Лекции по C/C++: указатели и строки Си

Указатель - это переменная, содержащая *адрес* некоторого объекта в оперативной памяти (ОП). Смысл применения указателей - косвенная адресация объектов в ОП, позволяющая динамически менять логику программы и управлять распределением ОП.

Основные применения:

- работа с массивами и строками;
- прямой доступ к ОП;
- работа с динамическими объектами, под которые выделяется ОП.

Описание указателя имеет следующий общий вид:

```
тип *имя;
```

то есть, указатель всегда *адресует определённый тип объектов*! Например,

```
int *px; // указатель на целочисленные данные
char *s; //указатель на тип char (строку Си)
```

Опишем основные операции и действия, которые разрешены с указателями:

1. Сложение/вычитание с числом:

```
px++;
 //переставить указатель px на sizeof(int) байт вперед
s--;
 //перейти к предыдущему символу строки
 //(на sizeof(char) байт, необязательно один)
```

2. Указателю можно присваивать адрес объекта унарной операцией "`&`":

```
int *px; int x,y;
px=&x;
 //теперь px показывает на ячейку памяти со
 // значением x
px=&y; //а теперь - на ячейку со значением y
```

3. Значение переменной, на которую показывает указатель, берется унарной операцией "`*`" ("взять значение"):

```
x=*px;   //косвенно выполнили присваивание x=y
(*px)++; //косвенно увеличили значение y на 1
```

Важно! Из-за приоритетов и ассоциативности операций C++ действие

```
*px++;
```

имеет совсем другой смысл, чем предыдущее. Оно означает "взять значение `y` (`*px`) и затем перейти к следующей ячейке памяти (`++`)"

Расшифруем оператор

```
x=*px++;
```

Если `px` по-прежнему показывал на `y`, он означает "записать значение `y` в `x` и затем перейти к ячейке памяти, следующей за `px`". Именно такой подход в классическом Си используется для сканирования массивов и строк.

Вот пример, с точностью до адресов памяти показывающий это важное различие. Комментарием приведены значения и адреса памяти переменных `x` и `y`, а также значение, полученное по указателю `px` и адрес памяти, на который он показывает. Обратите внимание, что после выполнения второго варианта кода значение, полученное по указателю, стало "мусором", так как он показывал на переменную, а не на нулевой элемент массива.

```
#include <cstdio>

int main() {
 int x=0,y=1; int *px=&y;
 printf ("\nx=%d on &%p, y=%d on &%p, *px=%d on &%p",x,&x,y,&y,*px,px);
 x=(*px)++; //после первого запуска замените на x=*px++;
 printf ("\nx=%d on &%p, y=%d on &%p, *px=%d on &%p",x,&x,y,&y,*px,px);
 /*
 Действие (*px)++
 x=0 on &002CFC14, y=1 on &002CFC08, *px=1 on &002CFC08
 x=1 on &002CFC14, y=2 on &002CFC08, *px=2 on &002CFC08
 Действие *px++
 x=0 on &0021F774, y=1 on &0021F768, *px=1 on &0021F768
 x=1 on &0021F774, y=1 on &0021F768, *px=-858993460 on &0021F76C
 */
 getchar(); return 0;
}
```

Приведём пример связывания указателя со статическим массивом:

```
int a[5]={1,2,3,4,5};
int *pa=&a[0];
for (int i=0; i<5; i++) cout << *pa++ << " ";
```

или

```
for (int i=0; i<5; i++) cout << pa[i] << " ";
```

Эти записи абсолютно эквиваленты, потому что в Си конструкция `a[b]` означает не что иное, как `*(a+b)`, где `a` - объект, `b` – смещение от начала памяти, адресующей объект. Таким образом, обращение к элементу массива `a[i]` может быть записано и как `*(a+i)`, а присваивание указателю адреса нулевого элемента массива можно бы было записать в любом из 4 видов

```
int *pa=&a[0];
int *pa=&(*(a+0));
int *pa=&(*a);
int *pa=a;
```

Важно! При любом способе записи это одна и та же операция, и это - не "присваивание массива указателю", это его установка на нулевой элемент массива.

4. Сравнение указателей (вместо сравнения значений, на которые они указывают) в общем случае *может быть некорректно*!

```
int x;
int *px=&x, *py=&x;
if (*px==*py) ... //корректно
if (px==py) ...   //некорректно!
```

Причина – адресация ОП не обязана быть однозначной, например, в DOS одному адресу памяти могли соответствовать разные пары частей адреса "сегмент" и "смещение".

