *findEntry (struct entry  *listPtr, int match)
12  {
13      while  ( listPtr != (struct entry *) 0 )
14          if ( listPtr->value == match )
15              return (listPtr);
16          else
17              listPtr = listPtr->next;
18
19      return (struct entry *) 0;
20  }
```

```
21
22      int main (void)
23      {
24          struct entry  *findEntry (struct entry  *listPtr, int match);
25          struct entry  n1, n2, n3;
26          struct entry  *listPtr, *listStart = &n1;
27
28          int search;
29
30          n1.value = 100;
31          n1.next =  &n2;
32
33          n2.value = 200;
34          n2.next =  &n3;
35
36          n3.value = 300;
37          n3.next =  0;
38
39          printf ("Enter value to locate: ");
40          scanf ("%i", &search);
41
42          listPtr = findEntry (listStart, search);
43
44          if ( listPtr != (struct entry *) 0 )
45              printf ("Found %i.\n", listPtr->value);
46          else
47              printf ("Not found.\n");
48
49          return 0;
50      }
```

Введем несколько значений и посмотрим, что выдаст программа:

```
Enter value to locate: 200
Found 200.

Enter value to locate: 400
Not found.
```

Заголовок функции

```
    struct entry  *findEntry (struct entry  *listPtr, int match)
```

говорит о том, что функция возвращает указатель на тип `struct entry`, а на вход получает указатель на первый элемент связного списка и искомое число. Внутри функции реализован цикл `while`, который продолжается пока не будет найдено искомое число, либо не будет достигнут нулевой указатель. Во втором случае функция именно его и возвращает как индикатор отсутствия искомого числа в списке.

Указатель, возвращенный функцией, обычно используется для доступа к другим элементам записи списка. Возвращаясь к словарю, можно по указателю получить, например, перевод слова на нескольких разных языках или его определение. Еще одним достоинством использования связного списка в словаре - возможность очень легкого добавления элементов в список и удаления элементов из него, – задача сводится к определению начального и нулевого указателей и переопределению нескольких указателей, если нужно что-то добавить или убрать.

У связных списков есть и свои недостатки, например, через связные списки невозможно реализовать быстрые алгоритмы поиска, можно только последовательно перебирать элементы списка, потому что доступ к следующему элементу можно получить только через предыдущий элемент списка.

Этот недостаток можно обойти, если использовать другие структуры данных, например, **дерево**. Таблицы хэшей тоже являются неплохим вариантом. Но это будет рассмотрено в курсе алгоритмики.

Указатели и массивы

Указатели очень часто используются в применении к массивам. Часто с их помощью можно ускорить работу программы и снизить потребление оперативной памяти.

Допустим, у нас есть массив `values` из `100` чисел, а для доступа к нему будем использовать указатель `valuesPtr`:

```
int  *valuesPtr;
```

То есть, если нужен указатель на массив, не нужно как-то обозначать, что мы указываем на массив, достаточно просто указать тип элементов массива.

Чтобы установить указатель на первый элемент массива, достаточно написать

```
valuesPtr = values;
```

Адресный оператор здесь не нужен, потому что в языке С компилятор воспринимает название массива как указатель на массив (*мы это уже проходили в уроке про пользовательские функции, где у нас функции не копировали массив во внутреннюю переменную, а работали с ним напрямую через указатель на переданный в функцию массив*).
То есть, `values` в выражении выше уже является указателем на массив.

Но можно и традиционным способом сделать указатель на первый элемент массива:

```
valuesPtr = &values[0];
```

Принципиальной разницы нет, здесь только дело вкуса. Кстати с подходом выше можно установить указатель на **любой** элемент массива.

Настоящий потенциал указателей на массивы раскрывается когда нужно перемещаться по массиву. Если мы установили указатель `valuesPtr` на первый элемент массива `values`, то через выражение

```
*valuesPtr
```

можно получить доступ к первому элементу массива, т.е. к `values[0]`. Для доступа к `values[3]` достаточно прибавить 3 к значению указателя:

```
*(valuesPtr + 3)
```

То есть, в общей форме, чтобы получить доступ к элементу `values[i]`, нужно написать

```
*(valuesPtr + i)
```

Операции над элементами массива выглядят как обычно:

```
*(valuesPtr + 10) = 27;
```

Чтоб перевести указатель на следующий элемент массива, достаточно увеличить его на 1:

```
++valuesPtr;
```

Можно и в обратную сторону двигаться, достаточно `++` заменить на `--`. Также можно прибавлять и отнимать числа.

Рассмотрим еще несколько вариантов работы с указателями.

В языке С также допустимо **сравнение** двух указателей. Это особенно полезно, когда нужно сравнить два указателя на один и тот же массив. Например, можно проверить, не вышел ли указатель за пределы массива. Так как наш массив имеет 100 элементов, проверка будет выглядеть так:

```
valuesPtr > &values[99]
```

выражение выше вернет `TRUE`, если указатель вышел за пределы массива `values`. Еще можно вот так:

```
valuesPtr > values + 99
```

Это выражение допустимо, потому что когда мы пишем просто `values`, это создает указатель на **первый**

элемент массива, т.е. это то же самое, что `&values[0]`.

Программа ниже иллюстрирует работу указателей на массивы. Функция `arraySum()` находит сумму всех элементов массива.

