Указатели

Строки в языках C/C++

Список литературы по курсу

1. Князев А.В. Основы языка C++. Учебное пособие. –М.: Издательство МЭИ, 2013 – 80 с. ISBN 978-5-7046-1425-8.

2. Князев А.В. Работа со сложными структурами данных на языке C++. Учебное пособие. –М.: Издательство МЭИ, 2015 – 48 с. ISBN 978-5-7046-1658-0.

3. Программирование. Сборник задач. Учебное пособие. –Санкт-Петербург: Лань, 2019 – 140 с. ISBN 978-5-8114-3857-0.

URL: https://e.lanbook.com/book/121485

Небезопасные функции ввода/вывода

В Microsoft Visual Studio

Что делать с «небезопасными» функциями fopen и fscanf?

Если компилятор Microsoft Visual Studio выдаёт сообщение, что функции *fopen* и *fscanf* небезопасны (unsafe), можно сделать следующее.

- Если вы создавали пустой проект, можно вставить следующую директиву препроцессора:

```
#define _CRT_SECURE_NO_WARNINGS
```

Вставить её необходимо в самое начало файла, перед всеми директивами #include.

Если вы создавали непустой проект, то придётся использовать новые функции – *fopen_s* и *fscanf_s*. Функция *fscanf_s* отличается от функции *fscanf* только тем, что при вводе строки первая функция требует указания размера памяти, выделенной для строки. При вводе данных других типов функции ничем не отличаются, так что можно просто заменить имя *fscanf* на *fscanf_s*.

- В отличие от функций ввода функции открытия файла *fopen* и *fopen_s* имеют различный синтаксис, поэтому строку

```
if ((file = fopen(fname, "r")) == NULL)
```

придётся заменить на следующие две строки

```
fopen_s(&file, fname, "r");
if (file == NULL)
```

- Можно отключить отслеживание устаревших функций в свойствах проекта. Но необходимо помнить, что сообщения компилятора являются важным инструментом отладки, и злоупотреблять этим способом нельзя. В норме необходимо корректировать программу так, чтобы компилятор не выдавал даже предупреждения, а отключать их можно только в крайнем случае. Для отключения нужно выбрать пункт меню *Проект – Свойства*, в появившемся диалоге раскрыть элемент *C/C++*, выбрать *Дополнительно* и отредактировать пункт *Отключить указанные предупреждения*. При этом появится новый диалог, в котором нужно ввести 4996 (номер отключаемого предупреждения), и сохранить изменения.

- В более новых версиях Microsoft Visual Studio использование fscanf – это уже не предупреждение, а ошибка. Её также можно отключить, пройдя по

Project -> Имя_проекта Properties -> Configuration Properties -> C|C++ -> справа в окне найти SDL Checks -> поставить No

Указатели и ссылки

В C/C++

Указатели

Каждая переменная, которую вы объявляете в программе, имеет **адрес** – номер ячейки памяти, в которой она расположена. Адрес является неотъемлемой характеристикой переменной. Можно объявить другую переменную, которая будет хранить этот адрес и которая называется **указателем**.

Указатели применяются при передаче в функцию параметров, которые мы хотим изменить, при работе с массивами, при работе с динамической памятью и в ряде других случаев.

Объявление указателя имеет следующий синтаксис:

*<тип> *<идентификатор>* [= *<инициализатор>*];

- Указатель может указывать на значения базового, перечислимого типа, структуры, объединения, функции, указателя.

