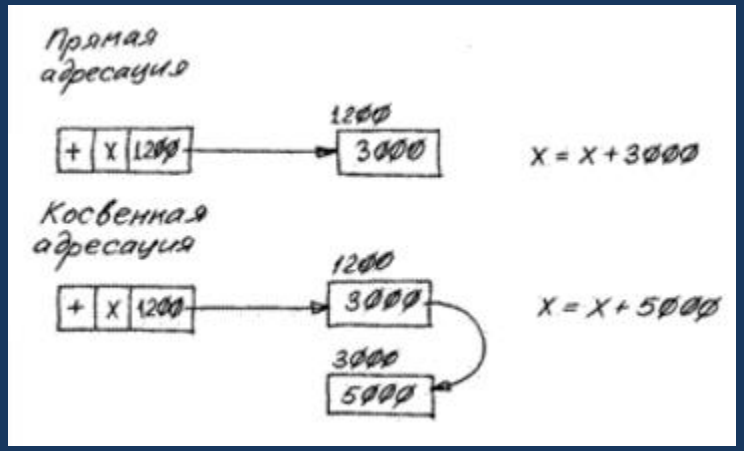
Указатели

Указатель – переменная, содержащая адрес другой (указуемой) переменной
Имеет отношение к среде исполнения программы на уровне архитектуры и к системе команд:

- непрерывное адресное пространство – виртуальное (процесс, задача) или реальное (ядро ОС), передавать указатель можно из любой части программы в другую (пример, монолитное ядро ОС)
- доступ к указуемой переменной: способ адресации в системе команд - *косвенная адресация*



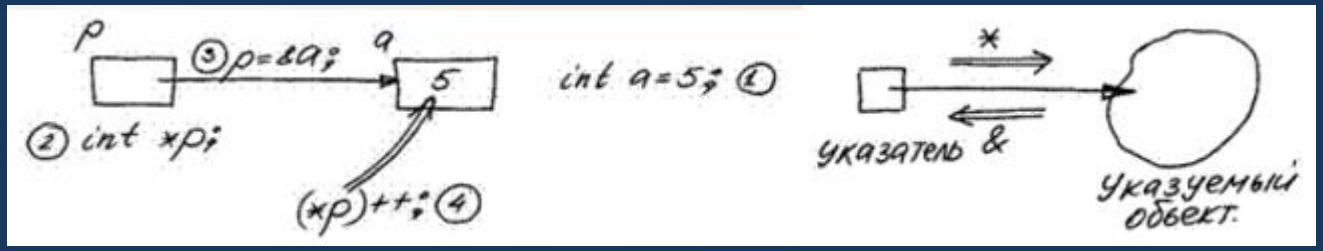
Варианты использования указателей:

- указатель на отдельную переменную (Си,Паскаль), **ссылка** (передача параметров в функцию) (cprog 5.2)
- **Адресная арифметика –** адресация к памяти, массив переменных указуемого типа (cprog 5.2)
- **Адресная арифметика + преобразование типов указателей –** работа с памятью на низком (физическом уровне), доступ к физическому представлению данных, управление памятью (cprog 9.2)

Указатели (синтаксис)

- Указатель типизирован – тип указуемой переменной задан в определении
- Последовательность действий:
  - Определение указателя
  - Назначение на указуемую переменную (объект)
  - Косвенная адресация **(разыменование)**

- Ошибка: обращение через неназначенный (неинициализированный) указатель по случайному адресу (Java – синтаксический контроль)




```
int     a,x;    // Обычные целые переменнные
int     *p;     // Переменная - указатель на другую  целую переменную
```

