Оглавление

Многомерные динамические массивы

И, снова в бой! Мы с Вами уже сталкивались с динамическими массивами, однако нам бы хотелось коснуться этой темы еще раз и рассказать вам кое-что о создании многомерных динамических массивов.

Многомерный массив в С по своей сути одномерен. Операции new и delete позволяют создавать и удалять динамические массивы, поддерживая при этом иллюзию произвольной размерности. Деятельность по организации динамического массива требует дополнительного внимания, которое окупается важным преимуществом: характеристики массива (операнды операции new) могут не быть константными выражениями. Это позволяет создавать многомерные динамические массивы произвольной конфигурации. Следующий пример иллюстрирует работу с динамическими массивами.

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

 void main()
 {
        int i, j;

         // Переменные для описания характеристик массивов.
         int m1 = 5, m2 = 5;

         /*
              Организация двумерного динамического массива производится в два этапа.
          Сначала создаётся одномерный массив указателей, а затем каждому элементу
              этого массива присваивается адрес одномерного массива. Для характеристик
               размеров массивов не требуется константных выражений.
         */
         int **pArr = new int*[m1];
         for (i = 0; i < m1; i++)
               pArr[i] = new int[m2];

         pArr[3][3] = 100;
         cout << pArr[3][3] << "\n";



        //Последовательное уничтожение двумерного массива…

         for (i = 0; i < m1; i++)
               delete[]pArr[i];
         delete[]pArr;
}
```

Примеры на многомерные динамические массивы

Пример 1. Организация двумерного "треугольного" динамического массива.

Сначала создаётся одномерный массив указателей, а затем каждому элементу этого массива присваивается адрес одномерного массива. При этом размер (количество элементов) каждого нового массива на единицу меньше размера предыдущего. Включённая в квадратные скобки переменная, которая является операндом операции new, позволяет легко сделать это.

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

void main()
{
        int i, j;

        // Переменные для описания характеристик массивов.
        int m1 = 5, wm = 5;
        int **pXArr = new int*[m1];

        for (i = 0; i < m1; i++, wm--)
                pXArr[i] = new int[m1];


        //Заполнение массива нулями и показ его на экран
        for (i = m1 - 1; i >= 0; i--, wm++) {
                for (j = 0; j < wm; j++){
                        pXArr[i][j]=0;
                        cout<<pXArr[i][j]<<"\t";
                }
                cout<<"\n\n";
        }

         /* Последовательное уничтожение двумерного массива
треугольной конфигурации */

        for (i = 0; i < m1; i++)
                delete[]pXArr[i];
        delete[]pXArr;
}
```

Пример 2. Организация трехмерного динамического массива.

Создание и уничтожение трёхмерного массива требует дополнительной итерации. Однако здесь также нет ничего принципиально нового.

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

void main()
```

```
{
        int i, j;

        // Переменные для описания характеристик массивов.
        int m1 = 5, m2 = 5, m3 = 2;

        // указатель на указатель на указатель :)
        int ***ppArr;

        // Создание массива
        ppArr = new int**[m1];
        for (i = 0; i <m1; i++)
                ppArr[i] = new int*[m2];

        for (i = 0; i < m1; i++)
                for (j = 0; j < m2; j++)
                        ppArr[i][j] = new int[m3];

        ppArr[1][2][3] = 750;
        cout << ppArr[1][2][3] << "\n";

        // Удаление в последовательности, обратной созданию
        for (i = 0; i < m1; i++)
                for (j = 0; j < m2; j++)
                        delete[]ppArr[i][j];

        for (i = 0; i < m1; i++)
                delete[]ppArr[i];
        delete[] ppArr;
}
```

Перечисляемые типы

Перечислимый тип вводится ключевым словом enum и задает набор значений, определяемый пользователем. Набор значений заключается в фигурные скобки и является набором целых именованных констант, представленных своими идентификаторами. Эти константы называются перечислимыми константами. Рассмотрим объявление:

