Динамическая память
Указатели в C++

Оперативная память

- **Статическая память** - *выделяется при запуске программы* для размещения глобальных и статических объектов, а также объектов, определённых в пространствах имён.
- **Автоматическая память** - *резервируется при запуске программы* для размещения локальных объектов. Автоматическую память часто называют **стеком**.
- **Динамическая память** - выделяется из доступной свободной оперативной памяти непосредственно *во время выполнения программы* под размещение конкретных объектов.

В адресном пространстве каждого процесса содержится:

- образ EXE-файла программы;
- все несистемные DLL, загруженные вашей программой;
- глобальные данные программы (как доступные для чтения и записи, так и предназначенные только для чтения);
- стек программы;
- динамически выделяемая память, в том числе куча Windows и куча *библиотеки С периода выполнения (* CRT);
- блоки памяти, совместно используемые несколькими процессами;
- локальная память отдельных выполняемых потоков;
- всевозможные особые системные блоки памяти, в том числе таблицы виртуальной памяти;
- ядро, исполнительная система и DLL-компоненты Windows.

Виртуальное пространство FLAT - модели

4 Гбайт	Системная область	0xFFFFFFFF
3 Гбайт	Область совместного использования	0xC0000000
2 Гбайт	DLL-пользователя Стек Куча EXE -файл	0x80000000 0x00010000
4 Мбайт	Запрещено	

Виртуальная память может находиться в трех состояниях

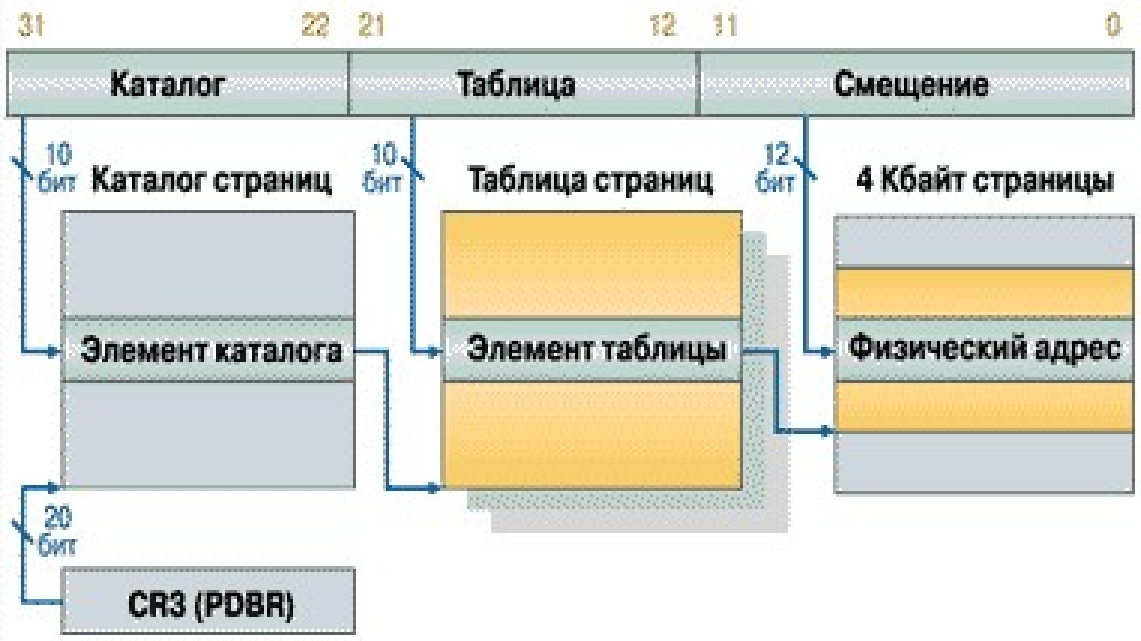
- Свободная. Ссылки на блок памяти отсутствуют, и он доступен для выделения.
- Зарезервировано. Блок памяти доступен для использования разработчиком и не может использоваться для какого-либо другого запроса на выделение. Однако сохранение данных в этот блок памяти невозможно, пока он не будет выделен.
- Выделена. Блок памяти назначен физическому хранилищу.

Виртуальное адресное пространство может стать фрагментированным.

Средства защиты памяти

- **Объектно-ориентированная защита памяти.** Каждый раз, когда процесс открывает указатель на блок адресов, монитор ссылок безопасности проверяет, разрешен ли доступ процесса к данному объекту.
- **Отдельное адресное пространство для каждого процесса.** Аппаратура запрещает процессу доступ к физическим адресам другого процесса.
- **Два режима работы**: режим ядра, в котором процессам разрешен доступ к системным данным, и пользовательский режим, в котором это запрещен.
- **Страничный механизм защиты.** Каждая виртуальная страница имеет набор признаков, который определяет разрешенные типы доступа в пользовательском режиме и в режиме ядра.
- **Принудительная очистка страниц**, освобождаемых процессами.

