В.Л. Тарасов

Лекции по программированию на C++

Лекция 10

Указатели

Понятие указателя широко используется в языке Си. В C++ указатели также применяются, но во многих случаях вместо указателей более удобно использовать ссылки, рассмотренные в §5.8.

10.1. Указатели и адреса

Указатели - это переменные, значениями которых являются *адреса* других переменных. При объявлении указателя используется символ звездочка (*):

```
ТИП* ИМЯ_УКАЗАТЕЛЯ;
```

Здесь `ТИП` это тип переменной, адрес которой может храниться в указателе. Например, следующая инструкция создает указатель `pd` на переменную типа `double`:

```
double* pd;          // pd – указатель на переменную типа double
```

Унарный оператор & выдает адрес своего операнда. После выполнения инструкций:

```
double d = 3.14159;  // d - переменная типа double
pd = &d;             // В указатель pd скопирован адрес переменной d
```

указатель `pd` будет содержать адрес переменной `d` (говорят, что `pd` *указывает* или *ссылается* на `d`). Схема расположения в памяти переменной `d` и указателя `pd`, содержащего ее адрес, показана на рис.10.1.

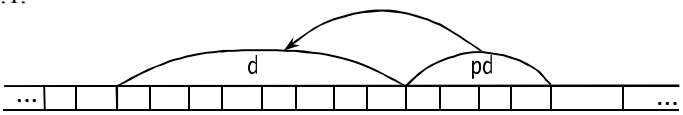

Рис. 10.1. Связь объекта и указателя на него

Переменная d типа double занимает 8 байтов памяти, а указатель pd – четыре. Такое количество памяти выделяется под указатели на 32 – разрядных компьютерах, в которых для формирования адресов используется 4 байта, состоящих из 8 двоичных разрядов – битов.

К указателям можно применять унарный оператор *, возвращающий объект, на который ссылается данный указатель, например,

```
*pd = 2.71828;        // Теперь d = 0
```

Иначе говоря, если pd указывает на d, то *pd и d – это одно и то же.

Можно создавать указатели на переменные любых типов.

Программа 10.1. Работа с указателями

В программе выполняются действия с указателями, о которых шла речь выше. Выводятся как значения самого указателя pd, так и переменной d, на которую он указывает.

```
#include <iostream>
#include <clocale>
#include <cstdlib>
using namespace std;

int main()
{
    setlocale(LC_ALL, "Russian");

    double* pd;                   // Указатель на double

    cout << "Адрес указателя    pd: &pd = " << &pd << endl;

    double d = 3.14159;           // Переменная типа double

    cout << "Адрес переменной    d: &d  = " << &d << endl;

    pd = &d;              // В указатель pd скопирован адрес переменной d

    cout << "Значение указателя pd: pd  = " << pd << endl;
    cout << "Значение переменной d: d   = " << d << endl;
    cout << "Значение переменной, на которую указывает pd: *pd = "
         << *pd << endl;

    *pd = 2.71828;        // Изменение переменной d через указатель на нее pd

    cout << "Значение переменной d: d   = " << d << endl;

    char* pc;             // Указатель на символ
    int*  pi;             // Указатель на целое

    cout << "Размеры указателей:\n";

    cout << "sizeof(char*)   = " << sizeof(pc) << "\n"
        << "sizeof(int*)     = " << sizeof(pi) << "\n"
        << "sizeof(double*)  = " << sizeof(pd) << "\n";
    system("pause");
```

```
    return 0;
}
```

Программа выводит:

```
Адрес указателя    pd: &pd = 001BF7AC
Адрес переменной    d: &d  = 001BF79C
Значение указателя pd: pd  = 001BF79C
Значение переменной d: d   = 3.14159
Значение переменной, на которую указывает pd: *pd = 3.14159
Значение переменной d: d   = 2.71828
Размеры указателей:
sizeof(char*)   = 4
sizeof(int*)    = 4
sizeof(double*) = 4
```

Видно, что для вывода значений указателей, то есть адресов памяти компьютера, используется по умолчанию шестнадцатеричная система счисления. Ее удобство состоит в том, что содержимое одного байта памяти представляется двузначным шестнадцатеричным числом. Так как значения указателей выводятся в виде набора 8 шестнадцатеричных цифр, ясно, что указатели занимают 4 байта памяти. Программа выводит размеры указателей трех типов: `char`, `int`, `double`, и все они равны 4 байта.

Программа показывает, что с переменной `d` можно работать напрямую, через имя переменной, и, используя выражение `*pd`, при условии, что в указателе `pd` хранится адрес переменной `d`. Таким образом, используя указатели, можно иметь косвенный доступ к переменным, на которые ссылаются указатели, и изменять значения этих переменных.

10.2. Указатели как аргументы функций

Если указатель на переменную является аргументом функции, то внутри функции становится известен адрес этой внешней по отношению к функции переменной, что позволяет работать с этой переменной, в том числе изменять ее значение.