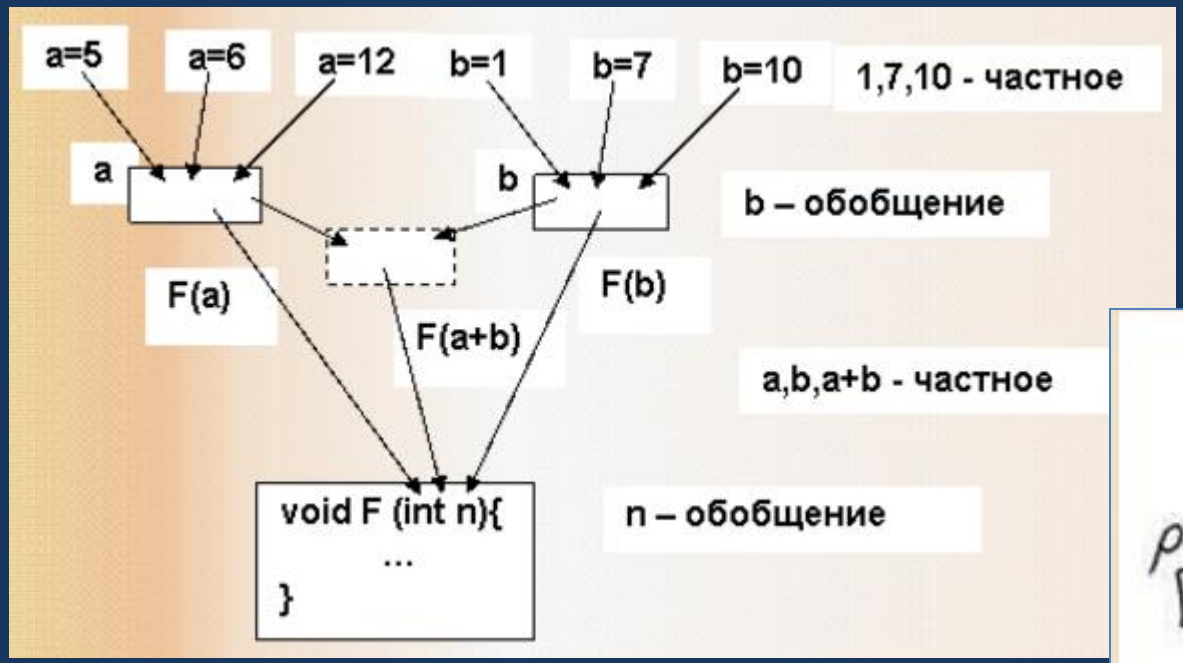
```
p = &a; // Указатель содержит адрес переменной a
```

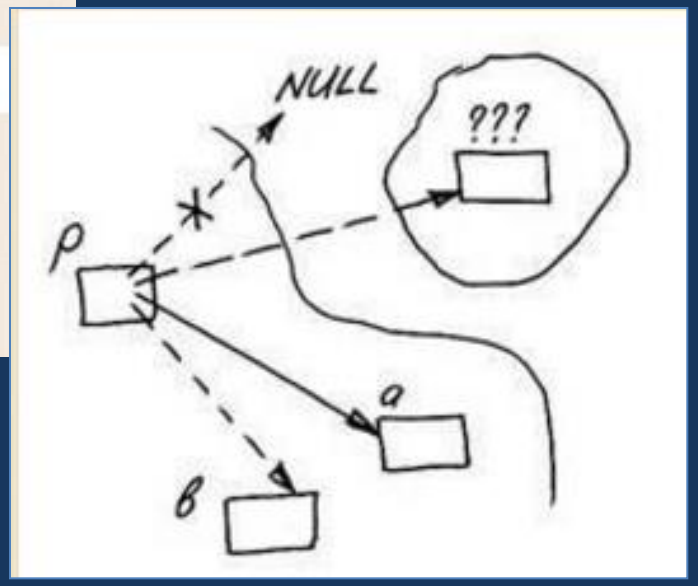
```
*p=100;         // Эквивалентно a=100
x = x + *p;     // Эквивалентно x=x+a
(*p)++;         // Эквивалентно a++
```

Указатель - философия

Указатель – **степень свободы** программы, элемент абстракции:

- Обычная (именованная) переменная
- Динамическая (выделенная память)
- Из адресного пространства другого приложения (пример: ядро ОС «видит» данные приложения)





Аналогично: константа – переменная – формальный параметр

Указатель и ссылка в Си++

Избыточность: указатель и ссылка различны синтаксически (метафорически), но дают одинаковый код

Ссылка – неявный указатель, имеющий синтаксис указуемого объекта (синоним), метафора использования - **отображение**

```
int a=5;          // Переменная – прототип
int &b=a;         // Переменная b – ссылка на переменную a
b++;              // Операция над b есть операция над прототипом a
```

```
// Формальный параметр - ссылка
void inc (int &vv) { vv++; }          // Передается указатель - синоним nn
void main(){ int nn=5; inc(nn); }     // nn=6
```

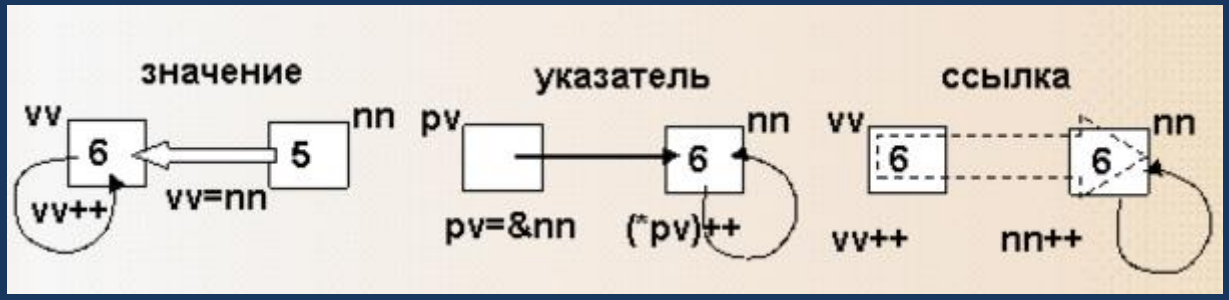
```
//-------------------------------------------------------52-02.cpp
//------ Функция возвращает ссылку на минимальный элемент массива
 int &ref_min(int A[], int n){
 for (int i=0,k=0; i<n; i++)
          if (A[i]<A[k]) k=i;
 return A[k];}
 void main(){
 int B[5]={4,8,2,6,4};
 ref_min(B,5)++;
 for (int i=0; i<5; i++) printf("%d ",B[i]); }
```

