Динамические структуры данных

1. Структура памяти программы
2. Динамическая память и динамические переменные.
3. Динамические массивы.
4. Функции управления памятью.

Динамическая память и динамические переменные

Статическая память выделяется до начала работы программы под глобальные переменные (до *main()*) или статические переменные (*static*) и освобождается только при завершении программы. Статическая память инициализируется случайными значениями. Поскольку область статической памяти выделяется заранее (в момент начала работы программы), она имеет фиксированный размер, что не всегда удобно и оправдано (например, массивы фиксированной длины).

Динамическая память (*куча- Heap)*, выделяется явно по запросу программы из ресурсов операционной системы и контролируется указателем. По окончании работы программы, вся затребованная программой динамическая память возвращается ОС.

Особенности динамической памяти:

- она не инициализируется автоматически и должна быть явно освобождена;
- в отличие от статической памяти динамическая память практически не ограничена (только размером оперативной памяти) и может динамически меняться в процессе работы программы;
- поскольку она контролируется указателем, доступ к ней осуществляется несколько дольше, чем для статической памяти;
- программист сам должен заботиться о выделении и освобождении памяти, что чревато большим количеством потенциальных ошибок.

Динамическое выделение и освобождение памяти

Динамической переменной называется переменная, память для которой выделяется во время работы программы с помощью оператора **New**.

Динамическая память должна быть связана с некоторым указателем, подходящего типа

```
int* p;
p = new int;
*p = 10;
cout << *p; // 10
delete p; // память освобождена
```

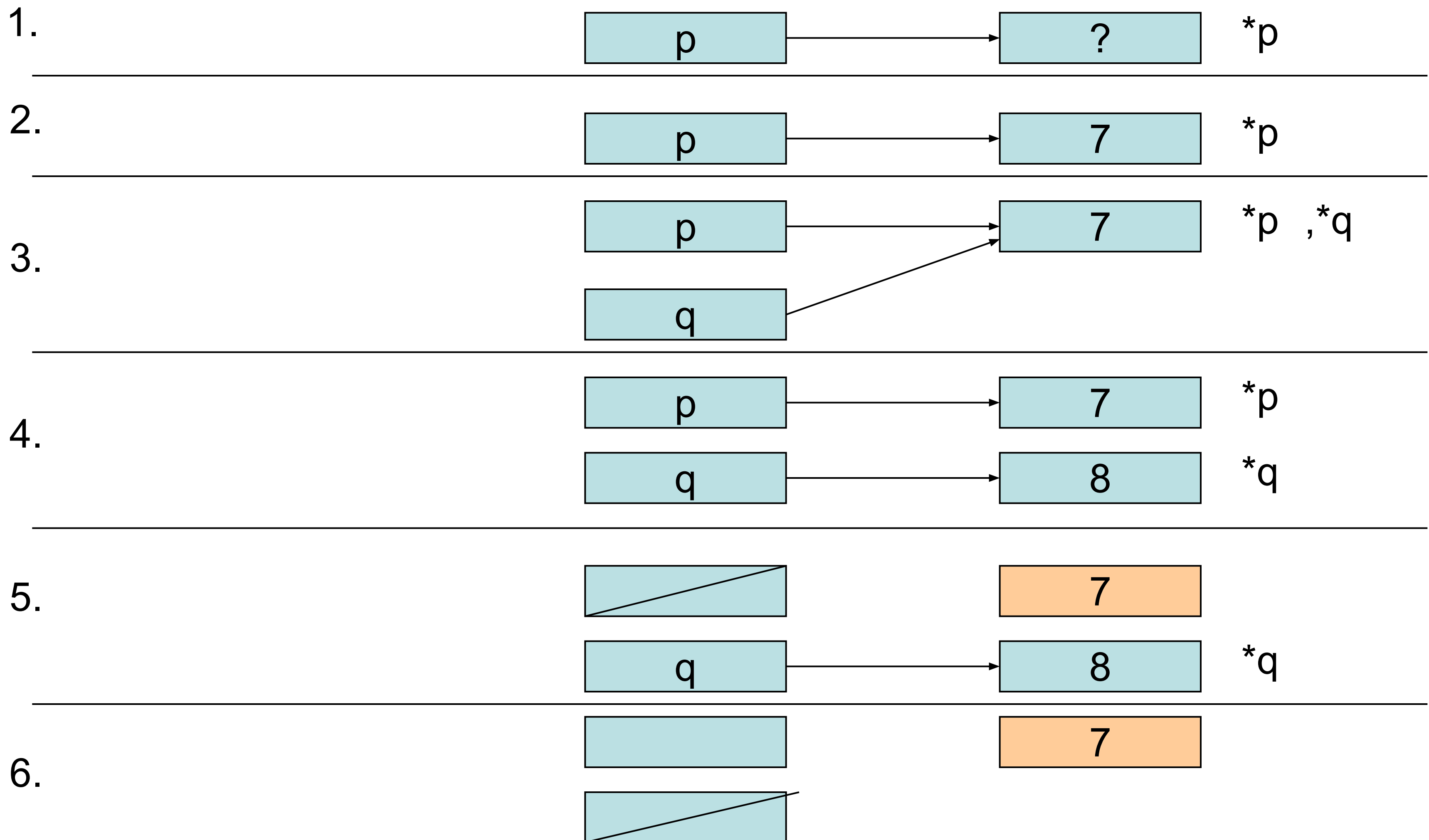
При создании одной динамической переменной можно сразу инициализировать её значение:

```
int* p;
p = new int(10);
cout << *p; // 10
delete p; // память освобождена
```