- int *pi; // Указатель на int
- char *ppc; // Указатель на char
- int* p, s; // Плохой стиль объявления, *s* – не указатель!
- int *p, s; // Видно, что s – не указатель
- int *p, *s; // Два указателя

Операции над указателями

Существуют две операции, которые имеют отношение к работе с указателями. Этими операциями являются:

- операция взятия адреса (адресация) **&**;
- операция взятия значения по адресу (косвенная адресация или разыменование) *.

```
int a, *p;
p = &a; // Переменной p присваивается адрес переменной a
*p = 0; // Значение по адресу, находящемуся в переменной p
т.е. значение переменной a), становится равным 0
```

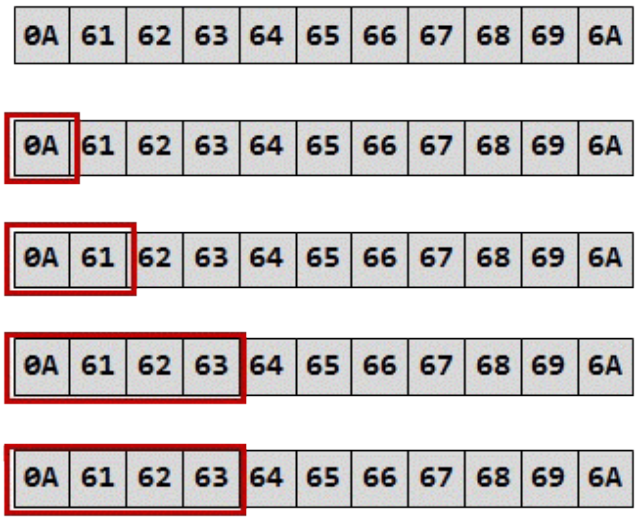
Адресная арифметика

Над указателями можно производить следующие действия:

- складывать указатель и целое число;
- вычитать из указателя целое число;
- вычитать из указателя указатель – получается целое число.

При этом прибавляемое/вычитаемое или полученное целое число обозначает не количество байт, а количество **элементов** того типа, на который указывает указатель, т.е. это число умножается или делится на размер типа.

Примеры



Вычитание указателей друг из друга определено только в том случае, если оба указателя указывают на элементы одного и того же массива (хотя язык не позволяет быстро проверить, так ли это).
Результатом вычитания одного указателя из другого будет количество элементов массива (целое число) между этими указателями.
В результате прибавления целого числа к указателю и вычитания целого числа из указателя получается новый указатель.

Если полученный таким образом указатель не указывает на элемент того же массива (или на элемент, следующий за последним), что и исходный указатель, то результат его использования не определён.

- Указатели можно использовать для обработки массивов.
  int a[100], n, *end, *p;
  end = a + n; // *n* - количество элементов массива *a*.

Имя массива является адресом его начала

// Таким образом, *end* - адрес элемента, находящегося после последнего элемента массива.

```
for (p = a; p < end; p++)
  printf("%4d", *p);
```

- Из того, что можно вычитать из указателя целое число, следует, что возможно использование отрицательных чисел в операции **[ ]**.

```
int a[N];
int *endA = a + N - 1, i;
for (i = 0; i < N; i++)
printf("%4d", endA[-i]);
```

Указатель на void

Специальное применение имеют указатели на тип **void**. Указатель на тип void может указывать на значения любого типа. Однако для выполнения операций над указателем на void либо над указуемым объектом, необходимо явно привести тип указателя к типу, отличному от указателя на void.

Указатель на объект любого типа можно присвоить переменной типа void*, один void* можно присвоить другому void*, пару void* можно сравнивать на равенство и неравенство, и, наконец, void* можно явно преобразовать в указатель на другой тип. Прочие операции могут оказаться опасными, потому что компилятор не знает, на какого сорта объект ссылается указатель на самом деле. Поэтому другие операции вызывают сообщение об ошибке на этапе компиляции. Чтобы воспользоваться void*, необходимо явно преобразовать его в указатель определённого типа.

Примеры правильного и неправильного использования

```
void f(int *pi)
{ void *pv = pi;          // Правильно – неявное
                          преобразование типа из
                          int* в void*
*pv;                      // Ошибка – нельзя
                          разыменовывать void*
pv++;                     // Ошибка – нельзя
                          произвести инкремент
                          void*
double *pd1 = pv;         // Ошибка
double *pd2 = pi;         // Ошибка
}
```

Как правило, не безопасно использовать указатель, преобразованный к типу, отличному от типа объекта, на который он указывает.

