компьютерного Центр обучения «СПЕЦИАЛИСТ» при МГТУ им. Н.Э.Баумана®

Язык программирования C

В.Г.Тетерин – Microsoft Solution Developer (Visual C++)
teterin@specialist.ru

компьютерного Центр обучения «СПЕЦИАЛИСТ» при МГТУ им. Н.Э.Баумана®

МОДУЛЬ 9

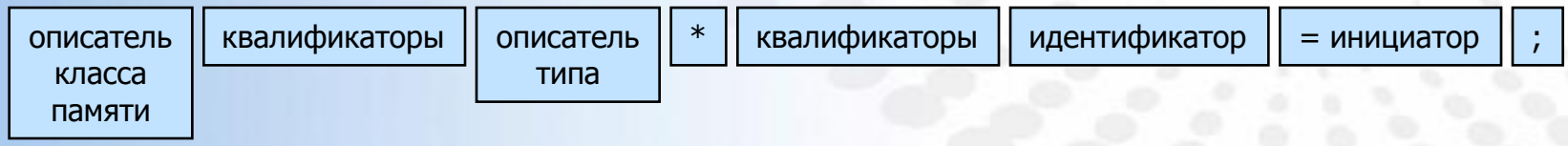
УКАЗАТЕЛИ

Модуль 9. Указатели

- Декларация указателей
- Операции с указателями
- Использование указателей как аргументов функции
- Указатели на функции
- Указатели и массивы
- Указатели и строки

Декларация указателей

- Тип *указатель* имеют переменные, значениями которых являются *адреса* других объектов программы.
- В С указатель обязательно связывается с *типом* того объекта, на который он ссылается, что должно быть явно указано в описании:



  - Например:

```
int *p;
double *q;
const char *s;
char * const t = s;
```

- В одном описании можно совмещать определение обыкновенных переменных и указателей, как показано ниже:

```
int x, *px;
```

  Здесь определены переменная x целого типа и указатель **px**, который может адресовать любой объект целого типа.

- Основное применение указатели находят для *косвенной адресации* объектов программы, главным образом:
  - для передачи в функцию параметров *"по адресу"* ,
  - для работы с *динамической памятью («кучей», heap)*.

Операции с указателями

- **Адресные операции**. В С существуют две специальных унарных адресных операции:

  & - получение адреса объекта (получение ссылки на объект)

  ***** - доступ к значению объекта по адресу (по ссылке, по указателю)

  – Операндом операции **&** может быть переменная или составной объект.

    - Нельзя получить адрес константы или выражения: недопустимы конструкции вида

      ```
      &5, &"Hello", &(x+1)
      ```

    - Нельзя получить адрес битового поля или переменной с классом памяти **register**.

  – Операндом операции ***** должен быть указатель, результатом будет значение и тип того объекта, на который ссылается указатель.

    - Пример Пусть **x** и **y** - переменные типа **int**, а **p** - указатель на **int**:

      ```
      extern int x;  int y, *p=&x;
      ```

      тогда выражение

      ```
      y=*p;
      ```

      присвоит **y** содержимое того, на что указывает **p**, то есть значение **x**. Последнее выражение эквивалентно выражению

      ```
      y=x;
      ```

- Над указателями можно выполнять некоторые арифметические операции, которые реализуются по специальным правилам *адресной арифметики*, и ряд других специальных операций.

Операции с указателями (продолжение)

- **Адресная арифметика** рассматривает арифметические операции, в которые входят *указатели*.
  - Набор таких операций, где использование указателей является осмысленным, достаточно ограничен.
- **Присваивание** указателю значения *другого указателя*
  - Пример. Если **p** и **q** - указатели на объекты, имеющие одинаковый тип, то выражение

```
p = q
```

    копирует значение **q** в **p**, то есть **p** будет указывать на тот же объект, что и **q**.

  - Если указатели ссылаются на *разные типы* данных, то при компиляции такого выражения выдается предупреждающее *сообщение* .
    - Такое предупреждение можно подавить с помощью операции *явного приведения типа*, когда такое присваивание имеет смысл.
  - Если указатель **p** ссылается на тип **void**, то такому указателю можно присваивать значение указателя на *любой тип* данных.
  - Указателю на функцию *нельзя* присвоить значение указателя на данные и наоборот.
  - Присваивание указателю одного типа значения указателя другого типа лишь в крайне редких ситуациях может оказаться оправданным.