5. Указатели и ссылки могут использоваться для передачи функциям аргументов по адресу (то есть, для "выходных" параметров функций), для этого есть 2 способа:

Способ 1, со ссылочной переменной C++

```
void swap (int &a, int &b) {
 int c=a; a=b; b=c;
}
//...
int a=3,b=5; swap (a,b);
```

Этот способ можно назвать "передача параметров по значению, приём по ссылке".

Способ 2, с указателями Си

```
void swap (int *a, int *b) {
 int c=*a; *a=*b; *b=c;
}
//...
int a=3,b=5; swap (&a,&b);
int *pa=&a; swap (pa,&b);
```

Передача параметров по адресу, прием по значению.

Указатели и строки языка Си

Как правило, для сканирования Си-строк используются указатели.

```
char *s="Hello, world";
```

Это установка указателя на первый байт строковой константы, а не копирование и не присваивание!

Важно!

1. Даже если размер символа равен одному байту, эта строка займёт не 12 (11 символов и пробел), а 13 байт памяти. Дополнительный байт нужен для хранения нуль-терминатора, символа с кодом `0`, записываемого как `'\0'` (но не `'0'` – это цифра 0 с кодом 48). Многие функции работы с Си-строками автоматически добавляют нуль-терминатор в конец обрабатываемой строки:

```
char s[12];
strcpy(s,"Hello, world");
 //Вызвали стандартную функцию копирования строки
 //Ошибка! Нет места для нуль-терминатора
char s[13]; //А так было бы верно!
```

2. Длина Си-строки нигде не хранится, её можно только узнать стандартной функцией `strlen(s)`, где `s` – указатель типа `char *`. Для строки, записанной выше, будет возвращено значение 12, нуль-терминатор не считается. Фактически, Си-строка есть массив символов, элементов типа `char`.

Как выполнять другие операции со строками, заданными с помощью указателей `char *`? Для этого может понадобиться сразу несколько стандартных библиотек. Как правило, в новых компиляторах C++ можно подключать и "классические" си-совместимые заголовочные файлы, и заголовки из более новых версий стандарта, которые указаны в скобках.

Файл `ctype.h` (`cctype`) содержит:

1) функции с именами `is*` - проверка класса символов (`isalpha`, `isdigit`, ...), все они возвращают целое число, например:

```
char d;
if (isdigit(d)) {
 //код для ситуации, когда d - цифра
}
```

Аналогичная проверка "вручную" могла бы быть выполнена кодом вида

```
if (d>='0' && d<='9') {
```

2) функции с именами `to*` - преобразование регистра символов (`toupper`, `tolower`), они возвращают преобразованный символ. Могут быть бесполезны при работе с символами национальных алфавитов, а не только латиницей.

Модуль `string.h` (`cstring`) предназначен для работы со строками, заданными указателем и заканчивающимися байтом `'\0'` ("строками Си"). Имена большинства его функций начинаются на "str". Часть функций (`memcpy`, `memmove`, `memcmp`) подходит для работы с буферами (областями памяти с известным размером).

Примеры на работу со строками и указателями

1. Копирование строки

```
char *s="Test string";
char s2[80];
strcpy (s2,s);
 //копирование строки, s2 - буфер, а не указатель!
```

2. Копирование строки с указанием количества символов

```
char *s="Test string";
char s2[80];
char *t=strncpy (s2,s,strlen(s));
cout << t;
```

Функция `strncpy` копирует не более `n` символов (`n` - третий параметр), но не запишет нуль-терминатор, в результате чего в конце строки `t` выведется "мусор". Правильно было бы добавить после вызова `strncpy` следующее:

```
t[strlen(s)]='\0';
```

то есть, "ручную" установку нуль-терминатора.

3. Копирование строки в новую память

```
char *s="12345";
char *s2=new char [strlen(s)+1];
strcpy (s2,s);
```

Здесь мы безопасно скопировали строку `s` в новую память `s2`, не забыв выделить "лишний" байт для нуль-терминатора.