```
1    // Function to sum the elements of an integer array
2
3    #include <stdio.h>
4
5    int  arraySum (int  array[], const int  n)
6    {
7         int  sum = 0, *ptr;
8         int  * const arrayEnd = array + n;
9
10        for ( ptr = array;  ptr < arrayEnd;  ++ptr )
11             sum += *ptr;
12
13        return sum;
14   }
15
16   int main (void)
17   {
18        int  arraySum (int  array[], const int  n);
19        int  values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };
20
21        printf ("The sum is %i\n", arraySum (values, 10));
22
23        return 0;
24   }
```

Программа выведет:

```
The sum is 21
```

В функции `arraySum()` указатель-константа `arrayEnd` указывает на место сразу после последнего элемента входного массива. `n` – количество элементов в массиве `array[]`.

Далее запускается цикл `for`, который устанавливает указатель `ptr` в начало массива и каждый цикл увеличивает его на 1, пока указатель не дойдет до конца массива (`arrayEnd`). Попутно суммируя элементы массива, на которые будет указывать `ptr`.

Немного про оптимизацию программ

Думаю, очевидно, что в рассмотренной программе можно обойтись без переменной `arrayEnd` и вместо нее можно проводить проверку значения `ptr` напрямую:

```
for ( ...; pointer <= array + n; ... )
```

Но использование `arrayEnd` помогает ускорить работу цикла. При использовании выражения выше программе придется каждый раз вычислять сумму двух элементов, тратя ценное время. Да, разница неощутима при работе с массивами из 10 элементов, но она будет существенной, если элементов тысячи. Поэтому намного оптимальнее будет рассчитать условие выполнения цикла до его начала, а не перерассчитывать его каждую итерацию цикла, так как сумма `array + n` никак не может поменяться по ходу работы цикла.

Далее рассмотрим целесообразность применения указателей по сравнению с традиционным подходом (через переменную-счетчик `i`). В целом, при последовательном переборе элементов массива подход с указателями работает быстрее, так как компилятор генерирует более оптимальный код *прим.: Анон говорит, что современные компиляторы компилируют оба варианта в одинаково быстрый код. Разница будет только в случае работы с массивами переменной длины – VLA*.

Но во всех других случаях производительность двух подходов одинакова, – расчет выражения `*(pointer + i)` занимает столько же времени, сколько `array[i]`.

Как отличить массив от указателя

Как мы уже знаем, если в любом выражении написать название массива, компилятор обработает его как указатель на первый элемент массива. Это объясняет то, почему мы можем изменять элементы массива, не копируя его в локальные переменные работающей с этим массивом функции.

Но может возникнуть вопрос, раз мы передаем в функцию массив, но при этом фактически передается указатель на этот массив, почему же мы при объявлении функции не используем запись

```
int  *array;
```

Или почему мы до этого урока не использовали указатели для работы с массивами?

Чтобы ответить на эти вопросы, вспомним, что valuesPtr указывает на один и тот же тип данных, так как это элементы массива values и поэтому выражение `*(valuesPtr + i)` полностью эквивалентно выражению `values[i]`, при условии, что `valuesPtr` указывает на начало массива `values`. Из этого следует, что `*(valuesPtr + i)` можно использовать для обращения к `i`-му элементу массива, т.е. в общем случае выражение `x[i]` может быть заменено на `*(x + i)`.

Из вышесказанного видно, что указатели и массивы тесно связаны в C и именно это позволяет нам в функции `arraySum()` объявить `array` как "массив `int`-ов" **или** как "указатель на `int`". Оба варианта будут работать одинаково хорошо.

Так как теперь мы понимаем, что ничто не мешает в функции объявить массив как указатель, перепишем нашу программу с учетом этого, попутно избавившись от переменной `ptr`, потому что она больше не будет нам нужна.