Программа 10.2. Расчет треугольника

Пусть требуется вычислить периметр и площадь треугольника по трем его сторонам *a, b, c*. Напишем для этого функцию `triangle()`. Так как треугольник существует не для любых значений длин сторон, функция должна как-то информировать об этом. Пусть она будет возвращать `true`, если для заданных длин сторон треугольник существует, и `false`, если не существует. Две остальные величины –

периметр и площадь – будем возвращать из функции через аргументы, имеющее тип указателя.

Вычисления можно проводить по формулам:

$$P = a + b + c \text{ – периметр,}$$

$$p = P/2 \text{ – полупериметр,}$$

$$A = \sqrt{p(p-a)(p-b)(p-c)} \text{ – площадь.}$$

В main() вводятся исходные данные, и вызывается функция triangle().

```
// Файл Triangle.cpp

#include <iostream>
#include <cmath>
#include <clocale>
using namespace std;

// triangle: вычисление периметра и площади треугольника.
// Возвращает true, если треугольник существует и false если не существует

bool triangle(double a, double b, double c,
              double* p_perim, double* p_area)

// a, b, c - стороны треугольника
// p_perim - указатель на переменную для периметра
// p_area  - указатель на переменную для площади

{

// Проверка существования треугольника
    if(a > b + c || b > a + c || c > a + b)
        return false;        // Треугольник не существует, выход из функции

    double p = (a + b + c) / 2.0;// Полупериметр
    *p_perim = p * 2.0;      // Периметр
    *p_area = sqrt(p * (p - a) * (p - b) * (p - c));  // Площадь
    return true;
}
int main()
{
    setlocale(LC_ALL, "Russian");
    double r, s, t;                              // Стороны треугольника
    double P, A;                                 // Периметр и площадь
    cout << "Введите три стороны треугольника: ";
    cin >> r >> s >> t;
    if( triangle(r, s, t, &P, &A) == false )
        cout << "Такого треугольника не существует\n";
    else
        cout << "Периметр: " << P << ", площадь: " << A << "\n";
    system("pause");
```

```
    return 0;
}
```

Так как в функцию `triangle()` передается указатель `p_perim` на переменную для периметра, то внутри функции доступ к этой переменной получаем с помощью выражения `*p_perim` и присваиваем ей значение периметра. Аналогично используется указатель `p_area` на переменную для площади.

При вызове `triangle()` ей в качестве аргументов передаются `&P` и `&A` – адреса переменных `P` и `A`. Если возвращаемое `triangle()` значение равно `false`, выводится сообщение, что треугольник не существует, иначе выводятся значения периметра `P` и площади `A`, вычисленные внутри `triangle()`. Пример запуска программы:

```
Введите три стороны треугольника: 3 4 5
Периметр: 12, площадь: 6
```

Еще пример запуска программы:

```
Введите три стороны треугольника: 1 2 5
Такого треугольника не существует
```

Взаимодействие формальных параметров и фактических аргументов функции `triangle()` иллюстрируется рис.10.2.

Рис. 10.2. Параметры `a, b, c` получают значения внешних переменных `r, s, t`; параметры `p_perim` и `p_area` получают значения адресов `P` и `A`

При вызове функции `triangle()` формальные параметры `a, b, c` получают *значения* фактических аргументов `r, s, t`. Это символизируется стрелками, ведущими изнутри `r, s, t` внутрь `a, b, c`.

Размеры прямоугольников `p_perim`, `p_area` в два раза меньше, чем размеры прямоугольников `a, b, c`, так как `a, b, c` имеют тип `double` размером 8 байт, а указатели `p_perim`, `p_area` имеют размер 4 байта. То, что формальные параметры `p_perim`, `p_area` получают значения *адресов* внешних переменных `P` и `A`, показано стрелками, ведущими *от границ* прямоугольников `P` и `A` *внутрь* прямоугольников `p_perim` и `p_area`.

10.3. Указатели и массивы

Определение:

```
int a[10];
```

создает массив из 10 элементов, то есть блок из 10 расположенных последовательно переменных целого типа с именами `a[0]`, `a[1]`, …, `a[9]`.

Пусть определен указатель:

```
int *pi;
```

После присваивания

```
pi = &a[0];
```

указатель `pi` будет содержать адрес первого элемента массива `a`. На рис.10.3 это показано стрелкой, причем для `pi` нарисован прямоугольник, чтобы подчеркнуть, что этот указатель размещен где-то в памяти.