Результат-ссылка в левой части присваивания

```
//---------------------------------------------------------- 36
int &F39(int &n1, int &n2){
return n1 > n2 ? n1 : n2; }
void main10(){ int x=5,y=6,z; F39(x,y)=0; F39(x,y)++; }
```

Указатель и ссылка в Си++

Избыточность: указатель и ссылка различны синтаксически (метафорически), но дают одинаковый код



Передача параметров в функцию **по значению и по ссылке**.
Свойства любой переменной: значение = содержимое памяти, ссылка = адрес (указатель). Дуализм имени переменной в присваивании: левая часть – ссылка, правая часть – значение.

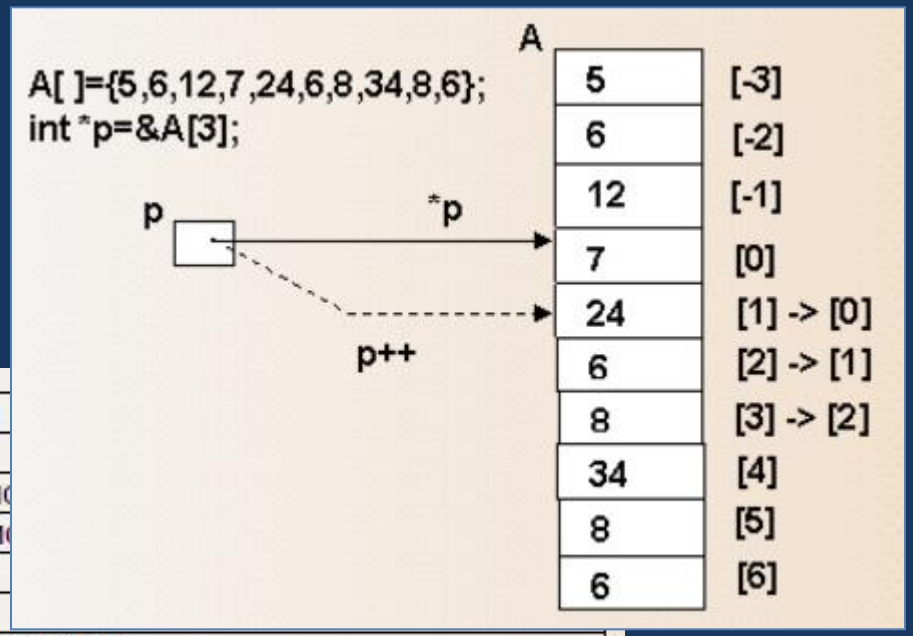
```
//-------------------------------------------------------------
// Формальный параметр - значение
void inc(int vv){ vv++; }              // Передается значение - копия nn
void main(){ int nn=5; inc(nn); }      // nn=5
//-------------------------------------------------------------
// Формальный параметр - указатель
void inc(int *pv) { (*pv)++; }                  // Передается указатель - адрес nn
void main(){ int nn=5; inc(&nn); }  // nn=6
//-------------------------------------------------------------
// Формальный параметр - ссылка
void inc (int &vv) { vv++; }           // Передается указатель - синоним nn
void main(){ int nn=5; inc(nn); }      // nn=6
```

L-value - семантическое свойство выражения: L-value = true, если в него можно присвоить, т.е. транслятор ассоциирует его с допустимым адресом объекта в памяти
Примеры: **a+b** = false, **A[i]** = true, **F(5,6)** = ???

Указатели и массивы

Любой указатель в Си ссылается на неограниченную в обе стороны область памяти (массив), заполненную переменными указуемого типа с индексацией элементов относительно текущего положения указателя:

- границы памяти программно не ограничены
- «смысл» указателя (массив или отдельный объект) – из контекста использования



	Смысл
*p	Значение указуемой переменной
p+i	Указатель на i-ю переменную после указуемо
p-i	Указатель на i-ю переменную перед указуемо
*(p+i)	Значение i-й переменной после указуемой
p[i]	Значение i-й переменной после указуемой
p++	Переместить указатель на следующую переменную
p--	Переместить указатель на предыдущую переменную
p+=i	Переместить указатель на i переменных вперед
p-=i	Переместить указатель на i переменных назад
*p++	Получить значение указуемой переменной и переместить указатель к следующей
*(--p)	Переместить указатель к переменной, предшествующей указуемой, и получить ее значение
p+1	Указатель на свободную память вслед за указуемой переменной