Основными применениями void * являются передача указателей функциям, которым не позволено делать предположения о типе объектов, а равно возврат объектов «неуточненного» типа из функций. Чтобы воспользоваться таким объектом, необходимо явно преобразовать тип указателя.

Функции, использующие указатели void *, обычно существуют на самых нижних уровнях системы, где происходит работа с аппаратными ресурсами. Наличие void * на более высоких уровнях подозрительно и, скорее всего, является индикатором ошибки на этапе проектирования.

Указатели на функции

С функцией можно проделать только две вещи: вызвать её и получить её адрес. Указатель, полученный взятием адреса функции, можно затем использовать для вызова функции.

```
void f(int x) { ... }
void (*pf)(int); // Указатель на функцию. Скобки обязательны!
 void g()
   { pf = &f; // pf указывает на функцию f
     pf(0); // Вызов функции f через указатель pf
   }
```

Компилятор распознаёт, что *pf* является указателем и вызывает функцию, на которую он указывает. То есть, разыменование указателя на функцию при помощи операции ***** необязательно. Аналогично, необязательно пользоваться операцией **&** для получения адреса функции.

Параметры указателей на функцию объявляются точно так же, как и параметры самих функций. При присваивании типы функций должны в точности совпадать.

```
typedef void (*PF)(int);
    // Для объявления типа «указатель на функцию» можно
    использовать объявление typedef
PF pf;
    // Объявляем сам указатель на функцию, используя
    предварительно определённый тип
void f1(int x) { ... }
int f2(int x) { ... }
void f3(char x) { ... }
    void f ()
{   pf = &f1; // Правильно
    pf = &f2; // Ошибка - не тот возвращаемый тип
    pf = &f3; // Ошибка - не тот тип параметра
}
```

Правила передачи параметров при вызове функций через указатель те же самые, что и при непосредственном вызове функций.

Указатель на указатель

Возможно объявление переменной, которая содержит адрес другой переменной, которая, в свою очередь, также является указателем. Такая переменная может быть необходима, если в функции нужно изменить адрес какого-либо объекта. Однако наличие более двух звёздочек в объявлении переменной говорит, скорее всего, о плохом проектировании.

```
int **ppi; // Объявляем указатель на указатель на целое
 void f(int **ppi)
  {
 int *pi = *ppi; // Указателю на целое присваивается
значение, хранящееся по адресу,
 ... // содержащемуся в указателе на указатель на целое
}
```

Ссылки

- Ссылка является альтернативным именем объекта. *Ссылка* – это объект, который синтаксически выглядит как переменная, а по семантике является адресом. Объявление ссылки, кроме случаев, когда ссылка является параметром функции, возвращаемым функцией-значением, должно содержать инициализатор.
- Далее все операции производятся не над самой ссылкой, а над тем объектом, на который она указывает.

```
int a = 10;
int &r = a; // Объявляем и инициализируем ссылку
r++;   // Значение переменной a становится 11
```

```
void f(double &a)
{ a += 3.14; }
….
 double d = 0;
f(d);  // d = 3.14

 int v[20];
 int& f(int i)
{ return v[i]; }

 f(3) = 7; // Элементу массива v[3] присваивается 7
```

На первый взгляд, ссылка является удобной заменой указателю, но она затрудняет понимание программы из-за несовпадения синтаксиса и семантики ссылки. Однако ссылки могут быть полезны для того, чтобы не передавать по значению (и не копировать) параметр функции, который имеет большой размер.

Строки в языках C/C++

Строка – это последовательность (массив) символов (типа char), которая заканчивается специальным символом – признаком конца строки. Это символ записывается как **'\0'** (не путайте с символом переноса строки **'\n'**) и равен 0.

При вводе строки символ конца строки добавляется автоматически. Все функции работы со строками – и стандартные, и создаваемые программистом – должны ориентироваться на этот символ.