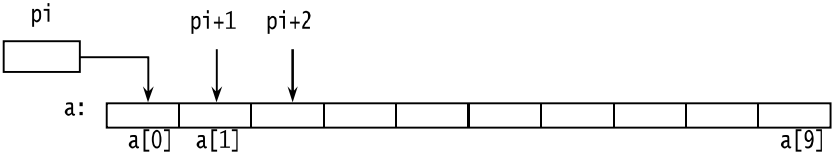

Рис. 10.3. Массив в памяти и указатели на элементы массива

По определению, `pi + 1` указывает на следующий элемент массива, `pi + i` указывает на `i`-й элемент после `pi`, `pi - i` указывает на `i`-й элемент перед `pi`. Для выражений `pi + 1`, `pi + 2` на схеме прямоугольники не нарисованы, так как это выражения, существующие только во время выполнения программы, отдельная память под них не выделяется.

Имея указатель на начало массива, можно получить доступ к любому его элементу, например, `*pi` есть первый элемент массива, `*(pi + 1)` – второй и т.д. Присваивание

```
*(pi + 1) = 0;       // a[1] = 0
```

обнуляет второй элемент массива, номер которого равен 1.

По определению, *имя массива* имеет значение *адреса первого элемента* массива, соответственно имя массива имеет тип *указателя на элемент массива*, например, `a` имеет тип `int*` (указатель на целое).

Доступ к `i` - му элементу массива можно получить, используя индексацию `a[i]` или выражение `*(a + i)`.

Указатель – это переменная, которой можно присваивать различные значения, например,

```
pi = a + 1;
```

Теперь pi указывает на второй элемент массива a.

Имя массива является *константой*, так как содержит адрес конкретного участка памяти, и записи типа a = pi; a++ недопустимы. Значение имени массива изменить нельзя, во всем остальном имя массива подобно указателю.

Теперь можно разобраться, почему, когда массив является аргументом функции, он не копируется внутрь функции. Пусть объявлена функция с аргументом – массивом:

```
void f(char s[]);
```

Так как имя массива – это указатель на первый элемент массива, то данное объявление эквивалентно такому:

```
void f(char* s);
```

Отсюда видно, что внутри функции создается *копия указателя* на первый элемент массива и принцип передачи аргументов *по значению* остается в силе.

10.4. Адресная арифметика

Как уже говорилось, указатели можно складывать и вычитать с целыми. Если p – указатель на некоторый элемент массива, то выражение p++ или ++p изменяет p так, чтобы он указывал на следующий элемент массива, а выражение p-- или --p переводит указатель на предыдущий элемент массива. Выражение p += i изменяет p так, чтобы он указывал на i - й элемент, после того, на который он указывал ранее.

Если указатели p и q указывают на элементы одного и того же массива, то к ним можно применять операторы сравнения: ==, !=, <, <=, >, >=. Выражение p < q истинно, если p указывает на более ранний элемент массива, чем q.

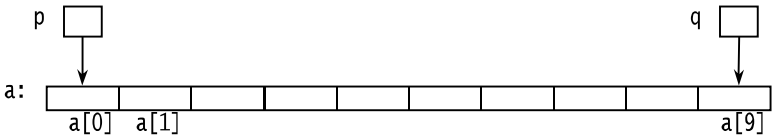

Рис. 10.4. Два указателя на элементы массива

Любой указатель можно сравнивать на равенство и неравенство с нулем. Если значение указателя равно нулю, это трактуется так, что указатель никуда не указывает.

Допускается вычитание указателей. Если p и q указывают на элементы одного и того же массива и p < q, то q - p есть число элементов от p до q. Для примера, показанного на рис.10.4, q - p равно 9.

Ниже приведена программа, работающая с массивами и указателями.

Программа 10.3. Массивы и указатели

```
// Файл ArraysAndPointers.cpp

#include <iostream>
#include <locale>
using namespace std;
int main()
{
    setlocale(LC_ALL, "Russian");
    int a[10];              // Массив
    int *p, *q;             // Два указателя
    int i;

    for (i = 0; i < 10; i++)
        a[i] = i;

    p = &a[0];  // Указатель на первый элемент массива, можно p = a;
    cout << "Значение имени массива a = " << a << endl
         << "Значение указателя     p = " << p << endl;

    cout << "Элементы массива:\n";

    cout << "Использование индексации:\n";
    for (i = 0; i < 10; i++)
        cout << a[i] << "  ";

    cout << "\nИспользование имени массива:\n";
    for (i = 0; i < 10; i++)
        cout << *(a + i) << "  ";

    cout << "\nИспользование указателя:\n";
    for (i = 0; i < 10; i++)
        cout << *(p + i) << "  ";
    q = &a[9];              // Указатель на последний элемент массива

    cout << "\np = " << p << ", q = " << q << ", q - p = " << (q - p);
    cout << "\nint(q) - int(p) = " << (int(q) - int(p));
    ++p;  --q;              // Изменение указателей
    cout << "\n++p = " << p << ", --q = " << q << ", q - p = " << q - p;

    cin.get();
    return 0;
```

```
}
```

Программа выводит:

```
Значение имени массива a = 00E6F91C
Значение указателя     p = 00E6F91C
Элементы массива:
Использование индексации:
0  1  2  3  4  5  6  7  8  9
Использование имени массива:
0  1  2  3  4  5  6  7  8  9
Использование указателя:
0  1  2  3  4  5  6  7  8  9
p = 00E6F91C, q = 00E6F940, q - p = 9
int(q) - int(p) = 36
++p = 00E6F920, --q = 00E6F93C, q - p = 7
```

Значения указателей выводятся в шестнадцатеричном виде. Для вывода указателей в десятичной системе можно преобразовать их к целому типу с помощью выражений вида `int(p)`.