Указатели и массивы

МАССИВ = ПАМЯТЬ+УКАЗАТЕЛЬ
УКАЗАТЕЛЬ = МАССИВ-ПАМЯТЬ

Имя массива – адресная константа, адрес 0-го элемента массива (A = &A[0])

Массив	Указатель	Различия и сходства
int A[20]	int *p	
A	p	Оба интерпретируются как указатели и оба имеют тип int*
---	p=&A[3]	Указатель требует настройки «на память»
A[i] &A[i] A+i *(A+i)	p[i] &p[i] p+i *(p+i)	Работа с областью памяти как с обычным массивом, так и через указатель полностью идентична вплоть до синтаксиса
----	p++ *p++ p+=i	Указатель может перемещаться по памяти относительно своего текущего положения

С чем работает указатель:

- p++, p[i] – массив
- Только *p – переменная

Ошибка: N указателей на массив это не N массивов (), указатель не связан с ресурсом памяти, на которую он ссылается

Операции над указателями

- p++, p[i], p+i
- *p
- Присваивание указателей: копирование адресов (ссылаются на одно и то же)
- Сравнение на равенство: ссылаются на одно и то же
- Сравнение на <, > - сравнение адресов (ближе к началу при работе в общем массиве)
- Вычитание – «расстояние» в количестве адресуемых переменных при работе в общем массиве
- Значение указателя NULL (присваивание, сравнение)
- Указатель void* - «чистый адрес», копировать можно, *p – недопустимо, приведение к void* неявное, обратно – явное (указатель любого типа является чистым адресом)
- Указатель char* - адрес физической памяти с байтной структурой
- Приведение (преобразование к БТД) – **указатель = адрес = маш.слово = int**
  - printf(“%x”,(int)p);
  - *(int*)0x1000 = 0;
- Преобразование указуемого типа («типа указателя») char* -> int* (cprog 9.2)

Указатель char*

Строковая константа - указатель на статический массив, инициализированный символами строки

```
char *q = "ABCD";;                          // Программа
char *q;                                    // Эквивалент
char A[5] = {'A','B','C','D','\0'};
q = A;
```

```
char c1 = "ABCD"[3];
char c2 = ("12345" + 2)[1];
for (char *q = "12345"; *q !='\0'; q++);
char c3=*(--q);
```

```
//---- Поиск в строке заданного фрагмента
 char *find (char *p,char *q){                  // Попарное сравнение
 for (; *p!='\0'; p++){                         // до обнаружения расхождения
          for ( int i=0 ; q[i]!='\0' && q[i]==p[i]; i++);
          if ( q[i] == '\0') return p;          // Конец подстроки - успех
          }                                     // иначе продолжить поиск
 return NULL;}
```

```
//----- Поиск всех вхождений фрагмента в строке
 void main()
 { char c[80]="find first abc and next abc and last abc",*q="abc", *s;
 for (s=find(c,q); s!=NULL; s=find(s+strlen(q),q)) puts(s);
 }
```

Примеры

Сочетание p++ и p[i] – относительная адресация фрагмента строки от текущей позиции указателя (c[k] – k-ый относительно текущего положения c, c[-k]==c[k] - симметрия)

```
//----------------------------------------------------------19
int F19(char *p){
char *c; int ns;
for (c=p,ns=0; *c !='\0'; c++) {
        for (int k=0; c-k >=p && c[k] !='\0'; k++)
                if (c[-k] != c[k]) break;
        if (k >=3) ns++;
        }
return ns; }
```

```
//----------------------------------------------------------13
char *F13(char *c, int &m){
char *b=NULL;
for (int k=0;*c!=0;c++)
        if (*c!=' '){
                for (k=1;c[k]!=' ' && c[k]!=0;k++);
                if (k>m) m=k,b=c;
                c+=k-1;
        }
return b;
}
```

Указатели и динамическая память

Локальные переменные – стек, размер сегмента ограничен и задается при старте программы

Глобальные переменные – сегмент данных, размер определяется при трансляции по размерам переменных, занимает место в программном файле (exe)

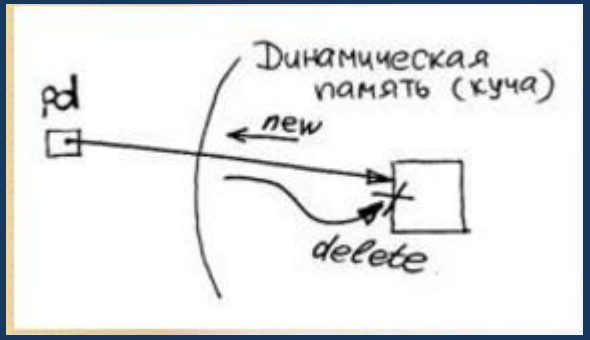
Куча (heap) – расширяемый сегмент данных, в котором библиотека окружения строит структуру данных с элементами переменной размерности (cprog 9.2)