Страничное преобразование



Элемент таблицы страниц (**P**age **T**able **E**lement)

Страничное преобразование

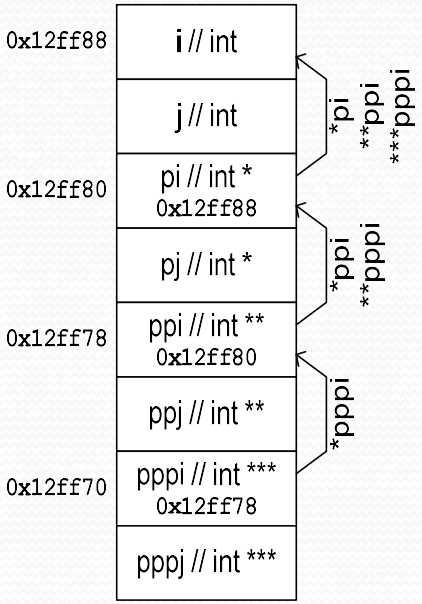
- 32-разрядный виртуальный адрес в ОС Windows разбивается на три части:
  - Старшие 10 разрядов адреса определяют номер одного из 1024 элементов в каталоге страниц, адрес которого находится в регистре процессора CR3. Этот элемент содержит физический адрес таблицы страниц.
  - Следующие 10 разрядов линейного адреса определяют номер элемента таблицы. Элемент, в свою очередь, содержит физический адрес страницы виртуальной памяти.
  - Размер страницы - 4 Кбайт, и младших 12 разрядов линейного адреса как раз хватает (2^{12} = 4096), чтобы определить точный физический номер адресуемой ячейки памяти внутри этой страницы.

Элемент таблицы страниц

- Защита – Win32 API поддерживает ряд значений, в том числе: *PAGE_NOACCESS, PAGE_READONLY, PAGE_READWRITE, PAGE_EXECUTE*.
- Базовый физический адрес страницы в памяти.
- Pagefile – индекс используемого файла подкачки (один из 16 возможных в системе файлов).
- State – состояние страницы в системе:
  - *T (Transition)* – отмечает страницу как переходную;
  - *D (Dirty)* – страница, в которую была произведена запись;
  - *P (Present)* – страница присутствует в ОП или находится в файле подкачки.

ПРИМЕР РАБОТЫ С УКАЗАТЕЛЯМИ

```
#include <iomanip.h>
#include <iostream.h>
int main()
{ using namespace std;
  int i, j
  int *pi, *pj;
int **ppi, **ppj;
 int ***pppi, ***pppj;
pi=&i; ppi=&pi;
   pppi=&ppi;
    cout<<"pi(adress of i)=ox"<<hex<<pi<<"\n;
   cout<<"ppi(adress of pi)=ox"<<hex<<ppi<<"\n";
   cout<<"pppi(adress of ppi)=ox"<<hex<<pppi<<"\n";
*pi=1;  cout << "i=" << i << "\n";;
 **ppi=2;    cout << "i=" << i << "\n";
 ***pppi=3;  cout << "i=" << i << "\n";
i=4;   cout << "*pi=" << *pi << "\n" <<
         "**ppi=" << **ppi << "\n" <<
        "***pppi=" << ***pppi << "\n";
}
```




```
*pi=1; //mov [0x12ff88], 1

**ppi=2; //*ppi=0x12ff80
         //
**ppi=[0x12ff80]=0x12ff88
         //mov [0x12ff88], 2

***pppi=3; //*pppi=0x12ff78
           //
**pppi=[0x12ff78]=0x12ff80
           //
***pppi=[0x12ff80]=0x12ff88
           //mov [0x12ff88], 3
```

Результаты:

```
pi(adress of i)=0x12ff88
ppi(adress of pi)=0x12ff80
pppi(adress of ppi)=0x12ff78
i=1
i=2
i=3
*pi=4
**ppi=4
***pppi=4
```

#include <malloc.h>

void *malloc(size_t size);

```
char * str;
int * count;
str = (char *) malloc (5); // отводится 5*sizeof(char) байт
count = (int *) malloc (10*sizeof(int)); // отводится 10*sizeof(int) байт
free(str);                    // освобождение памяти
free(count);
```

void *calloc(size_t nitem, size_t size);

```
char * str;
int * count;
str = (char *) calloc (10, sizeof(char));
count = (int *) calloc (5, sizeof(int));

char far *fptr;
fptr = (char far *) farmalloc(10);
Farfree(fptr);
```

void *realloc(void *block, size_t size);

```
 count = (int *) realloc (count, 10*sizeof(int));
```