Операции с указателями (продолжение)

- **Присваивание** указателю *числового значения*.
  - Единственное числовое значение, которое разрешено присваивать указателю, - это *нулевое значение*.
  - Это значение рассматривается в С не как адрес, а как *признак отсутствия* значения у указателя и для него с помощью директивы препроцессора **`#define`** введено специальное имя **`NULL`**:

```
p = NULL
```

    - Определение директивы **`#define`** содержится в заголовочных файлах **`<stdio.h>`** и **`<stdlib.h>.`**
  - В С допускается присваивание указателям и иных значений *целочисленных* констант и переменных и, наоборот, переменным целого типа - значение указателя, но при компиляции таких выражений выдается предупреждающее сообщение.
    - Как обычно, такое предупреждение при необходимости можно подавить с помощью применения операции *явного приведения* типа, если присваиваемое значение является правильным адресом.
- **Сравнение** указателей.
  - К указателям можно применять все операции сравнения:

`==`, **`!=`**, **`>`**, **`>=`**, **`<`**, **`<=`**.

Операции с указателями (продолжение)

- **Сложение указателя с целым значением**
  - Пусть указатель

```
int i = 3, x = 0, *p = &x;
```

    «настроен» на некоторый допустимый адрес памяти, тогда значением выражения

```
p + 1
```

    будет адрес памяти, расположенный сразу вслед за тем элементом типа **`int`**, который адресуется указателем **`p`**, а значением выражения

```
p + i
```

    будет адрес, смещенный на **`i`** элементов типа **`int`** относительно точки, адресуемой указателем **`p`**.
  - Это происходит благодаря специальному свойству *адресной арифметики* языка С – перед сложением с адресом автоматически производится умножение **`i`** на **`sizeof(type)`** - размер в байтах типа элемента, адресуемого указателем.
    - Это справедливо, для любых типов объектов, на которые ссылается указатель **`p`** – компилятор будет *масштабировать* **`i`** в соответствии с типом из описания для **`p`**.
- **Вычитание из указателя целого значения**
  - Операция вычитания

```
p - i
```

    выполняется аналогично сложению – с *масштабированием* **`i`** в соответствии с типом из описания для **`p`**.

Операции с указателями (продолжение)

- **Продвижение указателя**
  - Операции

    `++p` и `p++`

    обеспечивают продвижение указателя к следующему, а операции

    `--p` и `p--`

    - к предыдущему адресу с автоматическим масштабированием величиной `sizeof(type)`.

  - Различие между префиксной и постфиксной формами этих операций проявляется тогда, когда продвижение указателя совмещается с обработкой.
    - Так результатом операции

      `*++p = 0`

      будет продвижение указателя, а затем присваивание нулевого значения, а операции

      `*p++ = 0`

      - вначале присвоение нулевого значения по текущему адресу, а потом продвижение указателя.

  - Для продвижения указателя на `i` элементов используются операции

    `p += i` и `p -= i`.

- **Разность двух указателей**
  - При вычитании двух указателей одного типа

    `p-q`

    разность адресов (значений этих указателей) автоматически делится на размер `sizeof(type)` общего для них типа, заданного в описании указателей.
    - В противном случае результат операции не определен.

Операции с указателями (продолжение)

- **Другие свойства адресной арифметики**
  - Иные арифметические операции, кроме рассмотренных выше, к указателям не применимы:
    - Указатели нельзя складывать между собой, умножать, делить, использовать в операциях с ними значения плавающего типа и так далее.
- **Многоступенчатая косвенная адресация**
  - Глубина косвенной адресации в Си не ограничивается, так описание

```
int **ptr;
```

вводит указатель **ptr**, значением которого будет адрес некоторого другого указателя, который в свою очередь будет ссылаться на объект целого типа (двухступенчатая косвенная адресация).

Например, если:

```
int x = 10,
int *p = &x;
int **ptr = &p;
```

тогда верны следующие равенства:

```
 *ptr == p;
**ptr == x;
```

  - Появление третьего символа ***** в описании приведет к трехступенчатой косвенной адресации и так далее.

Использование указателей как аргументов функции

- При рассмотрении функций отмечалось, что в языке С при вызове функции все ее параметры передаются *"по значению":*
  - вызванная функция получает в промежуточных переменных (фактически в стеке) *копии значений* своих аргументов.
  - если функция изменяет свои аргументы, то все эти изменения касаются лишь значений локальных копий и никак не отражаются на значениях переменных в вызвавшей ее функции.