Сегмент	Регистры	Что содержит	Когда создается
Сегмент команд	CS- сегментный, IP- адрес команды	Программный код (операции, операторы)	Трансляция
Сегмент данных	DS – сегментный	Глобальные (статические) данные	Трансляция
Сегмент стека	SS – сегментный, SP- указатель стека	Локальные данные функций, «история» работы программы	При загрузке
Динамическая память	DS – сегментный	Динамические переменные, создаваемые при работе программы	При загрузке, выполнении
Динамически связываемые библиотеки (DLL)	CS- сегментный, IP- адрес команды	Программный код разделяемых библиотек	При загрузке

Динамические переменные

- создаются и уничтожаются работающей программой путем выполнения специальных операторов (new) или вызовов функций (malloc)
- количество и размерность динамических переменных (массивов) может меняться в процессе работы программы. Это определяется числом вызовов соответствующих функций их параметрами
- динамическая переменная **не имеет имени**, доступ к ней возможен только через указатель (операция * - **разыменование** указателя)
- функция создания динамической переменной ищет (или создает) в «куче» свободную память необходимого размера и возвращает указатель на нее (адрес)
- функция уничтожения динамической переменной получает указатель на уничтожаемую переменную и помечает память как свободную («склеивание» соседних областей, **фрагментация**)

Цель: эффективное использование памяти программой «по потребности», Динамические структуры данных



Динамические переменные и массивы

```
double   *pd;
pd = new double;                    // Обычная динамическая переменная
if (pd !=NULL){
        *pd = 5;
        delete pd;}
```

Функции низкого уровня – работа с библиотекой ДРП. Операторы new/delete вызывают конструкторы/деструкторы для объектов (+много чего – скрытый код)

```
void      *malloc(int size);          // выделить область памяти размером
                                      // в size байтов и возвратить адрес
void      free(void *p);              // освободить область памяти,
                                      // выделенную по адресу p
void      *realloc(void *p, int size);
                                      // расширить выделенную область памяти
                                      // до размера size, при изменении адреса
                                      // переписать старое содержимое блока
#include <alloc.h>                    // библиотека функций управления памятью
double *pd;                 // Обычная динамическая переменная
pd = (double*)malloc(sizeof(double));
if (pd !=NULL){
        *pd = 5;
        free(pd); }
```

Динамические переменные и массивы

Динамические массивы: ДП + адресная арифметика. Синтаксически неотличимы от обычных (статических)

```
#include <malloc.h>                  // Библиотека функций управления памятью
double *pd1,*pd2;                    // Массивы динамических переменных
pd1=new double[n1];                  // Выделение памяти под ДМ
pd2 = (double*)malloc(n2*sizeof(double));
if (pd1 ==NULL || pd2==NULL) return;
for (i=0; i<n1; i++) pd1[i]=0;       // Работа с ДМ
for (i=0; i<n2; i++) pd2[i]=i;
delete [ ]pd1;                       // Освобождение памяти
free(pd2);
```

Ошибки при работе с ДП:

- библиотека ДРП быстродействующая, слабо защищенная
- не освобождает память – **утечки памяти,** увеличение объема памяти под приложением, виртуальная память в ОС – пробуксовка из-за замещения страниц
- Освобождает «не то, что брал», освобождает два раза – наведенный сбой при последующих new (malloc) – нарушение целостности структуры данных ДРП, например: **int a=0; delete &a;**

Размерность динамических массивов

«Кофе в постель могу себе позволить, но сначала нужно встать, одеться, сварить, а потом раздеться, лечь и выпить». М.Жванецкий.

- может быть вычислена заранее
- может быть получена предварительной прокруткой того же алгоритма
- резервирование большей памяти при переполнении текущей и копирование содержимого (realloc)

```
//-----------------------------------------------------56-01.cpp
//------ Динамический массив простых множителей числа
//       Размерность массива определяется двойным вычислением
int *mnog(long vv){
long nn=vv;
for (int sz=0; vv!=1; sz++){          // Цикл определения количества
     for (int i=2; vv%i!=0; i++);     // Определить очередной множитель
          vv = vv / i; }
int *p=new int[sz+1];                 // Создать динамический массив
for (int k=0; nn!=1; k++){            // Повторный цикл заполнения
     for (int i=2; nn%i!=0; i++);     // Определить очередной множитель
          p[k]=i;                     // Сохранить множитель в массиве
          nn = nn / i; }
p[k]=0; return p;}                    // Вернуть указатель на ДМ
```

Размерность динамических массивов

резервирование большей памяти при переполнении текущей и копирование содержимого (realloc)