4. Приведём собственную реализацию стандартной функции `strcpy`:

```
char *strcpy_ (char *dst, char *src) {
 char *r=dst;
 while (*src!='\0') {
  *dst=*src; dst++; src++;
 }
 *dst='\0';
 return r;
}
```

Вызвать нашу функцию можно, например, так:

```
char *src="Строка текста";
char dst[80];
strcpy_ (&dst[0],&src[0]);
```

Сократим текст функции strcpy_:

```
char *strcpy_ (char *dst, char *src) {
 char *r=dst;
 while (*src) *dst++=*src++;
 *dst='\0';
 return r;
}
```

5. Сцепление строк – функция `strcat`

```
char *s="Test string";
char *s2;
char *t2=strcat (s2,strcat(s," new words"));
```

Так как `strcat` не выделяет память, поведение такого кода непредсказуемо!

А вот такое сцепление строк сработает:

```
char s[80]; strcpy (s,"Test string");
char s2[80];
strcat (s," new words");
strcpy (s2,s);
char *t2=strcat (s2,s);
```

То есть, *всегда должна быть память, куда писать* - статическая из буфера или выделенная динамически.

6. Поиск символа или подстроки в строке.

```
char *sym = strchr (s,'t');
if (sym==NULL) puts ("Не найдено");
else puts (sym); //выведет "t string"
//для strrchr вывод был бы "tring"
```

```
char *sub = strstr (s,"ring");
puts (sub); //выведет "ring"
```

7. Сравнение строк – функции с шаблоном имени `str*cmp` - "string comparing"

```
char *a="abcd",*b="abce";
int r=strcmp(a,b);
//r=-1, т.к. символ 'd' предшествует символу 'e'
//Соответственно strcmp(b,a) вернет в данном случае 1
//Если строки совпадают, результат=0
```

8. Есть готовые функции для разбора строк - `strtok`, `strspn`, `strcspn` - см. пособие, пп. 8.1-8.3

9. Преобразование типов между числом и строкой - библиотека `stdlib.h` (`cstdlib`)

```
char *s="qwerty";
int i=atoi(s);
 //i=0, исключений не генерируется!
```

Из числа в строку:

1) `itoa`, `ultoa` - из целых типов

```
char buf[20];
int i=-31189;
char *t=itoa(i,buf,36);
 //В buf получили запись i в 36-ричной с.с.
```

2) `fcvt`, `gcvt`, `ecvt` - из вещественных типов

Работа с динамической памятью

Как правило, описывается указатель нужного типа, который затем связывается с областью памяти, выделенной оператором `new` или си-совместимыми функциями для управления ОП.

1. Описать указатель на будущий динамический объект:

```
int *a;
 //Надёжнее int *a=NULL;
```

2. Оператором `new` или функциями `malloc`, `calloc` выделить оперативную память:

```
a = new int [10];
```

или

```
#include <malloc.h>
            //stdlib.h, alloc.h в разных компиляторах
//...
a = (int *) malloc (sizeof(int)*10);
```

или

```
a = (int *) calloc (10,sizeof(int));
```

В последнем случае мы выделили 10 элементов по `sizeof(int)` байт и заполнили нулями `'\0'`.

Важно! Не смешивайте эти 2 способа в одном программном модуле или проекте! Предпочтительней `new`, кроме тех случаев, когда нужно обеспечить заполнение памяти нулевыми байтами.

3. Проверить, удалось ли выделить память - если нет, указатель равен константе константе `NULL` из стандартной библиотеки (в ряде компиляторов `null`, `nullptr`, `0`):

```
if (a==NULL) {
 //Обработка ошибка "Не удалось выделить память"
}
```

4. Работа с динамическим массивом или строкой ничем не отличается от случая, когда они статические.

5. Когда выделенная ОП больше не нужна, её нужно освободить:

```
delete a; //Если использовали new
```

```
free (a); //Пытается освободить ОП,
          //если использовали malloc/calloc
```

Важно! Всегда старайтесь придерживаться принципа стека при распределении ОП. То есть, объект, занявший ОП последним, первым её освобождает.

Пример. Динамическая матрица размером n*m.

```
const int n=5,m=4;
int **a = new (int *) [n];
for (int i=0; i<n; i++) a[i] = new int [m];
```

После этого можно работать с элементами матрицы `a[i][j]`, например, присваивать им значения. Освободить память можно было бы так:

```
for (int i=n-1; i>-1; i--) delete a[i];
delete a;
```

Рекомендуемые задачи

1. Написать собственную функцию работы со строкой, заданной указателем, сравнить со стандартной.

2. Написать собственную функцию для работы с одномерным динамическим массивом, заданным указателем.

3. Написать свои версии функций преобразования строки в число и числа в строку.

Оглавление серии лекций

теги: textprocessing c++ учебное