```
1   // Function to sum the elements of an integer array  Ver. 2
2
3   #include <stdio.h>
4
5   int  arraySum (int  *array, const int  n)
6   {
7        int  sum = 0;
8        int  * const arrayEnd = array + n;
9
10       for (  ; array < arrayEnd;  array++ )
11            sum += *array;
12
13       return sum;
14  }
15
16  int main (void)
17  {
18       int  arraySum (int  *array, const int  n);
19       int  values[10] = { 3, 7, -9, 3, 6, -1, 7, 9, 1, -5 };
20
21       printf ("The sum is %i\n", arraySum (values, 10));
22
23       return 0;
24  }
```

Суть изменений довольно очевидна и не требует подробных пояснений. Единственное, на что стоит обратить внимание – указатель `*array` изменяется только внутри функции, т.к. мы уже говорили о том, что указатель передается как локальная переменная, т.е. мы можем проводить любые операции над передаваемым в функцию указателем, не боясь за его сохранность. На данные в массиве операции над указателем также никак не влияют.

Указатели на символьные строки

Указатели также часто применяются при работе со строками. Прежде чем мы приступим, напишем функцию `copyString()`, которая будет копировать одну строку в другую.

```
1    void copyString (char  to[], char  from[])
2    {
3         int  i;
4
5         for ( i = 0;  from[i] != '\0';  ++i )
6              to[i] = from[i];
7
8         to[i] = '\0';
9    }
```

Здесь цикл for завершается перед тем, как нулевой символ будет скопирован в массив to, поэтому мы его дописываем после цикла.

Теперь перепишем эту функцию, используя указатели. Это поможет нам избавиться от счетчика `i`.

```
1    #include <stdio.h>
2
3    void copyString (char  *to, char  *from)
4    {
5         for (  ;  *from != '\0';  ++from, ++to )
6              *to = *from;
7
8         *to = '\0';
9    }
10
11   int main (void)
12   {
13        void  copyString (char  *to, char  *from);
14        char  string1[] = "A string to be copied.";
15        char  string2[50];
16
17        copyString (string2, string1);
18        printf ("%s\n", string2);
19
20        copyString (string2, "So is this.");
21        printf ("%s\n", string2);
22
23        return 0;
24   }
```

Программа выдаст:

```
A string to be copied.
So is this.
```

В функцию передаются два аргумента – `to` и `from`, но в виде указателей, а не в виде массивов, как это было в начальном варианте функции. Затем в цикле `for` происходит поэлементное копирование одной строки в другую, при этом каждую итерацию цикла значения указателей увеличиваются на 1. Цикл завершается по достижению нулевого символа в массиве `from`.

Указатели и строки-константы

В последней программ вызов функции

```
copyString (string2, "So is this.");
```

интересен тем, что здесь передается указатель на строку-константу **целиком**. Когда создается строка-константа, на нее **всегда** создается указатель. Таким образом, если задан указатель на `char`:

```
    char  *textPtr;
```

то выражение

```
    textPtr = "A character string.";
```

присваивает в `textPtr` указатель на строку `"A character string."` целиком, а не на первый ее символ. Различайте указатели на `char` и массивы `char`, потому что операция присваивания, показанная выше, **не будет работать** с символьными массивами. Например, если у нас есть массив

```
    char  text[80];
```

то **нельзя** делать вот так:

```
    text = "This is not valid.";
```

Единственный случай, когда в языке С можно присвоить значение массиву символов – при инициализации:

```
    char  text[80] =  "This is okay.";
```

причем здесь будет создан указатель на первый символ в массиве, как это было с массивом `int`-ов.

Если мы объявим `text` как указатель на `char`, то при его инициализации