Если требуется сформировать новую строку, то обязательно надо добавлять признак конца строки. Если этого не сделать, то при дальнейшей работе возникнут ошибки.

'a' // Символьная константа - один символ
“a” // Строковый литерал - массив из **двух** символов 'a' и '\0', заменяется на **адрес**.

```
char str[51]; // Объявление строки
char *str; // Нельзя, т.к. не выделяется память под элементы строки char *str = "abcd"; // Можно, но очень осторожно!
char *str1 = "abc", *str2 = "abc"; // Не известно, будет ли выполняться str1 == str2?
```

- *Строковым литералом* называется последовательность символов, заключённых в двойные кавычки. В строковом литерале на один символ больше, чем используется при его записи – добавляется символ '\0'.
- Тип строкового литерала есть «массив с надлежащим количество константных символов». Строковый литерал можно присвоить переменной типа char *. Это разрешается, потому что в предыдущих определениях C и C++ типом строкового литерала был char *. Однако изменение строкового литерала через такой указатель является ошибкой.

```
char *str = "C & C++"; str[2] = '?'; // Ошибка времени выполнения!
```

- То, что строковые литералы являются константами, не только является очевидным, но и позволяет при реализации произвести значительную оптимизацию методов хранения и доступа к строковым литералам. Если же нужна строка, которую можно модифицировать, следует объявить и инициализировать массив символов.

```
char str[] = "C & C++"; // Массив из 8 символов
 str[2] = '?'; // Правильно
```

- Поскольку мы не знаем, сколько в строке содержится символов, но знаем, что в конце стоит символ конца строки, цикл для обработки строки пишется следующим образом:

```
for (int i = 0; str[i] != '\0'; i++) { ... }.
```

Можно опустить сравнение с нулем, для C++ это будет эквивалентно: `for (int i = 0; str[i]; i++) { ... }.`

Можно использовать указатели для обработки строк:

```
char str[50], *p;
...
for (p = str; *p; p++) { ... }.
```

Стандартные функции работы со строками

Заголовки стандартных функций работы со строками хранятся в файле <string.h>.

Определение длины строки int strlen(const char *_str_);
– Сравнение строк int strcmp(const char *_str1_, const char *_str2_);
– Копирование char *strcpy(char *_str1_, const char *_str2_);
– Конкатенация строк char *strcat(char *_str1_, const char *_str2_);
– Поиск символа в строке char *strchr(const char *_str_, char _c_);
– Поиск подстроки char *strstr(const char *_str1_, const char *_str2_);

Ввод/вывод строки: Ввод строки до пробела или другого разделителя
– функция scanf с форматом *%s*; Ввод строки, содержащей пробелы
– char *gets (char *_buffer_); Ввод строки из файла, *n* задаёт максимальное количество символов для ввода
– char *fgets(char *_string_, int *n*, FILE *_stream_); Вывод строки с форматированием
– функция printf с форматом *%s*; Вывод строки
– int puts(const char *_string_); Вывод строки в файл
– int fputs(const char *_string_, FILE *_stream_);

Примеры работы со строками и указателями

- **Пример 1.** Функция, которая меняет все вхождения буквы «я» на «а», «а» – на «б», «б» – на «в» и т.д. Остальные символы остаются без изменения.

```
char* Change(char *str)
{
 char *p;
 for (p = str; *p; p++)
    if (*p == 'я') *p = 'а';
    else if ('а'<= *p && *p <= 'ю') (*p)++;
  return str;
}
```

- **Пример 2.** Функция, формирующая строку, состоящую из символов исходной строки, не входящих в заданное множество

```
char* NotEntered(char *dest, const char *source,
const char *symbols)
{
int i, j;
for (i = 0, j = 0; source[i]; i++)
  if (!strchr(symbols, source[i])) dest[j++] = source[i];
dest[j] = '\0'; // Обязательно добавляем признак
  конца строки в формируемую строку
 return dest;
}
```