- **Передача параметров по адресу**
  - Вызывающая функция может передавать вызываемой функции не значения, а *адреса* своих переменных.
    - В этом случае формальные аргументы в вызываемой функции должны быть *указателями*.
    - Тогда вызванная функция, используя *операцию разадресации* указателей, получает доступ к значениям этих объектов и возможность их изменения.
    - Пример:

```
void swap(int *a, int *b)      //агументы - указатели!!!
{
            int t=*a; *a=*b; *b=t;      //разадресация указателей
}
```

В этом случае, при вызове

```
int x = 5, y = 2;
swap(&x,&y);   //фактические аргументы д.б. адресами!!!
```

переменные x и y обменяются своими значениями.

Указатели на функцию

- В языке С можно определить *указатель на функцию*, его значением является *адрес точки входа* в соответствующую функцию.
  - Очевидно, что применение арифметических и условных операций к такому указателю не имеет смысла, зато его можно *передавать в функции* и *получать* оттуда, помещать в массивы и т.п.
  - Важнейшая операция - это *доступ по указателю*, то есть *вызов функции*, на которую он ссылается.
- Объявление указателя на функцию:

```
double (*pf)(double, int);
```

  сообщает, что **pf** - это указатель на функцию, возвращающую значение типа **double** и имеющую два аргумента типа **double** и **int** соответственно.
  - Заметим, что скобки вокруг ***pf** обязательны, без них описание

```
double *pf(double, int);
```

    объявляло бы функцию **pf**, возвращающую указатель на переменную типа **double**.
- Подобно имени массива, имя функции без последующих круглых скобок трактуется как указатель-константа на функцию, поэтому присвоить указателю ссылку на конкретную функцию можно как явным присваиванием:

```
extern double sin(double);

double (*f)(double);

f = sin;
```

  так и передачей имени как параметра в функцию:

```
s = integral(0., 3.14, 20, sin);
```

Указатели на функцию (продолжение)

- Вызов функции по указателю осуществляется выражением:

  `(*f)(x)`

  – Здесь все согласовано с описанием: `f` - указатель на функцию, `*f` - сама функция, а `(*f)(x)` - обращение к этой функции.

  – Все скобки необходимы - они требуемым образом выделяют компоненты обращения.

  – Однако, поскольку указатель на функцию используется, главным образом, для ее вызова, допускается и *упрощенная форма*:

  `f(x)`

  Внешне она выглядит так, будто указатель `f` и есть имя функции.

- Пример функции, использующей указатель на другую функцию:

```
/* вычисление методом трапеций интеграла от f на интервале
   [a, b] при разбиении его на n частей */
double integral(double a, double b, int n, double (*f)(double))
{
          int i; double s, h;
          h =(b-a) / n;
          s = ( f(a) + f(b) ) / 2;
          for(i=1, a+=h; i<n; i++, a+=h)
              s += f(a);
          return s*h;
}
```

Указатели и массивы

- Концепции массивов и указателей в С очень тесно взаимосвязаны. Пусть имеется следующее определение массива:
  - `double list[10];`
    - Как уже упоминалось, имя массива трактуется в Си как *указатель-константа* на первый элемент массива:
  - `list` есть адрес `list[0]`.
    - Кроме того, оказываются верными и следующие соответствия:
  - `list+1` есть адрес `list[1]` ,
  - `list+2` есть адрес `list[2]` ,
  - **и т.д.**
  - Это вытекает из свойств адресной арифметики – при увеличении (или уменьшении) значения указателя на целое число оно автоматически *масштабируется* на размер типа указателя, что и приводит к вычислению правильного адреса.
- В силу сказанного, получить доступ к элементам массива можно не только используя его имя с соответствующим индексом, но также и с помощью указателя подходящего типа, "настроенного" на этот массив.
  - Если определить указатель на тип `double` и присвоить ему адрес 1-го элемента массива `list`:

    ```
    double *ptr = list;
    ```

    то после увеличения

    ```
    ptr=ptr+3;
    ```

    указатель ptr будет равен адресу элемента `list[3]`.