```
    char  *text =  "This is okay.";
```

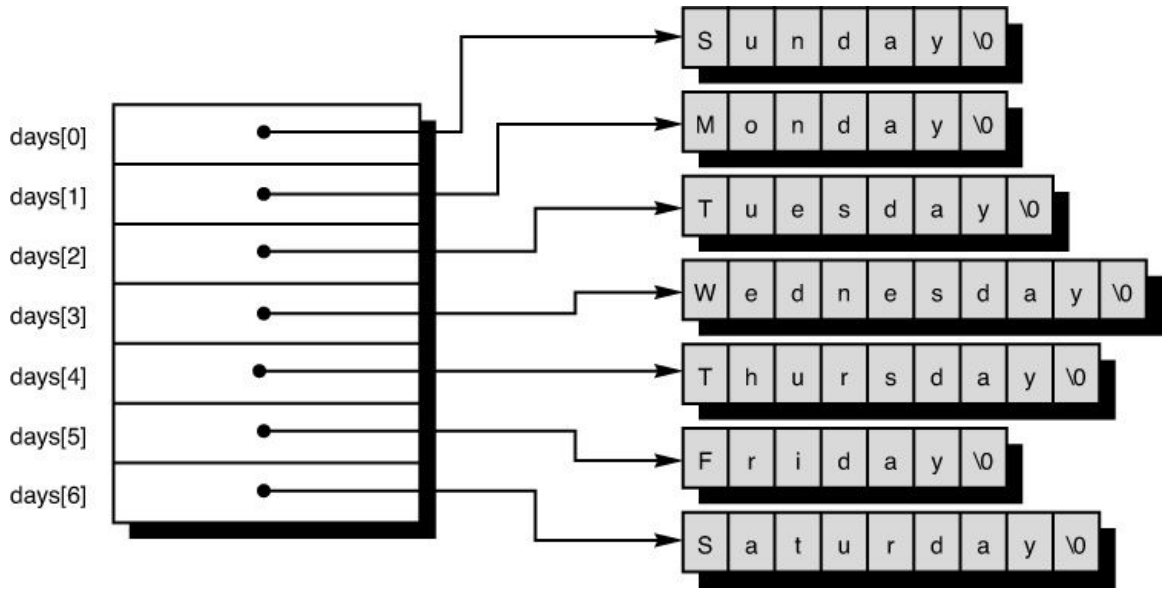
ему будет присвоен указатель на строку `"This is okay."`.

Для лучшей демонстрации работы указателей со строками-константами создадим массив `days`, который будет хранить указатели на названия дней недели.

```
    char *days[] =
        { "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday" };
```

Массив days содержит 7 элементов, каждый указывает на строку. Так, `days[0]` указывает на строку `"Monday"`, `days[1]` на `"Tuesday"` и так далее (см. рисунок ниже). Можно вывести на экран любой из элементов массива при помощи выражения

```
    printf ("%s\n", days[3]);
```



Массив указателей на строки-константы

Возвращаясь к операторам инкремента и декремента. Порядок выполнения операций.

До сих пор, мы ставили эти операторы как попало: то в начало, то в конец переменной и вроде все работало одинаково правильно вне зависимости от расположения оператора.

Но если вместе с операторами инкремента и декремента нужно использовать другие операторы, то порядок их написания уже будет иметь значение и нужно точно знать, как с ними работать.

Если `++` написан **перед** операндом, то это называется **преинкрементом,** а если **после** операнда, то **постинкрементом**. То же самое для оператора декремента. Выражение

```
--i;
```

производит **предекремент** переменной `i`, а выражение

```
i--;
```

производит ее **постдекремент**. Здесь оба выражения уменьшают `i` на 1.

Суть расположения этих операторов проявляется, когда в выражении используется больше одного оператора. Допустим, у нас есть два числа, `i` и `j`. Пусть `i = 0`, а для `j` запишем выражение

```
j = ++i;
```

После такого `j` будет присвоено `1`, потому что `i` была увеличена на `1` **до того**, как была выполнена операция присваивания. То есть, такое выражение эквивалентно выражениям

```
++i;
j = i;
```

Если же вместо пре- использовать постинкрементальный оператор:

```
j = i++;
```

то `i` будет увеличено на 1 **после того** как ее значение будет присвоено `j`. Эквивалент из двух выражений выглядит так:

```
j = i;
++i;
```

То же самое работает и с более длинными выражениями. Так, если `i = 5`, то выражение

```
j = ++i * 3 - 2;
```

выдаст 16, но выражение

```
j = i++ * 3 - 2;
```

выдаст 13, а `i` будет инкрементировано уже после операции присваивания.

Теперь допустим, что `i = 1`, тогда выражение

```
x = a[--i];
```

присвоит переменной `x` значение `a[0]`, потому что `i` была уменьшена **до того**, как ее значение было использовано для обращения к массиву `a`. Если же сделать наоборот:

```
x = a[i--];
```

то переменной `x` будет присвоено значение `a[1]`, потому что здесь декремент происходит уже **после** обращения к `i`.

В качестве третьего примера можно привести вызов функции

```
printf ("%i\n", ++i);
```

которая инкрементирует i и потом отправляет ее значение в функцию printf(), но вызов

```
printf ("%i\n", i++);
```

увеличивает i уже после того, как ее значение было напечатано на экране. То есть, если i = 100, то в первом случае будет напечатано 101, а во втором 100. В обеих случаях после выполнения вызовов значение i будет 101.

Если мы напишем вот так:

```
    *valuesPtr + 10
```

то это не сдвинет указатель на 10 элементов. Здесь на 10 увеличится элемент массива, на который указывает `valuesPtr`, потому что мы к нему обратились через **операцию разыменования** (*).
Но если мы напишем так:

```
    *valuesPtr++
```

то это сдвинет указатель на одну позицию вверх, потому что операция разыменования имеет **такой же** приоритет, что и операции инкремента и декремента (`++` и `--`) , то есть сначала к указателю присваивается единица. Затем, так как инкремент постфиксный, с помощью операции разыменования возвращается значение, которое было до инкремента. Получается, что такой код аналогичен такой записи:

```
    *valuesPtr
    valuesPtr++
```

Скобки изменят порядок операций:

```
    (*valuesPtr)++
```

Здесь сначала выполняется операция разыменования, а потом увеличение на 1 полученного по указателю значения. Аналогично будет с префиксным инкрементом:

```
    ++*valuesPtr
```

В данном случае сначала с помощью операции разыменования получаем значение по адресу из указателя `valuesPtr`, и к этому значению прибавляется единица.

Поменяем операторы местами:

```
    *++valuesPtr
```

Теперь сначала изменяется адрес в указателе, затем мы получаем по этому адресу значение. Полученное значение может быть неопределенным и стоит избегать такой записи (*уточнить*).

Выражение

```
    *(++textPtr)
```

сначала инкрементирует `valuesPtr`, а затем получает значение, на которое он будет указывать после инкремента., а выражение

```
    *(textPtr++)
```

сначала получает значение, на которое указывает `valuesPtr`, а потом уже икрементирует `valuesPtr`.

Теперь вернемся к нашей функции `copyString()` и перепишем ее, используя операторы инкремента прямо в цикле `for`:

```
    for (  ;  *from != '\0';  )
         *to++ = *from++;
```

То есть, сначала происходит получение символа, на который указывает `from`, затем `from` инкрементируется. Затем полученный символ из `from` записывается туда, куда указывает `to` и в конце инкрементируется `to`, переходя на следующий символ строки.

Внимательно проработайте этот цикл. Такие конструкции часто используются в С, поэтому очень важно понимать, как они работают.

У нас получилось, что в цикле `for` не осталось ни начального условия, ни инкремента. В таком случае рациональнее будет использовать цикл `while`:

```c
// Function to copy one string to another. Pointer Ver. 2

#include <stdio.h>

void  copyString (char  *to, char  *from)
{
     while ( *from )
          *to++ = *from++;

     *to = '\0';
}

int main (void)
{
     void  copyString (char  *to, char  *from);
     char  string1[] = "A string to be copied.";
     char  string2[50];

     copyString (string2, string1);
     printf ("%s\n", string2);

     copyString (string2, "So is this.");
     printf ("%s\n", string2);

     return 0;
}
```

Здесь мы использовали то, что нулевой символ численно равен нулю и цикл завершится как раз в этом случае, такая практика очень распространена в языке C.

Операции над указателями

Мы уже знаем, что к указателям можно прибавлять и вычитать целые числа. Также можно сравнивать между собой два указателя, – операторы `<`, `>` и `=` вполне применимы. Помимо этого еще можно **вычитать** указатели **одного** типа. Результатом вычитания будет количество элементов, находящихся между этими указателями. То есть, если `p` указывает на какой-то элемент массива `x`, выражение

```c
n = p - x;
```

присвоит переменной `n` индекс элемента в массиве `x`, на который указывает `p`. Например, если `p` указывает на 100-й элемент массива `x`:

```c
p = &x[99];
```

то после вычитания `n` будет равно 99.

прим.: тип значения (`int`, `long int` и т.д.), возвращаемого таким вычитанием, можно просмотреть в параметре `ptrdiff_t`, который находится в `<stddef.h>`.

С учетом этих знаний перепишем функцию `stringLength()` из урока 1.8 (*символьные строки*). Для движения по строке будем использовать указатель `cptr`. Когда указатель дойдет до **нулевого символа**, можно будет **вычесть** его значение из указателя на первый символ и получить искомую длину строки.

```c
// Function to count the characters in a string - Pointer version

#include <stdio.h>

int  stringLength (const char  *string)
{
     const char  *cptr = string;

     while ( *cptr )
          ++cptr;
```

```
11          return  cptr - string;
12      }
13
14      int main (void)
15      {
16          int  stringLength (const char  *string);
17
18          printf ("%i  ", stringLength ("stringLength test"));
19          printf ("%i  ", stringLength (""));
20          printf ("%i\n", stringLength ("complete"));
21
22          return 0;
23      }
```

Программа выдаст:

```
17  0  8
```

Указатели на функции

При работе с указателями на функции, компилятору C нужно не только знать, что указатель указывает на функцию, но и тип данных, возвращаемых этой функцией, а также количество и типы ее аргументов. Чтобы объявить `fpPtr` как указатель на функцию, возвращающую `int` и не принимающую аргументы, нужно написать:

```
int  (*fnPtr) (void);
```

Скобки вокруг `*fnPtr` **обязательны**, иначе компилятор воспримет это выражение как объявление функции `fnPtr`, которая возвращает указатель на `int`, потому что у оператора вызова функции `()` более высокий приоритет, чем у оператора разыменования (*).

Чтобы установить этот указатель на какую-то функцию, достаточно операции присваивания:

```
fnPtr = lookup;
```

Здесь, по аналогии с массивами, не нужно писать скобки и тогда компилятор C автоматически сформирует указатель на функцию. Можно для читабельности дописать `&`, но это не обязательно. Помните, что функция `lookup()` должна быть объявлена **до** присваивания указателя на нее. Либо нужно создать **прототип** этой функции. В общем, все так же, как и с вызовом функции.

Вызов функции через указатель производится через замену названия функции названием указателя. Например, выражение

```
entry = fnPtr ();
```

вызывает функцию, на которую указывает `fnPtr`, записывая выданное значение в переменную `entry`. Если функция принимает аргументы, они пишутся внутри скобок, как и при обычном вызове функций. Указатели на функции обычно применяются для передачи функций в качестве аргумента в другие функции. Например, в стандартной библиотеке C есть функция `qsort()`, которая выполняет сортировку переданного ей массива. В качестве одного из своих аргументов она принимает указатель на функцию, которая будет вызываться, каждый раз, когда нужно будет между собой сравнить два элемента сортируемого массива. Таким образом расширяется гибкость этой функции, потому что использование внешней функции сравнения позволяет сортировать массивы любого типа. Подробнее работу функции `qsort()` мы рассмотрим в конце курса.

Еще одно распространенное применение указателей на функции – создание так называемых **диспетчерских таблиц** (*dispatch tables*). Это массив, который хранит указатели на различные функции. Например, можно создать таблицу для обработки команд, вводимых пользователем. Каждый элемент таблицы может содержать название команды и указатель на функцию, выполняющую эту команду. И когда пользователь вводит команду, можно запустить поиск по диспетчерской таблице и выполнить

функцию, соответствующую названию введенной команды.

Указатели и адреса в памяти

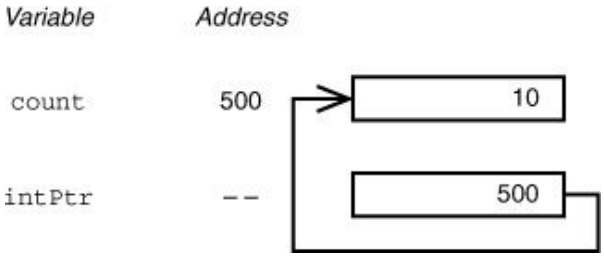
Прежде чем мы завершим этот урок, разберем, как именно представлены указатели в языке С. Память компьютера можно представить как последовательноость ячеек. У каждой ячейки памяти есть номер, который называется **адресом**. Обычно первый адрес в памяти – 0. В большинстве компьютерных систем ячейки называются **байтами**.

Когда мы объявляем переменную `count` типа `int`, система выделяет под нее адрес (или диапазон адресов) и все операции с этой переменной проводятся по этому адресу. К счастью, язык С – высокого уровня и поэтому нам не нужно постоянно думать о адресах памяти при написании программ, – за нас это делает система. Но понимание, как связаны переменные и выделенные для них адреса, поможет лучше понять принцип работы указателей.

Когда мы используем адресный оператор (`&`), он выдает адрес переменной в памяти. То есть,

```
intPtr = &count;
```

присвоит в `intPtr` адрес, который был выделен для переменной `count`. Так, если `count` находится по адресу 500 и равен 10, процесс можно проиллюстрировать так:



Как указатель указывает на адрес в памяти

Адрес `intPtr` здесь показан как `--` потому что он нас не особо интересует.

При использовании **оператора разыменования**:

```
*intPtr
```

будет получено значение, хранящееся по адресу, переданному в этот оператор, причем оно интерпретируется согласно типу указателя. Так, если `intPtr` – указатель на `int`, то и значение, возвращаемое оператором разыменования, будет тоже типа `int`. То есть в нашем случае оператор вернет 10 типа `int`, потому что типы указателя и переменной совпадают.

Запись значения по указателю:

```
*intPtr = 20;
```

происходит следующим образом. Значение intPtr интерпретируется как адрес в памяти. Затем указанное значение записывается по этому адресу. То есть, 20 будет записано по адресу 500.

В программировании встраиваемых систем, микроконтроллеров и т.д. работа с адресами часто бывает нужна, потому что там бывают случаи, когда нужно производить операции чтения и записи по четко определенным, фиксированным адресам памяти.

Вывести на экран адрес, на который указывает указатель, можно при помощи спецификатора `%p`. Напишем программу, в которой зададим переменную `count` и указатель `intPtr`, указывающий на нее, а затем выведем на экран адрес указателя и значение, полученное по этому адресу.