Видно, что указатели содержат адреса байтов памяти, начиная с которых расположены элементы массива. При увеличении или уменьшении указателя, адрес, хранящийся в указателе, изменяется с учетом размера объектов, на которые указывает указатель. После действия `++p` значение `p` увеличивается на 4, то есть на размер целого `int`. Аналогично, после `--q` значение указателя `q` уменьшается на 4. Разность указателей `q - p` вычисляется также с учетом размера объектов по формуле:

```
(значение(q) - значение (p)) / размер(int).
```

10.5. Массивы как аргументы функций

Как было сказано, аргументы передаются в функции по значению, поэтому нельзя изменить значения аргументов внутри функции. Это не относится к массивам. В случае с массивами в функцию передается *адрес* первого элемента массива. Элементы массива не копируются внутрь функции. Используя адрес первого элемента массива, можно внутри функции получить доступ к любому элементу массива и изменить его. Таким образом, *если массив является аргументом функции, его элементы можно изменить внутри функции*.

Программа 10.4. Сортировка массива

Сортировкой называется упорядочение массива по возрастанию или убыванию. Массивы часто используются для хранения и обработки больших объемов информации. Работа с упорядоченными массивами

выполняется, как правило, быстрее, поэтому сортировка массивов – это важная задача.

В программе надо выполнить действия:

1) заполнить массив значениями;
2) вывести исходный массив;
3) отсортировать массив;
4) вывести упорядоченный массив.

Заполнение, печать и сортировку массива реализуем в виде отдельных функций.

Массив для сортировки и его максимальный размер объявляются в начале программы как внешние переменные, а определяются в ее конце. Размер массива задается константой `SIZEARR` достаточно большого размера, чтобы выделенной под массив памяти хватило в большинстве случаев использования программы. Конкретный размер массива вводится пользователем и проверяется в программе на допустимость. При вводе недопустимого размера массива вызывается библиотечная функция

```
void exit(int k),
```

которая завершает работу программы и передает в вызывающую программу значение своего аргумента `k`. Эта функция объявлена в `stdlib.h`.

Массив `x` заполняется случайными числами функцией:

```
void get_array(int x[], int n);
```

Случайные целые числа из диапазона от `0` до `RAND_MAX` генерируется функцией стандартной библиотеки

```
int rand(),
```

объявленной в `cstdlib`. Величина `RAND_MAX` определена в файле `cstdlib`. Обычно это `32767`.

Функция

```
void srand(unsigned int seed);
```

настраивает генератор случайных чисел, используя для формирования первого псевдослучайного числа параметр `seed`. Чтобы получать при каждом запуске программы разную последовательности случайных чисел, функцию `srand()` следует вызывать каждый раз с разными аргументами. В программе функции `srand()` передается число секунд, прошедших от 1 января 1970 г., возвращаемое функцией `time(0)`. Эта функция объявлена также в `cstdlib`.

Для сортировки массива по возрастанию написана функция:

```
void bubble_sort(int x[], int n);
```

использующая *алгоритм «пузырька»*, состоящий в следующем. Организуется проход по массиву, в котором сравниваются соседние элементы. Если предшествующий элемент оказывается больше следующего, они и меняются местами. В результате первого прохода наибольший элемент оказывается на своем, последнем месте (*«всплывает»*). Затем проход по массиву повторяется до предпоследнего элемента, затем до третьего с конца массива и т.д. В последнем проходе по массиву сравниваются только первый и второй элементы.

```
// Файл SortArr.cpp
// Сортировка массива

#include <iostream>
#include <cstdlib>                    // Для exit(), rand()
using namespace std;

// Объявления функций

void get_array(int[], int n);         // Заполнение массива из n элементов
void bubble_sort(int[], int n);       // Сортировка массива методом пузырька
void prn_array(int[], int n);         // Вывод массива

int main()
{
    setlocale(LC_ALL, "Russian");
    const int SIZEARR = 500;          // Максимальный размер массива
    int x[SIZEARR];                   // Массив
    int n;                            // Размер массива
    cout << "Введите размер массива < " << SIZEARR << ": ";
    cin >> n;
    if(n > SIZEARR || n <= 0){        // Проверка размера
        cout << "Размер массива " << n
             << " недопустим " << endl;
        system("pause");
        exit(1);                      // Завершение программы
    }
    get_array(x, n);                  // Заполнение массива
    cout << "Исходный массив: \n";
    prn_array(x, n);                  // Вывод исходного массива
    bubble_sort(x, n);                // Сортировка
    cout << "\nОтсортированный массив: \n";
    prn_array(x, n);                  // Вывод упорядоченного массива
    cout << endl;
    system("pause");
    return 0;
}

// Определение функций
```

```
// get_array: заполняет массив y случайными числами
#include <time.h>
void get_array(int y[], int n)
{
    srand((unsigned) time(NULL));// Инициализация генератора случайных чисел
    for(int i = 0; i < n; i++)
        y[i] = rand();           // rand() генерирует целое случайное число
}

// prn_array: вывод массива
void prn_array(int y[], int n)
{
    for(int i = 0; i < n; i++)
        cout << y[i] << " ";
}

// bubble_sort: сортировка массива y методом пузырька
void bubble_sort(int y[], int n)
{
    for(int i = n - 1; i > 0; i--)    // i задает верхнюю границу
        for(int j = 0; j < i; j++)    // Цикл сравнений соседних элементов
            if(y[j] > y[j + 1]){      // Если нет порядка,
                int tmp = y[j];       // перестановка местами
                y[j] = y[j + 1];      // соседних
                y[j + 1] = tmp;       // элементов
            }
}
```