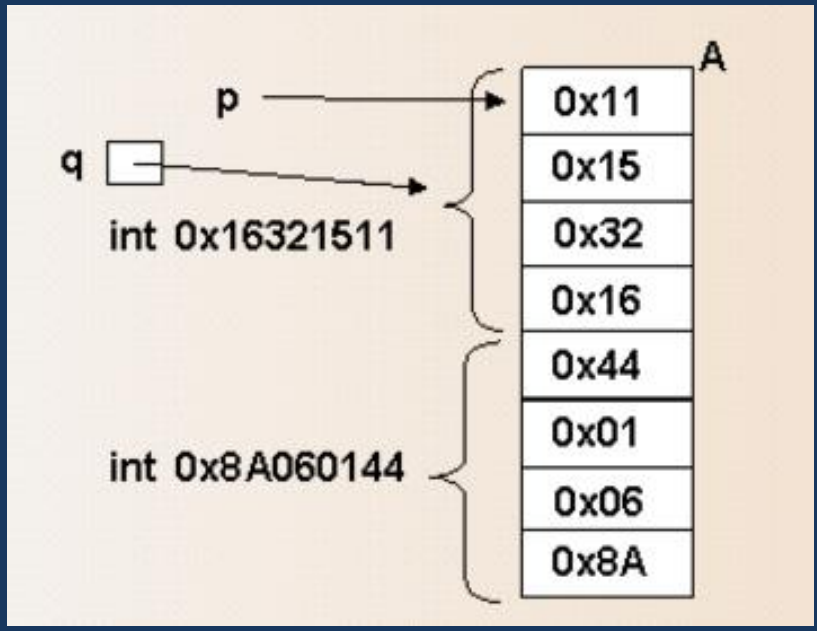
```
//-----------------------------------------------------56-01.cpp
// Загрузка динамического массива целых из файла
int *loadInt(char nm[],int &n){
FILE *fd=fopen(nm,"r");
int sz=10,*p=new int[sz]; n=0;
//---------------- или p=(int*)malloc(sz*sizeof(int));
while(fscanf(fd,"%d",&p[n])==1){      // пока есть числа в файле
    n++;
    if (n==sz){                        // массив заполнен
        sz*=2;                         // удвоить размерность
        int *q=new int[sz];            // создать новый
        for (int i=0;i<n;i++)          // копировать старый в новый
                q[i]=p[i];
        delete p;                      // уничтожить старый
        p=q;                           // считать новый за старый
//--------------- или p=(int*)realloc(p,sz*sizeof(int));
        }} return p;}
```

- Линейное увеличение размерности N+=m – «вычерпывание бочки чайной ложкой»
- Экспоненциальное (геометрическая прогрессия) N*=2, эффективность использования памяти 37.5% (1+2+4+….+n/2 = n-1) -> 75% / 2
- В целом неэффективная СД, алитенатива – двухуровневый массив (МУ на массивы), списки

Работа с памятью на низком уровне

- Интерпретация содержимого физической памяти в различных форматах (типах данных) – преобразование типа указателя, **преобразование указуемого типа данных**, операция явного преобразования типа, например (int*)
- Адресная арифметика – перемещение указателя по памяти

```
char A[20]={0x11,0x15,0x32,0x16,0x44,0x1,0x6,0x8A};
char      *p; int *q; long *l;
p = A; q = (int*)p;  l = (long*)p;
p[2] = 5;                    // записать 5 во второй байт области A
q[1] = 7;                    // записать 7 в первое слово области A
```



Запись последовательности разнотипных данных

- ***((int*)p)++** со сменой типа «на лету» (работало только в старом Borland C)
- Присваивание указателю другого типа или ***(int*)p, p+=sizeof(int)**
- Использование UNION

```
//------------------------------------------------------ 6
union x {int *pi; long *pl; double *pd;};
double F6(int *p)
{ union x ptr;
double dd=0;
for (ptr.pi=p; *ptr.pi !=0; )
        switch (*ptr.pi++) {
case 1: dd += *ptr.pi++; break;
case 2: dd += *ptr.pl++; break;
case 3: dd += *ptr.pd++; break;
    }
return dd;}
//
```

```
//------------------------------------------------------ 10
double F10(char *p)
{ double s; char *q;
for (q=p; *q!=0; q++);
for (q++; *p!=0; p++)
        switch(*p) {
case 'd': s+=*((int*)q); q+=sizeof(int); break;
case 'f': s+=*((double*)q); q+=sizeof(double); break;
case 'l': s+=*((long*)q); q+=sizeof(long); break;
    }
return s; }
//------------------------------------------------------11
```