```
1    #include <stdio.h>
2
3    int main (void)
4    {
5         int count = 10;
6         int *intPtr = &count;
```

```
7          printf("intPtr: address = %p \t value = %d \n", intPtr, *intPtr);
8     }
```

Программа выдала:

```
intPtr: address = 0028ff08          value = 10
```

Очевидно, адрес каждый раз будет разный.

Источники:

1. Stephen Kochan – Programming in C (4th Edition); chapter 10
2. Арифметика указателей – https://metanit.com/cpp/c/5.3.php
3. Correct format specifier to print pointer or address? – https://stackoverflow.com/questions/9053658/correct-format-specifier-to-print-pointer-or-address

Упражнения

2. *Write a function called* `insertEntry()` *to insert a new entry into a linked list. Have the procedure take as arguments a pointer to the list entry to be inserted (of type* `struct entry` *as defined in this chapter), and a pointer to an element in the list after which the new* `entry` *is to be inserted.*

```
1     void insertEntry(struct entry *newElement, struct entry *prevElement)
2     {
3          newElement->next = prevElement->next;
4          prevElement->next = newElement;
5
6          return;
7     }
```

3. *The function developed in exercise 2 only inserts an element after an existing element in the list, thereby preventing you from inserting a new entry at the front of the list. How can you use this same function and yet overcome this problem?*

```
1     #include <stdio.h>
2
3     struct entry
4     {
5          int          value;
6
7          struct entry *next;
8     };
9
10    void insertEntry(struct entry *newElement, struct entry *prevElement)
11    {
12         newElement->next = prevElement->next;
13         prevElement->next = newElement;
14
15         return;
16    }
17
18    int main (void)
19    {
20         void insertEntry(struct entry *newElement, struct entry *prevElement);
21
22         struct entry n0, n1, n2, n3, n4;
23         struct entry *list_pointer = &n1;
24
25         n0.next = &n1;
26
```

```
27          n1.value = 10;
28          n1.next  = &n2;
29
30          n2.value = 20;
31          n2.next  = &n3;
32
33          n3.value = 30;
34          n3.next  = 0;
35
36          n4.value = 40;
37          n4.next  = 0;
38
39          printf("Before:\n");
40          while ( list_pointer != (struct entry *) 0 ) {
41               printf ("%i\n", list_pointer->value);
42               list_pointer = list_pointer->next;
43          }
44
45          insertEntry(&n4, &n0);
46          list_pointer = n0.next;
47
48          printf("\nAfter:\n");
49          while ( list_pointer != (struct entry *) 0 ) {
50               printf ("%i\n", list_pointer->value);
51               list_pointer = list_pointer->next;
52          }
53
54          return 0;
55     }
```

4. *Write a function called `removeEntry()` to remove an entry from a linked list. The sole argument to the procedure should be a pointer to the list. Have the function remove the entry after the one pointed to by the argument.*

```
1     void removeEntry(struct entry *dElement)
2     {
3          dElement->next = dElement->next->next;
4
5          return;
6     }
```

5. *Define the appropriate structure definition for a doubly linked list entry and then write a small program that implements a small doubly linked list and prints out the elements of the list.*

```
1     #include <stdio.h>
2
3     struct entry
4     {
5          int          value;
6          struct entry *prev;
7          struct entry *next;
8     };
9
10    int main (void)
11    {
12
13
14         struct entry n1, n2, n3;
15         struct entry *list_pointer = &n1;
16
17         n1.value = 10;
18         n1.prev  = 0;
19         n1.next  = &n2;
20
21         n2.value = 20;
```

```
22          n2.prev  = &n1;
23          n2.next  = &n3;
24
25          n3.value = 30;
26          n3.prev  = &n2;
27          n3.next  = 0;
28
29          printf("Forward:\n");
30          while ( list_pointer != (struct entry *) 0 ) {
31               printf ("%i\n", list_pointer->value);
32               list_pointer = list_pointer->next;
33          }
34
35          printf("\nBackwards:\n");
36          list_pointer = &n3;
37          while ( list_pointer != (struct entry *) 0 ) {
38               printf ("%i\n", list_pointer->value);
39               list_pointer = list_pointer->prev;
40          }
41
42          return 0;
43     }
```

6. *Develop `insertEntry()` and `removeEntry()` functions for a doubly linked list that are similar in function to those developed in previous exercises for a singly linked list.*