Далее приведены результаты двух запусков программы. Первый запуск:

```
Введите размер массива < 500: 666
Размер массива 666 недопустим
```

Второй запуск:

```
Введите размер массива < 500: 12
Исходный массив:
10513 15931 17912 19330 19790 5792 31927 14775 21659 28468 31432 11871
Отсортированный массив:
5792 10513 11871 14775 15931 17912 19330 19790 21659 28468 31432 31927
```

Обратим внимание, что для обмена значений двух элементов массива используется специальная промежуточная переменная `tmp`.

Как было сказано в §10.3, имя массива есть константный указатель на начало массива. Это можно иметь в виду при написании функций, аргументами которых являются массивы. Например, функцию `get_array()` можно переписать в виде:

```
void get_array(int* y, int n)
{
    srand((unsigned) time(NULL));// Инициализация генератора случайных чисел
    for(int i = 0; i < n; i++)
```

```
        *(y + i) = rand();       // rand() генерирует целое случайное число
}
```

Для обращения к i-му элементу массива можно использовать выражение *(y + i) или y[i].

10.6. Символьные указатели

Для работы со строками символов часто используются указатели на char. Их определение имеет вид:

```
char* pc;
```

Строковая константа, написанная в виде "Я строка", есть массив символов с нулевым символом '\0' на конце. Адрес начала массива, в котором расположена строковая константа, можно присвоить указателю:

```
pc = "Я строка";
```

Здесь копируется только *адрес* начала строки, сами символы строки не копируются.

Указатель на строку можно использовать там, где требуются строки. Например, при выполнении следующей инструкции:

```
cout << pc << " длиной " << strlen(pc) << " символов";
```

будет напечатано:

```
Я строка длиной 8 символов
```

При подсчете символов строки учитываются и пробелы, а завершающий символ '\0' не учитывается.

В отличие от указателей других типов, при выводе символьных указателей выводится *не адрес*, хранящийся в указателе, *а строка символов*, на которую указывает указатель.

10.7. Массивы указателей

Указатели, как и любые другие переменные, можно группировать в массивы. Удобно, например, использовать массив символьных указателей при работе с несколькими строками, которые могут иметь различную длину.

Программа 10.5. Названия месяцев

Программа предлагает ввести номер месяца и выводит его название. Для доступа к строкам с названиями месяцев используется массив символьных указателей pmn.

```
// Файл NamesOfMonths.cpp

// Использование массива указателей
// Программа выводит название месяца по его номеру

#include <iostream>
#include <clocale>
using namespace std;

int main()
{
    setlocale(LC_ALL, "Russian");
    char* pmn[] = {                          // Массив указателей
                "Неверный номер месяца", "Январь", "Февраль", "Март",
                "Апрель", "Май", "Июнь", "Июль", "Август",
                "Сентябрь", "Октябрь", "Ноябрь", "Декабрь"
    };
    int month;                               // Номер месяца
    cout << "Введите номер месяца: ";
    cin >> month;
    if (0 < month && month <= 12)
        cout << "Это " << pmn[month] << endl;
    else
        cout << pmn[0] << endl;
    system("pause");
    return 0;
}
```

Массив pmn[], элементами которого являются символьные указатели, инициализируется строковыми константами с названиями месяцев. Сами строки размещены где-то в памяти, а их адреса запоминаются в указателях массива pmn с помощью инициализации. Далее приведены результаты двух запусков программы.