Запись последовательности разнотипных данных

- ***((int*)p)++** со сменой типа «на лету» (работало только в старом Borland C)
- Присваивание указателю другого типа или ***(int*)p, p+=sizeof(int)**
- Использование UNION

```
//------------------------------------------------------ 6
union x {int *pi; long *pl; double *pd;};
double F6(int *p)
{ union x ptr;
double dd=0;
for (ptr.pi=p; *ptr.pi !=0; )
        switch (*ptr.pi++) {
case 1: dd += *ptr.pi++; break;
case 2: dd += *ptr.pl++; break;
case 3: dd += *ptr.pd++; break;
    }
return dd;}
//
```

```
//------------------------------------------------------ 10
double F10(char *p)
{ double s; char *q;
for (q=p; *q!=0; q++);
for (q++; *p!=0; p++)
        switch(*p) {
case 'd': s+=*((int*)q); q+=sizeof(int); break;
case 'f': s+=*((double*)q); q+=sizeof(double); break;
case 'l': s+=*((long*)q); q+=sizeof(long); break;
    }
return s; }
//------------------------------------------------------11
```

Работа с памятью на низком уровне

Работа с памятью по физическим адресам, доступ к «регистрам внешних устройств» = область памяти в адресном пространстве

```
*(int*)0x1000=5;              // Запись целого 5 по шестнадцатеричному адресу 1000
#define REG *(char*)0xFFFFF0D0
REG=0xFF;                     // Запись константы «все 1» по адресу 0xFFFFF0D0
                              // регистра данных в пространстве адресов основной памяти
```

Пример: упаковка double[]

Если элемент массива не содержит дробной части и <255 – записать байтом. 255 – индикатор вещественного числа в последовательности байтов

```
int size(double in[], int sz){
    int size=sizeof(int);
    for(int i=0;i<sz;i++)
        if (in[i]<0 || in[i]<255 && (in[i]-(int)in[i]==0))
            size++;
        else
            size+=sizeof(double)+1;
    return size;
    }
```

```
unsigned char *pack(double in[],int sz){
    unsigned char *p0 = new unsigned char[size(in,sz)];
    *(int*)p0 = sz;
    unsigned char *p1 = p0 + sizeof(int);
    for(int i=0;i<sz;i++)
        if (in[i]<0 || in[i]<255 && (in[i]-(int)in[i]==0))
            *p1++ = (unsigned char )in[i];
        else{
            *p1++ = 255;
            *(double*)p1 = in[i];
            p1 += sizeof(double);
            }
    return p0;
    }
```

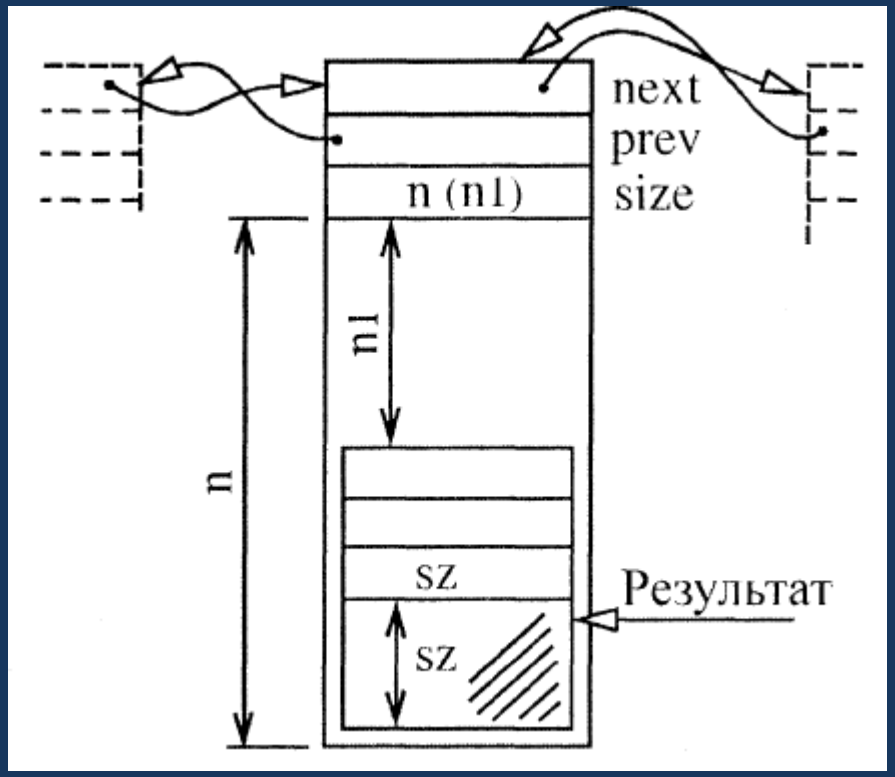
Пример: распаковка в double[]

```
double *unpack(unsigned char *p, int &sz){
    sz = *(int*)p;
    p += sizeof(int);
    double *out = new double[sz];
    for(int i=0;i<sz;i++)
        if (*p!=255)
            out[i]=*p++;
        else{
            p++;
            out[i] = *(double*)p;
            p += sizeof(double);
            }
    return out;
    }
#include <stdio.h>
int main(){
double a[]={1,3.3,3,5,7,9,1256,16,23,35};
int n=sizeof(a)/sizeof(double);
int k=size(a,n);
printf("size=%d\n",k);
double *pp = unpack(pack(a,n),k);
printf("size=%d\n",k);
for(int i=0;i<k;i++)
    printf("%5.3lf ",pp[i]);
return 0;
}
```