```
1     void insertEntry(struct entry *newElement, struct entry *prevElement)
2     {
3          newElement->next = prevElement->next;
4          newElement->prev = prevElement;
5          prevElement->next->prev = newElement;
6          prevElement->next = newElement;
7
8          return;
9     }
10
11    void removeEntry(struct entry *dElement)
12    {
13         if (dElement->next == (struct entry *) 0 )
14              dElement->prev->next = 0;
15         else
16              dElement->next->prev = dElement->prev;
17
18         if (dElement->prev == (struct entry *) 0 )
19              return;
20         else
21              dElement->prev->next = dElement->next;
22
23         return;
24    }
```

7. *Write a pointer version of the `sort()` function from "Working with Functions." Be certain that pointers are exclusively used by the function, including index variables in the loops.*

```
1     void  sort (int  *i, int  n)
2     {
3         int  *j, *end, temp;
4         end = i + n;
5         j = i;
6
7         for ( ;  i < end - 1;  ++i )
8             for ( j = i + 1;  j < end;  ++j )
9                 if ( (*i) > (*j) ) {
10                    temp = *i;
```

```
                    *i = *j;
                    *j = temp;
                }
    }
```

8. *Write a function called `sort3()` to sort three integers into ascending order. (This function is not to be implemented with arrays.)*

   Непонятно, что делать с этой задачей, не вижу какое она имеет отношение к указателям.

9. *Rewrite the `readLine()` function from Chapter 9 so that it uses a character pointer rather than an array.*

```
void  readLine (char  *buffer)
{
    char  character;

    do
    {
        character = getchar ();
        *buffer = character;
        ++buffer;
    }
    while ( character != '\n' );

    *--buffer = '\0';
}
```

10. *Rewrite the `compareStrings()` function from Chapter 9 to use character pointers instead of arrays.*

```
int  compareStrings (const char  *s1, const char  *s2)
{
    int answer;

    while ( *s1 == *s2 && *s1 != '\0'&& *s2 != '\0' )
        ++s1;
        ++s2;

    if ( *s1 < *s2 )
        answer = -1;               /* s1 < s2  */
    else if ( *s1 == *s2 )
        answer = 0;                 /* s1 == s2 */
    else
        answer = 1;                 /* s1 > s2  */

    return answer;
}
```

11. *Given the definition of a `date` structure as defined in this chapter, write a function called `dateUpdate()` that takes a pointer to a `date` structure as its argument and that updates the structure to the following day.*

```
struct date  dateUpdate (struct date  *today)
{
    struct date  tomorrow;
    int  numberOfDays (struct date  d);

    if ( today->day != numberOfDays (*today) ) {
        tomorrow.day = (today->day) + 1;
        tomorrow.month = today->month;
        tomorrow.year = today->year;
    }
```

```
11          else if ( today->month == 12 )  {   // end of year
12              tomorrow.day = 1;
13              tomorrow.month = 1;
14              tomorrow.year = (today->year) + 1;
15          }
16          else {                              // end of month
17              tomorrow.day = 1;
18              tomorrow.month = (today->month) + 1;
19              tomorrow.year = today->year;
20          }
21
22          return tomorrow;
23      }
```

12. *Given the following declarations, determine whether each* `printf()` *call from the following sets is valid and produces the same output as other calls from the set.*

```
1       int main (void)
2       {
3            char  *message = "Programming in C is fun\n";
4            char  message2[] = "You said it\n";
5            char  *format  = "x = %i\n";
6            int   x = 100;
7
8            /*** set 1 ***/
9            printf ("Programming in C is fun\n");        // OK
10           printf ("%s", "Programming in C is fun\n");  // OK
11           printf ("%s", message);                      // OK
12           printf (message);                            // OK
13
14           /*** set 2 ***/
15           printf ("You said it\n");                    // OK
16           printf ("%s", message2);                     // OK
17           printf (message2);                           // OK
18           printf ("%s", &message2[0]);                 // OK
19
20           /*** set 3 ***/
21           printf ("said it\n");                        // OK
22           printf (message2 + 4);                       // OK
23           printf ("%s", message2 + 4);                 // OK
24           printf ("%s", &message2[4]);                 // OK
25
26           /*** set 4 ***/
27           printf ("x = %i\n", x);                      // OK
28           printf (format, x);                          // OK
29
30           return 0;
31      }
```

Полезный вывод: при помощи указателей в `printf()` можно передать что угодно, главное что оно было правильного формата. Указатель на строку целиком можно передавать без указания типа данных.