Первый запуск:

```
Введите номер месяца: 13
Неверный номер месяца
```

Второй запуск:

```
Введите номер месяца: 11
Это Ноябрь
```

10.8. Операторы new и delete

Оператор new выделяет память под объект во время выполнения программы. Например, пусть в программе определен указатель:

```
double* pd;       // Указатель на double
```

Значение указателя pd после его создания не определено и его использование будет ошибкой. Инструкция:

```
pd = new double;
```

выделяет память под переменную типа double, адрес которой присваивается pd. После оператора new указывается *тип* создаваемого объекта. Теперь динамически созданную переменную можно использовать, например:

```
*pd = sqrt(3.0);      // Размещение в памяти значения
cout << *pd;          // Печать значения
```

Оператор delete освобождает память, выделенную ранее оператором new, например,

```
delete pd;
```

Теперь указатель pd можно использовать для других целей, а память, освобожденная оператором delete, может быть повторно использована под объекты, создаваемые оператором new.

Динамические массивы можно создавать оператором new[]. Например, массив из 80 символов можно создать инструкцией:

```
char* s = new char[80];  // Создается динамический массив из 80 символов
```

Для удаления динамических массивов служит оператор delete[], например,

```
delete[] s;  // Освобождение памяти, на которую указывает s
```

При освобождении памяти, выделенной оператором new, операторы delete и delete[] должны иметь возможность определять размер удаляемого объекта. Это обеспечивается тем, что под динамический объект памяти выделяется больше, чем под статический, обычно на одно слово, в котором хранится размер объекта.

С помощью оператора new[] можно создавать массивы, размер которых определяется в ходе работы программы.

Программа 10.6. Выделение и освобождение памяти

Демонстрируется выделение и освобождение памяти. Для наблюдения за адресами использован оператор вывода <<, который по умолчанию выводит адреса как целые десятичные числа.

```
// Файл AllocMem.cpp

#include <iostream>
#include <clocale>
#include <cstdlib>
#include <ctime>
#include <cmath>
using namespace std;

// get_array: заполнение массива y случайными числами

void get_array(int y[], int n)    // n – размер массива
{
    for (int i = 0; i < n; i++)
        y[i] = rand();            // rand() генерирует целое случайное число
}

// prn_array: вывод массива

void prn_array(int y[], int n)
{
    for (int i = 0; i < n; i++)
        cout << y[i] << " ";
}

int main()
{
    setlocale(LC_ALL, "Russian");
    double* pd = 0;                         // Указатель на double
    pd = new double;                        // Выделение памяти
    cout << "Адрес(pd) = " << pd << ", значение(*pd) = " << *pd;
    *pd = sqrt(3.0);                        // Занесение в память значения
    cout << "\nАдрес(pd) = " << pd << ", значение(*pd) = " << *pd;
    delete pd; pd = 0;                      // Освобождение памяти
    int *pi_1 = 0, *pi_2 = 0, *pi_3 = 0;  // Указатели на целое
    int size1, size2, size3;                // Размеры динамических массивов

    cout << "\nВведите размер массива 1: ";
    cin >> size1;
    pi_1 = new int[size1];                  // Выделение памяти под массив 1
    srand(time(0));                 // Инициализация генератора случайных чисел
    get_array(pi_1, size1);                 // Заполнение массива 1
    cout << "Массив 1\n";
    prn_array(pi_1, size1);                 // Вывод массива 1

    cout << "\nВведите размер массива 2: ";
    cin >> size2;
    pi_2 = new int[size2];                  // Выделение памяти под массив 2
    get_array(pi_2, size2);                 // Заполнение массива 2
```

```
    cout << "Массив 2\n";
    prn_array(pi_2, size2);              // Вывод массива 2

    pi_3 = new int[size1 + size2];       // Память для составного массива
    int k;                               // Индекс для массива 3
    for (k = 0; k < size1; ++k)          // Копируем массив 1 в составной
        pi_3[k] = pi_1[k];
    for (int i = 0; i < size2; ++i, ++k) // Копируем массив 2 в составной
        pi_3[k] = pi_2[i];

    cout << "\nМассив 1 + Массив 2:\n";
    prn_array(pi_3, size1 + size2);      // Вывод составного массива
    cout << endl;

    delete[] pi_1; pi_1 = 0;             // Удаление
    delete[] pi_2; pi_2 = 0;             // динамических
    delete[] pi_3; pi_3 = 0;             // массивов
    system("pause");
    return 0;
}
```

Программа выводит следующее:

```
Адрес(pd) = 006F9EC0, значение(*pd) = -6.27744e+66
Адрес(pd) = 006F9EC0, значение(*pd) = 1.73205
Введите размер массива 1: 5
Массив 1
12960 19674 16445 29262 27
Введите размер массива 2: 7
Массив 2
20642 32629 10039 4847 26564 7802 8994
Массив 1 + Массив 2:
12960 19674 16445 29262 27 20642 32629 10039 4847 26564 7802 8994
```

Из полученных результатов видно, что содержимое динамической памяти после ее выделения для программы заполнено неким «мусором», поэтому нельзя допускать использования неинициализированной динамической памяти. Также нельзя использовать указатели, которые не указывают на действительно выделенную память. Некоторой страховкой от использования неинициализированных указателей является присваивание им нулевого значения, так как при обращении по нулевому адресу возникает ошибка времени выполнения.