```
enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES};
```

С его помощью создается целочисленный тип набором из четырех названий мастей, именующих целочисленные константы. Перечислимые константы - это идентификаторы CLUBS, DIAMONDS, HEARTS и SPADES, имеющие значения - 0, 1, 2 и 3, соответственно. Эти значения присвоены по умолчанию. Первой перечислимой константе присваивают постоянное целое численное значение 0. Каждый последующий член списка на единицу больше, чем его сосед слева. Переменным типа Suit, определенного пользователем, может быть присвоено только одно из четырех значений, объявленных в перечислении

Другой популярный пример перечислимого типа:

```
    enum Months {JAN = 1, FEB, MAR, APR, MAY, YUN, YUL AUG, SEP,
OCT, NOV, DEC};
```

Это объявление создает определенный пользователем тип Months с константами перечисления, представляющими месяцы года. Поскольку первое значение приведенного перечисления установлено равным 1, оставшиеся значения увеличиваются на 1 от 1 до 12.

В объявлении перечислимого типа любой константе перечисления можно присвоить целое значение.

Примечание: Типичная ошибка. После того, как константа перечисления определена, попытка присвоить ей другое значение является синтаксической ошибкой.

Основные моменты использования перечислений.

1. Использование перечислений вместо целых констант облегчает чтение программы.

2. Идентификаторы в enum должны быть уникальными, но отдельные константы перечисления могут иметь одинаковые целые значения.

3. Набор идентификаторов перечислимого типа — собственный уникальный тип, отличающийся от других целочисленных типов.

4. Перечислимые константы могут определяться и инициализироваться произвольными целочисленными константами, а также константными выражениями:

```
    enum ages (milton = 47, ira, harold = 56, philip = harold + 7};
```

Примечание: Обратите внимание на то, что когда нет явного инициализатора, применяется правило по умолчанию, таким образом - ira = 48. Кроме того, значения перечислимых констант могут быть не уникальными.

5. Каждое перечисление является отдельным типом. Типом элемента перечисления является само перечисление. Например, в

```
    enum Keyword {ASM, AUTO, BREAK};
```

AUTO имеет тип Keyword.

6. Перечислимая константа может быть объявлена анонимно, то есть без имени типа.

```
    enum {FALSE, TRUE};
    enum {lazy, hazy, crazy} why;
```

Первое объявление — распространенный способ объявления мнемонических целочисленных констант. Второе объявление объявляет переменную перечислимого типа why, с допустимыми значениями этой переменной lazy,hazy и crazy.

7. Перечисления могут неявно преобразовываться в обычные целочисленные типы, но не наоборот.

```
enum boolean {FALSE, TRUE} q;
enum signal {off, on} a = on;  //a инициализируется в on
enum answer {no, yes, maybe = -1} b;

int i, j = true;  //верно true преобразуется в 1

a = off;  //верно

i = a;  //верно i становится 1

q = a;  //неверно два различных типа

q = (boolean)a;  //верно явное преобразование приведением
```

Указатели на функции

Прежде чем вводить указатель на функцию, напомним, что каждая функция характеризуется типом возвращаемого значения, именем и сигнатурой. Напомним, что сигнатура определяется количеством, порядком следования и типами параметров. Иногда говорят, что сигнатурой функции называется список типов ее параметров.

А теперь путём последовательности утверждений придем к обсуждению темы данного раздела урока.

1. При использовании имени функции без последующих скобок и параметров имя функции выступает в качестве указателя на эту функцию, и его значением служит адрес размещения функции в памяти.

2. Это значение адреса может быть присвоено некоторому указателю, и затем уже этот новый указатель можно применять для вызова функции.

3. В определении нового указателя должен быть тот же тип, что и возвращаемое функцией значение, и та же сигнатура.

4. Указатель на функцию определяется следующим образом:

```
тип_функции    (*имя_указателя)(спецификация_параметров);
```

Например: int (*func1Ptr) (char); - определение указателя func1Ptr на функцию с параметром типа char, возвращающую значение типа int.

Примечание: Будьте внимательны!!! Если приведенную синтаксическую конструкцию записать без первых круглых скобок, т.е. в виде int *fun (char); то компилятор воспримет ее как прототип некой функции с именем fun и параметром типа char, возвращающей значение указателя типа int *.

Второй пример: char * (*func2Ptr) (char * ,int); - определение указателя func2Ptr на функцию с параметрами типа указатель на char и типа int, возвращающую значение типа указатель на char.

Иллюстрируем на практике.

В определении указателя на функцию тип возвращаемого значения и сигнатура (типы, количество и последовательность параметров) должны совпадать с соответствующими типами и сигнатурами тех функций, адреса которых предполагается присваивать вводимому указателю при инициализации или с помощью оператора присваивания. В качестве простейшей иллюстрации сказанного приведем программу с указателем на функцию:

```
#include <iostream>
using namespace std;
void f1(void)        // Определение f1.
{
        cout << "Load f1()\n";
 }
void f2(void)        // Определение f2.
{
         cout << "Load f1()\n";
}
void main()
{
        void (*ptr)(void);  // ptr - указатель на функцию.
        ptr = f2;              // Присваивается адрес f2().
        (*ptr)();              // Вызов f2() по ее адресу.
        ptr = f1;              // Присваивается адрес f1().
        (*ptr)();              // Вызов f1() по ее адресу.
        ptr();                 // Вызов эквивалентен (*ptr)();
}
Результат выполнения программы:

        Load f2()
        Load f1()
        Load f1()
        Press any key to continue
```

Здесь значением имени_указателя служит адрес функции, а с помощью операции разыменования * обеспечивается обращение по адресу к этой функции. Однако будет ошибкой записать вызов функции без скобок в виде *ptr();. Дело в том, что операция () имеет более высокий приоритет, нежели операция обращения по адресу *. Следовательно, в соответствии с синтаксисом будет вначале сделана попытка обратиться к функции ptr(). И уже к результату будет отнесена операция разыменования, что будет воспринято как синтаксическая ошибка.

При определении указатель на функцию может быть инициализирован. В качестве инициализирующего значения должен использоваться адрес функции, тип и сигнатура которой соответствуют определяемому указателю.

При присваивании указателей на функции также необходимо соблюдать соответствие типов возвращаемых значений функций и сигнатур для указателей правой и левой частей оператора присваивания. То же справедливо и при последующем вызове функций с помощью указателей, т.е. типы и количество фактических параметров, используемых при обращении к

функции по адресу, должны соответствовать формальным параметрам вызываемой функции. Например, только некоторые из следующих операторов будут допустимы:

```
    char f1(char) {...}       // Определение функции.
    char f2(int) {...}        // Определение функции.
    void f3(float) {...}      // Определение функции.
    int* f4(char *){...}     // Определение функции.
    char (*pt1)(int);        // Указатель на функцию.
    char (*pt2)(int);         // Указатель на функцию.
    void (*ptr3)(float) = f3; // Инициализированный указатель.
    void main()
    {
        pt1 = f1;  // Ошибка - несоответствие сигнатур.
         pt2 = f3;   // Ошибка - несоответствие типов (значений и
сигнатур).
        pt1 = f4;  // Ошибка - несоответствие типов.
        pt1 = f2;  // Правильно.
        pt2 = pt1; // Правильно.
        char c = (*pt1)(44); // Правильно.
        c = (*pt2)('\t');    // Ошибка - несоответствие сигнатур.
    }
```

Следующая программа отражает гибкость механизма вызовов функций с помощью указателей.