Если не освобождать динамическую память, то она будет занята до завершения программы, что неприемлемо.

Операторы New и Delete

Динамическое распределением памяти (dynamic allocation) памяти происходит с помощью оператора ***new <тип>***, который выделяет новую ячейку памяти для переменной, и возвращает указатель на нее*:*



Динамические массивы в C++

Можно выделять сразу несколько ячеек динамической памяти, получая динамический массив. Для этого его размер указывается в квадратных скобках после типа **int [n]**. Чтобы удалить динамический массив и освободить память используется оператор **delete[]**.

```
int* p;
p = new int [12];
for (int i=0; i<12; i++) {
	*(p+i) = i + 1;
	cout << *(p+i) << ' '; // 1 2 3 ... 12
}
delete [] p; // память освобождена
```

Сразу после создания динамический массив автоматически заполняется нулями.

Двумерный динамический массив

```
// объявление двумерного динамического массива из 10 элементов
float **ptrarray = new float* [2];          // две строки в массиве
    for (int count = 0; count < 2; count++)
        ptrarray[count] = new float [5];    // и пять столбцов
```

float **ptrarray - указатель второго порядка , который ссылается на массив указателей float* [2]

```
// освобождение памяти, отводимой под двумерный дин. массив
  for (int count = 0; count < 2; count++)
          delete [] ptrarray[count];
//      где 2 – количество строк в массиве
```

Объявление и удаление двумерного динамического массива выполняется с помощью цикла.

“Потеря” памяти

Если в указатель, уже хранящий адрес какого-то фрагмента динамической памяти, записать новый адрес, то фрагмент динамической памяти будет потерян

```
int* p;
p = new int(12);
int a = 777;
p = &a; // теперь до 12 никак не добраться
```

Проблема становится особенно острой, когда в памяти теряются целые массивы (они занимают больше места, чем отдельные переменные).

```
int* p;
for (int i=1; i<=10; i++) {
    p = new int[100];
}
delete[] p;
```

Всего массивов будет создано 10, но только от последнего из них память будет освобождена после выхода из цикла.

9 массивов * 100 элементов * 4 байта = 3600 байт потерянной памяти,

Важно вовремя освобождать занятую память!