Утечка памяти

При работе с динамической памятью следует быть внимательным, чтобы не допустить ситуации, когда память выделяется динамически, но не освобождается, когда необходимость в ней отпадает, что приводит к уменьшению доступной свободной памяти. Это явление называют

«утечка памяти». Пример этого явления демонстрирует следующая программа.

Программа 10.7. Пример утечки памяти

В программе в бесконечном цикле вызывается функция `f()`, которая при каждом вызове создает массив из 10 миллионов целых чисел, то есть запрашивает 4 10^7 байт памяти.

```
// Файл MemoryLeak.cpp

#include <iostream>
using namespace std;
void f()
{
    // Создание массива из 10 миллионов целых
    int* pi = new int[1024 * 1024 * 1024];
}

int main()
{
    for(int i = 1; ; ++i){   // Бесконечный цикл
        f();
        cout << i << ' ';
    }
}
```

Программа выводит:

```
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

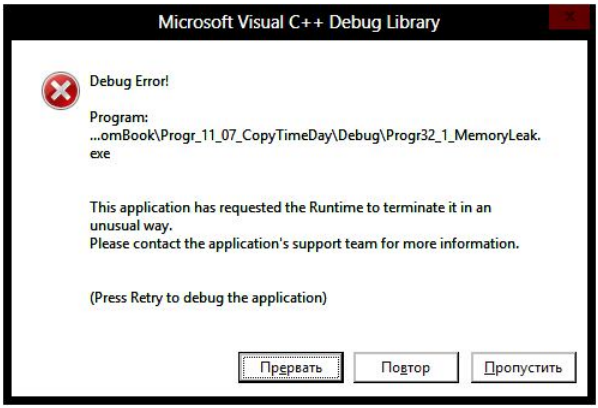

Рис. 10.5. Сообщение о необычном завершении программы

После чего выполнение программы было прекращено операционной системой с выдачей сообщения (рис.10.5). Программе удалось создать

50 массивов по 10^7 целых в каждом, то есть всего программа получила $50 \cdot 10^7 \cdot 4$ байт = $2 \cdot 10^9$ байт = 2 ГБ. Обозначение ГБ используется как сокращение для термина «гигабайт», значение которого рано 10^9 байт. Эта единица используется торговлей при продаже компьютерной техники.

Следует иметь в виду, что захват одной программой большого объема памяти может замедлить выполнение других программ.

10.9. Указатели на структуры

Можно создавать указатели на переменные любых типов, в том числе и на структуры. Например, в программе 9.4 была создана структура для моделирования времени суток:

```
struct TimeDay{          // Структура для моделирования времени суток
    int hour;            // Часы суток
    int min;             // Минуты
    void Set(int hh, int mm)         // Установка времени
    { hour = hh; min = mm; }
    void AddHour(int nh);            // Добавить nh часов
    void AddMin(int nm);             // Добавить nm минут
    void Print();                    // Вывод времени
    TimeDay Difference(const TimeDay& dt);    // Длительность промежутка
                                              // времени
};
```

Пусть создана переменная-структура:

```
TimeDay EndWork;
```

Создадим указатель на структуру и присвоим ему адрес структуры `EndWork`:

```
TimeDay* pew =  &EndWork;
```

Теперь со структурой `EndWork` можно работать или непосредственно:

```
EndWork.hour = 17; EndWork.min = 30;
```

или через указатель на нее:

```
(*pew).hour = 17; (*pew).min = 30;                    // (1)
```

Так как `pew` – указатель на структурную переменную `EndWork`, то выражение `*pew` есть сама эта переменная. В инструкциях (1) скобки необходимы, так как приоритет оператора «точка» (`.`) выше приоритета оператора «звездочка» (`*`). Без скобок выражение вида `*pew.hour` рассматривается как `*(pew.hour)`, что является ошибкой, так как `pew` является не структурой, а указателем, к которому нельзя применять оператор «точка».

Есть специальный оператор «стрелка» (->) для обращение к членам структуры через указатель на структуру. С его помощью присвоить членам структуры EndWork значения можно так:

```
pew->hour = 17; pew->min = 30;                    // (2)
```

Обращение к членам структуры через указатель на структуру с помощью оператора -> записываются более кратко и оставляют меньше возможности сделать ошибку.

Через указатель на структуру можно обращаться не только к данным-членам структуры, но и к функциям-членам структуры. Например, задать значение структуре EndWork можно инструкцией:

```
(*pew).Set(17, 30);
```

Или с помощью инструкции:

```
pew->Set(17, 30);
```

Рассмотрим пример программы, где использованы указатели на структуры.

Программа 10.8. Структура для времени и указатели

В заголовочном файле TimeDayPointer.h объявлена структура для моделирования времени суток и функции для работы с ней. Добавлены функции, использующие указатели на структуры.