Указатели и массивы (продолжение)

- Для указателей и массивов в C справедливы следующие утверждения.
  - Если указатель «настроен» на массив:

```
double list[10], *ptr = list;
```

  - то выполняются равенства:

```
ptr   == &list[0]
ptr+1 == &list[1]
     ...
ptr+i == &list[i]
```

  - После разадресации имеем:

```
*ptr     == list[0]
*(ptr+1) == list[1]
     ...
*(ptr+i) == list[i]
```

  - Для выражения `*(ptr+i)` в C используется следующая сокращенная запись:

```
*(ptr+i) == ptr[i]
```

  - Тогда имеем:

```
ptr[0] == list[0]
ptr[1] == list[1]
     ...
ptr[i] == list[i]
```

  - откуда ясно видна тесная взаимосвязь массивов и указателей.

Указатели и массивы (продолжение)

- **Функции, указатели и массивы**
  - Если функции в качестве ее параметра передается имя массива, то оно передается как местоположение начала массива.
  - Внутри вызванной функции этот аргумент является переменной, поэтому аргумент, связанный с именем массива, фактически есть *указатель*, и в него копируется адрес первого элемента массива.
  - В функции этот аргумент может использоваться двояко: его можно *индексировать* для получения доступа к требуемому элементу массива или применять *операцию разадресации*.
    - Можно даже, если это удобно, использовать вместе и то и другое толкование.
  - Пример с индексированием:

```
int sum(int a[], int n)
{     int i, s=0;
      for(i=0; i<n; ++i) s += a[i];
      return s;
}
```

  - Пример с указателем:

```
int sum(int *p, int n)
{     int s=0;
      while(n-- > 0) s += *p++;
      return s;
}
```

Указатели и массивы (продолжение)

- Важно отметить принципиальную разницу в использовании имени массива и указателя на массив.
  - Имя массива является *указателем-константой*, что исключает любую попытку изменения его значения, но подобное ограничение не распространяется на указатель, "настроенный" на массив.
  - Другое различие состоит в том, что при определении массива для него *резервируется требуемый объем памяти*, в то время как при определении указателя его "настройка" на свободный участок памяти не происходит.
  - Для обеспечения возможности использования указателя необходимо
    - либо присвоить ему адрес предварительно зарезервированного массива,
    - либо выделить необходимую память с помощью библиотечных функций динамического распределения памяти:

`void *malloc( size_t size );` - выделяет блок памяти размером **`size`** байт.

`void *calloc( size_t nitems, size_t size );` - выполняет то же, что и **`malloc`**, но размер блока вычисляется как **`nitems * size`** и выполняется очистка (обнуление) выделенной памяти.

`void *realloc( void *block, size_t newsize );` - изменяет первоначальный размер выделенного блока, приводя его к величине **`newsize`** и переписывая при необходимости в новое место памяти.

» Все три функции возвращают указатель на выделенный блок памяти или **`NULL`** при недостатке необходимого объема свободной памяти в "куче" (heap)

`void free( void *block );` - возвращают в "кучу" предварительно распределенный предыдущими функциями блок; указатель **`block`** должен содержать адрес начала выделенного блока.

– *Замечание*. Прототипы всех функций содержатся в файле **`<stdlib.h>`**

Указатели и строки

- Строки в языке С – это одномерные символьные массивы, завершенные нулевым байтом и потому не требующие при их обработке указания размера строки.
- Характерным приемом в С является использование указателей для работы со строками.
  - Пример часто используемого способа применения указателя на символьную строку:

```
const char *p = "Visual C++";
```

    - В данном случае использование указателя вполне корректно, поскольку символьная строка физически размещается компилятором где-то в статической области памяти в виде массива символов, а затем указатель получает значение адреса первого символа этой строки.
  - Пример функции копирования символьных строк:

```
void strcopy(char *t, const char *s)
{
    while(*t++ = *s++);
}
```

Итоги

- В этом модуле Вы изучили:
  - Понятие указателя и его роль в языке C
  - Виды указателей и синтаксис их определения
  - Семантику адресной арифметики
  - Основные способы применения указателей

Вопросы?

- В.Г.Тетерин – Microsoft Solution Developer (Visual C++)
- teterin@specialist.ru

ПРИЛОЖЕНИЕ

ЗАДАЧИ

Задачи

1. Рассмотрите возможности применения указателей в реализации задач модуля 8
2. Примените динамические массивы в реализации задач модуля 8