Функции управления памятью

Функция управления памятью	Действие
`malloc()`	Распределение
`calloc()`	Распределение
`realloc()`	Перераспределение
`free()`	Освобождение

Применяются там, где требуется более гибкое управление памятью, чем с помощью New и Delete, которые работают с блоками памяти, соответствующими типам указателей.

malloc()

malloc() – предназначена для выделения непрерывной области памяти заданного размера:

```
void * malloc(int size); // выделить область памяти размером
                         // в size байтов и возвратить адрес
```

```
#include <stdlib.h>
int *intarray;
/* захватываем пространство для 20 целых */
intarray = (int*) malloc(20*sizeof(int));
```

Функция malloc() возвращает указатель без типа.
Если выделить память не удалось, функция возвращает значение NULL.

calloc()

calloc() – предназначена для выделения памяти под заданное количество блоков:

```
void * calloc(int n, int size); // n - количество блоков
                                // size – размер каждого блока в байтах
```

```
#include <stdlib.h>
lalloc = (long*) calloc(40, sizeof(long));
```

Возвращает указатель на первый байт выделенной области памяти.
Если выделить память не удалось, функция возвращает значение NULL.

realloc()

realloc() – предназначена для изменения размера динамически выделенной области:

```
void* realloc( void* ptr,  int size);
// ptr – указатель на первоначальную область памяти
// size – новый размер области памяти.
```

```
#include <stdlib.h>
har *realloc(ptr,size);
```

free()

free() – предназначена для освобождения памяти, выделенной функциями malloc(),calloc(),realloc():

```
void free(void *ptr); //ptr -  указатель на захваченный блок памяти
```

```
char *alloc;
/* захватывает 100 байтов и освобождает их */
if ((alloc=malloc(100))==NULL
/* проверяет на правильность указателя */
printf("unable to allocate memory\n");
else {
.
.
free(alloc);
/* освобождает память для heap */
}
```

Рекомендации

Функции **malloc/free/realloc** представляют собой библиотеку «классического» Си и транслятором не контролируются.

С их помощью можно работать с обычными переменными и их массивами, но никак не с объектами.

Операции **new/delete** используют собственную систему контроля данных в динамической памяти, поэтому при работе с объектами необходимо использовать исключительно их.

Отсюда рекомендации:

- не применять для динамической переменной или динамического массива одновременно функции и операции управления динамической памятью;
- при работе с объектами использовать только операции **new/delete;**
- при освобождении памяти из-под динамического массива «подсказывать» транслятору, что указатель ссылается именно на массив, записывая перед указателем пустые скобки: **delete [] pd1.**

Пример

```
// Строка как динамический массив - объединение строк

char  *twoToOne(char *p1, char *p2){
char  *out; int   n1,n2;
for (n1=0; p1[n1]!='\0'; n1++);                    // длина первой строки
for (n2=0; p2[n2]!='\0'; n2++);                    // длина второй строки
if ((out = new char [n1+n2+1]) == NULL)
     return NULL;                                  // выделить память под результат
for (n1=0; *p1!='\0';)  out[n1++] = *p1++;
while(*p2!=0) out[n1++] = *p2++;                   // копировать строки
      out[n1] = '\0';
      return out;
}
```

Задачи

1. Объясните разницу между четырьмя объектами:

```
(a)    int ival = 1024;
(b)    int *pi = &ival;
(c)    int *pi2 = new int(1024);
(d)    int *pi3 = new int[1024];
```

2. Проверьте код: что делает этот код и где ошибка?

```
int *pi = new int(10); int *pia = new int[10];
while ( *pi < 10 ) {
    pia[*pi] = *pi; *pi = *pi + 1;
}
delete pi;
delete[] pia;
```

3. Реализуйте программу:
Пользователь вводит строку цифр с клавиатуры (максимальная длина строки — 80 символов). Программа должна выбрать из строки все чётные цифры (нуль отнести к ним), если они есть в строке, и поместить их в первый динамический массив, и все нечётные цифры, если они есть — поместить их во второй динамический массив. Вывести оба динамических массива на экран.