```
// Файл TimeDayPointer.h

#ifndef TimeDayPointerH
#define TimeDayPointerH

#include <iostream>
#include <ctime>
#include <cstdlib>
using namespace std;

struct TimeDay{          // Структура для моделирования времени суток
    int hour;            // Часы суток
    int min;             // Минуты часа
    void Set(int hh, int mm)         // Установка времени
    {hour = hh; min = mm;}
    void AddHour(int nh);            // Добавить nh часов
    void AddMin(int nm);             // Добавить nm минут
    void Print();                    // Вывод времени
    TimeDay Difference(const TimeDay& dt);    // Длительность промежутка
                                              // времени
    void Add(TimeDay* ptd);          // Увеличить на время,
                                     // на которое указывает ptd
    bool Earlier(TimeDay* ptd);      // Раньше ли момент времени того,
                                     // на который указывает ptd
```

```
};
    void Input(TimeDay* ptd);         // Ввод времени в структуру,
                                      // на которую указывает ptd
#endif
```

Функции-члены структуры определяются в следующем файле.

```
// Файл TimeDayPointer.cpp

#include "TimeDayPointer.h"

void TimeDay::AddHour(int nh)         // Добавить nh часов
{
    hour = (hour + nh) % 24;
}

void TimeDay::AddMin(int nm)          // Добавить nm минут
{
    hour = (hour + (min + nm) / 60) % 24;
    min = (min + nm) % 60;
}

void TimeDay::Print()                 // Вывод времени
{
    if(hour < 10)
        cout << '0';
    cout << hour << ':';
    if(min < 10)
        cout << '0';
    cout << min << " ";
}

// Difference: возвращает длительность промежутка времени
// от заданного в структуре до td

TimeDay TimeDay::Difference(const TimeDay& td)
{
    int m1 = hour * 60 + min;         // Число минут от начала суток
                                      // до первого момента времени
    int m2 = td.hour * 60 + td.min;   // Число минут
    // от начала суток до td2
    if(m2 < m1){                      // Момент времени td был раньше
        int m = m1;                   // Обмен значений m1 и m2
        m1 = m2; m2 = m;
    }
    TimeDay tmp;                      // Длительность промежутка времени
    tmp.hour = (m2 - m1) / 60;        // Число часов в промежутке времени
    tmp.min  = (m2 - m1) % 60;        // Число минут в промежутке времени
    return tmp;
}

// Add: увеличить время на указываемое ptd
void TimeDay::Add(TimeDay* ptd)
{
    AddHour(ptd->hour); AddMin(ptd->min);     // Добавляем часы и минуты
    // Можно:
```

```
    // AddHour((*ptd).hour); AddMin((*ptd).min);
}

// Earlier: проверяет, раньше ли первый момент времени,
// чем момент, на который указывает ptd

bool TimeDay::Earlier(TimeDay* ptd)
{
    if(hour < ptd->hour ||                    // Часы меньше
      (hour == ptd->hour && min < ptd->min))  // Часы равны, минуты меньше
        return true;
    return false;
}

// Input: ввести время в переменную, на которую указывает ptd
void Input(TimeDay* ptd)
{
    cin >> ptd->hour >> ptd->min;
    // Можно:
    // cin >> (*ptd).hour >> (*ptd).min;
}
```

В главной функции задается и выводится время окончания рабочего дня, определяется время до обеденного перерыва.

```
// Файл UseTimeDayPointer.cpp

#include "TimeDayPointer.h"

int main()
{
    setlocale(LC_ALL, "Russian");

    TimeDay EndWork;                      // Структура

    TimeDay* pew =  &EndWork;             // Указатель на структуру
    pew->Set(17, 30);                     // Можно: (*pew).Set(17, 30);
    cout << "Окончание рабочего дня: ";
    pew->Print();

    cout << "\nВведите текущее время: ";
    TimeDay Curr;                         // Текущее время
    Input(&Curr);                         // Ввод текущего времени
    TimeDay Dinner;                       // Время обеда

    cout << "Введите время обеда: ";
    Input(&Dinner);                       // Ввод времени обеда
    TimeDay ToDin;                        // Время до обеда

    if(Curr.Earlier(&Dinner) == false)
        cout << "Время обеда прошло\n";
    else{
        ToDin = Curr.Difference(Dinner);
        cout << "До обеда осталась "; ToDin.Print();
        cout << endl;
    }
```

```
    system("pause");
    return 0;
}
```

Первый запуск программы:

```
Окончание рабочего дня: 17:30
Введите текущее время: 11 15
Введите время обеда: 12 00
До обеда осталась 00:45
```

Второй запуск программы:

```
Окончание рабочего дня: 17:30
Введите текущее время: 14 15
Введите время обеда: 12 00
Время обеда прошло
```