```
    #include <iostream>
    using namespace std;
    // Функции одного типа с одинаковыми сигнатурами:
    int add(int n, int m) { return n + m; }
    int division(int n, int m) { return n/m; }
    int mult(int n, int m) { return n * m; }
    int subt(int n, int m) { return n - m; }
    void main()
    {
            int (*par)(int, int); // Указатель на функцию.
            int a =  6, b = 2;
            char c = '+';
            while   (c != ' ')
            {
                    cout << "\n Arguments: a = " << a <<", b = " <<
b;
                   cout << ". Result for c = \'" << c << "\':";

                   switch (c)
                   {
                   case '+':
                           par = add;
                           c = '/';
                           break;
                   case '-':
                           par = subt;
                           c = ' ';
```

```
                        break;
                case '*':
                        par = mult;
                        c = '-';
                        break;
                case '/':
                        par = division;
                        c = '*';
                        break;
                }
                   cout << (a = (*par)(a,b))<<"\n"; //Вызов  по
адресу.
            }
     }
     Результаты выполнения программы:


            Arguments: a = 6, b = 2. Result for c = '+':8

            Arguments: a = 8, b = 2. Result for c = '/':4

            Arguments: a = 4, b = 2. Result for c = '*':8

            Arguments: a = 8, b = 2. Result for c = '-':6
            Press any key to continue
```

Цикл продолжается, пока значением переменной с не станет пробел. В каждой итерации указатель par получает адрес одной из функций, и изменяется значение с. По результатам программы легко проследить порядок выполнения ее операторов.

Массивы указателей на функции.

Указатели на функции могут быть объединены в массивы. Например, float (*ptrArray[4]) (char) ; - описание массива с именем ptrArray из четырех указателей на функции, каждая из которых имеет параметр типа char и возвращает значение типа float. Чтобы обратиться, например, к третьей из этих функций, потребуется такой оператор:

```
     float a = (*ptrArray[2])('f');
```

Как обычно, индексация массива начинается с 0, и поэтому третий элемент массива имеет индекс 2.

Массивы указателей на функции удобно использовать при разработке всевозможных меню, точнее программ, управление которыми выполняется с помощью меню. Для этого действия, предлагаемые на выбор будущему пользователю программы, оформляются в виде функций, адреса которых помещаются в массив указателей на функции. Пользователю предлагается выбрать из меню нужный ему пункт (в простейшем случае он вводит номер выбираемого пункта) и по номеру пункта, как по индексу, из массива выбирается соответствующий адрес функции. Обращение к функции по этому

адресу обеспечивает выполнение требуемых действий. Самую общую схему реализации такого подхода иллюстрирует следующая программа для "обработки файлов":

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

/* Определение функций для обработки меню
(функции фиктивны т. е. реальных действий не выполняют):*/

void act1 (char* name)
{
        cout <<"Create file..." << name;
}
void act2 (char* name)
{
        cout << "Delete file... " << name;
}
void act3 (char* name)
{
        cout << "Read file... " << name;
}
void act4 (char* name)
{
        cout << "Mode file... " << name;
}
void act5 (char* name)
{
        cout << "Close file... " << name;
}

void main()
{
        // Создание и инициализация массива указателей
          void (*MenuAct[5])(char*)  = {act1, act2, act3, act4,
act5};

        int number;  // Номер выбранного пункта меню.
        char FileName[30];  // Строка для имени файла.

        // Реализация меню
        cout << "\n 1 - Create";
        cout << "\n 2 - Delete";
        cout << "\n 3 - Read";
        cout << "\n 4 - Mode";
        cout << "\n 5 - Close";

        while (1)  // Бесконечный цикл.
        {
                while (1)
                    { /* Цикл продолжается до ввода правильного
номера.*/
                        cout << "\n\nSelect action: ";
                        cin >> number;
                        if (number>>= 1 && number <= 5) break;
```

```
                        cout << "\nError number!";
                }
                if (number != 5)
                {
                        cout << "Enter file name: ";
                        cin >> FileName; // Читать имя файла.
                }
                else break;
                // Вызов функции по указателю на нее:
                (*MenuAct[number-1])(FileName);
        } // Конец бесконечного цикла.
}
```

Пункты меню повторяются, пока не будет введен номер 5 — закрытие.

Домашнее задание

1. Написать программу, которая осуществляет добавление строки или столбца в любое место двумерной матрицы по выбору пользователя.

2. Дана матрица порядка MxN (M строк, N столбцов). Необходимо заполнить ее значениями и написать функцию, осуществляющую циклический сдвиг строк и/или столбцов массива указанное количество раз и в указанную сторону.