10*8 = 80 байтов double[]
8 + 2*(8+1) +4 = 30 байтов упакованный

```
size=30
size=10
1.000 3.300 3.000 5.000 7.000 9.000 1256.000 16.000 23.000 35.000
```

Пример: динамическое распределение памяти



Двусвязный список с элементами переменной размерности. Новый элемент «отрезается» от самого большого.
Романов Е.Л. Практикум по программированию на Си++. 3.1. с. 188.

Пример: динамическое распределение памяти

Последовательность, заданная форматом: очередной int задает длину следующей за ним области байтов, знак определяет факт свободен/занят


```
//-------------------------------------------92-02.cpp
// Система динамического распределения памяти
char *pa;                          // Область распределяемой памяти – «кучи»
int sz0;                           // Исходная размерность кучи
void create(int sz){               // Начальное состояние – один свободный блок
        pa=new char[sz];
        *(int*)pa=sz-sizeof(int);  // Размерность свободного куска – записать в начало
        sz0=sz; }
//-------------------------------------------------------------------
void _show(){                      // Просмотр состояния кучи
char *p=pa;
int lnt;
while(p<pa+sz0){                   // Пока не достигли адреса конца области
        lnt=*(int*)p;              // Извлечь из-под указателя длину блока
        if (lnt<0){                // Занятый блок - пропустить
                lnt=-lnt;          // Инвертировать длину
                printf("busy:");
                }
        else printf("free:");      // Вывести адрес (шестнадцатеричный) и длину
        printf(" addr=%8x sz=%d\n",p,lnt);
        p+=lnt+sizeof(int);        // Сдвинуть указатель на длину блока + длина счетчика
                }}
```

Пример: динамическое распределение памяти

char * - указатель на память с байтной структурой, p+=sizeof(int) – пропуск целого, *(int*)p – извлечение целого из под указателя

```
//-----------------------------------------------92-02.cpp
// Поиск строго подходящего или отрезание от первого
void *_malloc(int sz){
char *p=pa;
int Int;
while(p<pa+sz0){                        // Пока не достигли адреса конца области
        Int=*(int*)p;                   // Извлечь из-под указателя длину блока
        if (Int<0)                      // Занятый блок - пропустить
                p+=-Int+sizeof(int);
        else    {
                if (sz==Int) {          // Свободный – строго подходящий
                *(int*)p=-Int;          // Обозначить как занятый
                return p+sizeof(int);   // Возвратить указатель на область данных
                }
                p+=Int+sizeof(int);     // К следующему блоку
                }
        }
Int=*(int*)pa;                          // Отрезать от первого – взять длину
if (sz+sizeof(int)>Int) return NULL;      // Остаток мал – нет памяти
Int -=sz+sizeof(int);                   // Уменьшить размер первого блока
*(int*)pa=Int;                          // и записать полученный остаток
p=pa+Int+sizeof(int);                   // Указатель на новый блок
*(int*)p=-sz;                           // Записать в него размерность (занят – <0)
return p+sizeof(int);                    // Возвратить указатель на память
}                                       // «вслед за» счетчиком
```

Пример: динамическое распределение памяти

```
//-----------------------------------------------92-02.cpp
// Функция освобождения памяти
void _free(void *q0){
char *q=(char*)q0-sizeof(int);          // Сместить указатель на счетчик длины
int lnt=*(int*)q;                       // Извлечь счетчик длины
lnt=-lnt;
char *pr=NULL,*p=pa;                    // pr – указатель на предыдущий блок
while(p!=q){                            // Пока не достигли освобождаемого блока
        int ln=*(int*)p;
        if (ln<0) ln=-ln;
        pr=p;                           // Текущий становится предыдущим
        p+=ln+sizeof(int);              // Переход к следующему
        }                               // Просмотр до предыдущего
*(int*)q=lnt;                           // Освободить блок
if (*(int*)pr>0) {
        *(int*)pr+=lnt+sizeof(int); // Склеить с предыдущим
        p=pr;                           // Сделать предыдущий текущим
        }
lnt=*(int*)p;
q=p+lnt+sizeof(int);
if (q<pa+sz0 && *(int*)q>0)             // Есть следующий и он свободен
*(int*)p+=*(int*)q+sizeof(int);         // Склеить со следующим
}
```