Операторы new и delete имеют две формы:

- управление динамическим размещением в памяти единичного объекта:

```
int * p= new int;
delete p;
```

- динамическое размещение массива объектов:

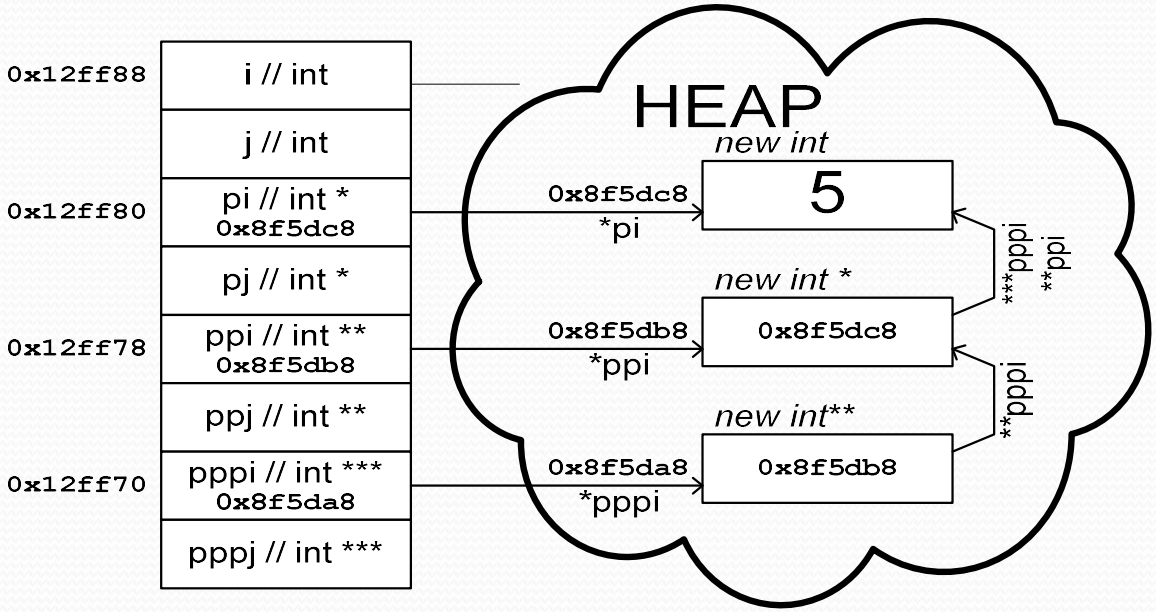
```
int * p= new int[100];
delete[] p;
```

Еще один пример с указателями

```
pppi = new int **;
  ppi  = new int *;
  pi   = new int ;
*pppi=ppi;
 *ppi=pi;  //   ppi=&pi; утечка памяти
  ***pppi=5;
  cout << "*pi=" << *pi << "\n";
delete pppi; delete ppi; delete pi;
```

Результат:

***pi=5**



Демонстрация работы с динамической памятью через функции new и delete на примере выделения памяти под динамический двумерный массив целых чисел.

```
#include <malloc.h>
#include <iostream>
void out(int ** mas);  // вывод массива на
    экран
void clear_mem(int ** mas);  // очистка
int n=2; // число строк
int m=3; // число столбцов
int  main()
{
 int ** d;
 int i,j;
 d = new int* [n];   // выделяем память под
    //указатели на строки
 for ( i=0; i<n; i++)  // выделяем память под
    //указатели на столбцы
     d[i]=new int [m];
// инициализация массива
 for ( i=0; i<n; i++)
    for ( j=0; j<m; j++)
    {d[i][j]=i+j;}
    out(d);
    clear_mem(d);
return 0;
}
```

