Лекция 8.

СЛОЖНЫЕ СТРУКТУРЫ ДАННЫХ НА ОСНОВЕ УКАЗАТЕЛЕЙ

Массивы указателей (каталоги)

Как и все обычные переменные, указатели можно помещать в массив. Объявление массива, состоящего из 10 целочисленных указателей, выглядит следующим образом.

```
int * mas_pint[10];
```

Чтобы извлечь значение динамической переменной, используя указатель `mas_pint[2]`, необходимо его разыменовать:

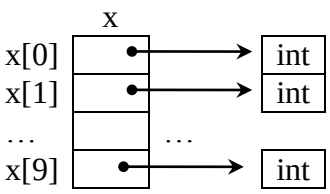
```
* mas_pint[2];
```

Массив указателей (каталог) передается в функцию как обычно — достаточно указать его имя в качестве параметра.

Пример 1.

```
const int size = 10;
void FillArray(int *p[]){
   for (int i=0; i<size; i++) {
      p[i] = new int;
      *p[i] = i;
   }
}

int main()
{
   int *x[size]{};
   FillArray(x);
   for (int i=0; i<size; i++)
      cout << *x[i] << " ";
   for (int i=0; i<size; i++)
      delete x[i];
   return 0;
}
```



Вопросы для самостоятельного изучения:

а). Что будет выведено на экран в результате работы программы из примера 1?

б). Что хранит `x[0]` ?

в). Равносильны ли выражения `*x[1]` и `(*x)[1]` ?

г). Можно ли выделить память для динамических переменных – элементов массива `x` из примера 1 оператором `x = new int [10];` ?

Далее информацию по работе с каталогами (построение, вывод, упорядочивание компонентов) читай в п. «2. Динамические структуры данных с каталогом» и «3. Многоуровневые структуры данных с каталогами» описания к лабораторной работе №5 «Массивы указателей».

Дальнейшим развитием темы массивов указателей является использование каталогов для составных типов данных, например, массивов.

Рассмотрим пример, в котором используются операции `new` и `delete` для управления сохранением строк в стиле C (массив символов с последним управляющим символом '\0'), длина которых заранее неизвестна.

Следующий код определяет функцию `getname()`, которая возвращает указатель на введённую пользователем строку. Эта функция читает ввод в большой временный массив, а затем использует `new[]` с указанием соответствующего размера, чтобы выделить фрагмент памяти в точности такого размера, который позволит вместить входную строку. После этого функция возвращает указатель на этот блок. Такой подход может сэкономить огромный объем памяти в программе, хранящей большое количество строк в стиле C.

Предположим, что ваша программа должна прочитать 100 строк, самая длинная из которых может составлять 79 символов, но большинство строк значительно короче. Если вы решите использовать массивы `char` для хранения строк, то вам понадобится 100 массивов по 80 символов

каждый, т.е. всего 8000 байт, причем большая часть этого блока памяти останется неиспользованной. В качестве альтернативы можно создать массив из 100 указателей на массив `char` и применить `new` для выделения ровно такого объема памяти, сколько необходимо для каждой строки (рис. 1).

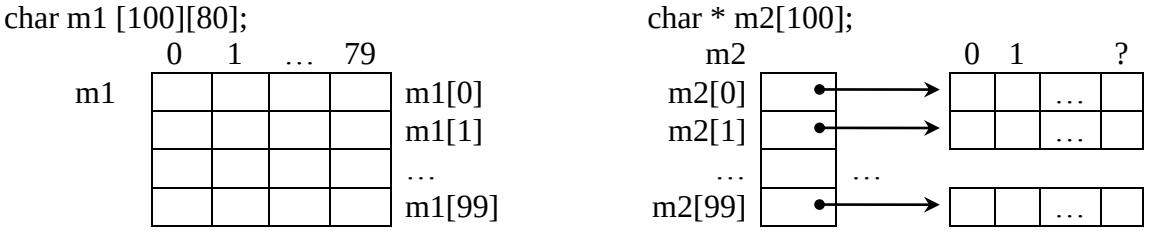


Рис. 1. Структуры данных: двухмерный массив символов (слева) и массив указателей на строки (справа)

Пример 2.

```
// delete.cpp — использование операции delete
#include <iostream>
#include <cstring>    // или string.h
using namespace std;
char * getname(void); // прототип функции
int main()
{
   char * name;     // объявление указателя, но без хранилища для символов
   name = getname(); // присваивание name адреса новой строки
   cout << name << " at " << (int *) name << "\n";
   delete [] name;   // освобождение памяти
name = getname();    // повторное использование освобожденной памяти
cout << name << " at " << (int *) name << "\n";
delete [] name;      // снова освобождение памяти
   return 0;
}
char * getname()     // возвращает указатель на новую строку
{
   char temp[80];    // временное хранилище
   cout << "Enter last name: "; // ввод фамилии
   cin >> temp;
   char * ps = new char [strlen (temp) + 1] ;
   strcpy(ps, temp); // копирование строки в меньшее пространство
   return ps;        // по завершении функции переменная temp теряется
}
```

Результаты работы программы (полужирным выделен ввод пользователя):
Enter last name: **Ivanov**
Ivanov at 0x004326b8
Enter last name: **Ivan**
Ivan at 0x004301c8

Пояснения к примеру:

①Функция `getname()` использует `cin` для размещения введённого слова в массив `temp`. Далее она обращается к `new` для выделения памяти, достаточной, чтобы вместить это слово. С учетом нулевого завершающего строку символа программе требуется сохранить в строке `strlen(temp)+1` символов, поэтому именно это значение передается `new`. После получения пространства памяти `getname()` вызывает стандартную библиотечную функцию `strcpy()`, чтобы скопировать строку `temp` в выделенный блок памяти. Функция не проверяет, поместится ли строка, но `getname()` гарантирует выделение блока памяти подходящего размера. В конце функция возвращает `ps` — адрес копии строки. При этом сама переменная `ps` после завершения работы функции уничтожается и доступ к памяти, на которую она указывает, становится возможным только через возвращаемый адрес `ps`.

②Внутри `main()` возвращенное значение (адрес) присваивается указателю `name`. Этот указатель объявлен в `main()`, но указывает на блок памяти, выделенный в функции `getname()`. Затем программа печатает строку и ее адрес.

③Далее, после освобождения блока, на который указывает `name`, функция `main()` вызывает `getname()` второй раз. C++ НЕ гарантирует, что только что освобожденная память будет выделена при следующем вызове `new`, и, как видно из вывода программы (`cout << (int*)name`), это и не происходит (выводятся разные адреса блоков выделенной памяти).

④Обратите внимание, что в рассматриваемом примере `getname()` выделяет память, а `main()` освобождает ее. Обычно это не слишком хорошая идея — размещать `new` и `delete` в разных функциях, потому что в таком случае очень легко забыть вызвать `delete`. Данный пример просто демонстрирует, что подобное возможно.

Упражнение по программированию.

В примере 2 описана функция `getname()`, которая возвращает указатель на строку произвольной длины. А как будет выглядеть функция, возвращающая массив из 100 указателей на строки произвольной длины?