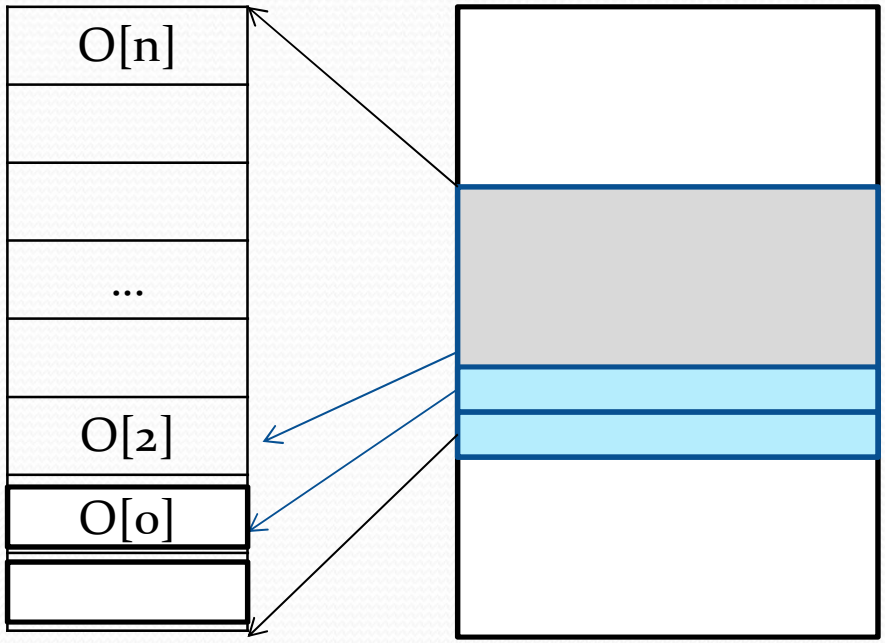
```
void out (int **d)
{
            for (int i = 0; i < n; i++)
             {  for (int j = 0; j < m; j++)
                 std:: cout << d[i][j] << " ";
                  std::cout << "\n";}
}

void clear_mem(int **d)
{
   for (int i = 0; i < n; i++)
    delete[] d[i]; // освобождаются
                          //столбцы
    delete[] d;   // освобождаются
            //указатели на строки
}
```

Алгоритмы для управления областями памяти

1. для объектов одного типа: менеджер объектов или stab-аллокатор.
2. для объектов одного размера: битовые маски или алгоритм "близнецов".
3. для объектов произвольного размера и типа метод граничных маркеров.

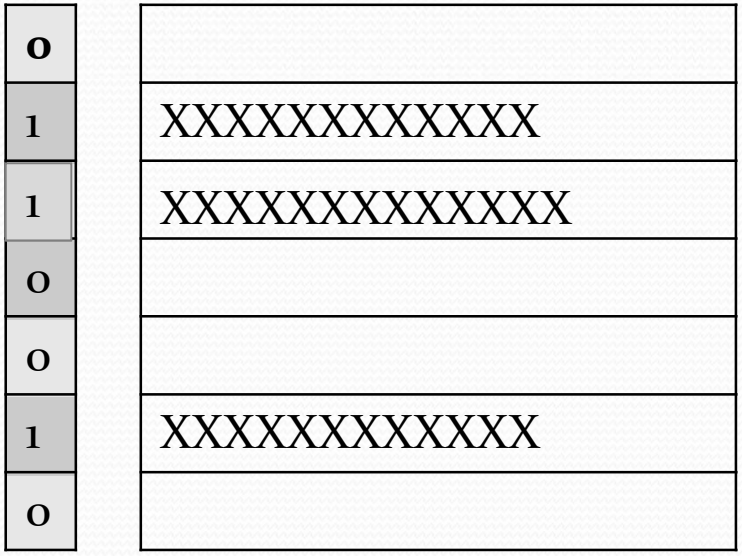
Менеджер объектов

Список свободных объектов



Область памяти

Алгоритм битовой маски



Метод граничных маркеров

Структура элемента памяти

Флаг занятости	Размер блока	Следующий	Предыдущий	Элемент памяти	Конец блока

Утечки памяти

```
#define _CRTDBG_MAP_ALLOC
#include <stdlib.h>
#include <crtdbg.h>   /* _DEBUG

_CrtDumpMemoryLeaks();

_CrtMemState s1;
_CrtMemCheckpoint( &s1 );
_CrtMemDumpStatistics( &s1 );
```

Процесс сборки мусора

- Этап маркировки, выполняющий поиск всех используемых объектов и составляющий их перечень.
- Этап перемещения, обновляющий ссылки на сжимаемые объекты.
- Этап сжатия, освобождающий пространство, занятое неиспользуемыми объектами и сжимающий выжившие объекты (объекты, которые не уничтожаются при сборке мусора).

Сборка мусора. Поколения объектов

- **Поколение 0**. Это самое молодое поколение содержит короткоживущие объекты. Примером короткоживущего объекта является временная переменная. Сборка мусора чаще всего выполняется в этом поколении.
- **Поколение 1**. Это поколение содержит коротко живущие объекты и служит буфером между короткоживущими и долгоживущими объектами.
- **Поколение 2**. Это поколение содержит долгоживущие объекты. Примером долгоживущих объектов служит объект в серверном приложении, содержащий статические данные, которые существуют в течение длительности процесса.