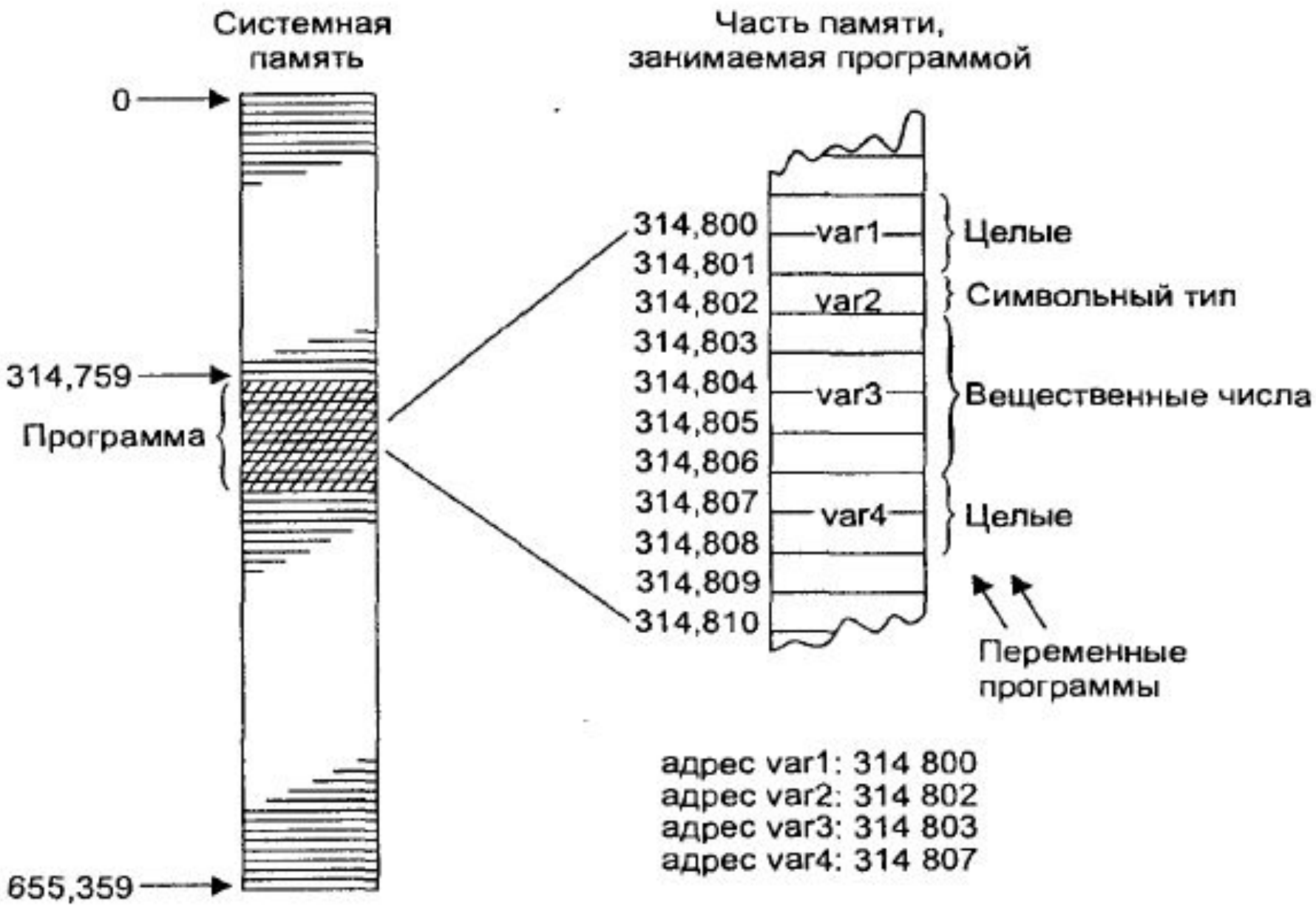
Тема: Указатели в C++

Указатели

Язык C++ позволяет пользователю определить объект, который будет содержать указатель на объекты любого основного или производного типа данных.

Указатели

Адреса и указатели
Каждый байт памяти компьютера имеет адрес.
Адреса — это те же числа, которые мы используем для домов на улице.
Числа начинаются с 0, а затем возрастают — 1, 2, 3 и т. д. Если у нас есть 1 Мбайт памяти, то наибольшим адресом будет число 1 048 575 (хотя обычно памяти много больше). Загружаясь в память, наша программа занимает некоторое количество этих адресов. Это означает, что каждая переменная и каждая функция нашей программы начинается с какого-либо конкретного адреса


Системная память
Часть памяти, занимаемая программой
0
314,759
Программа
655,359
314,800
314,801
314,802
314,803
314,804
314,805
314,806
314,807
314,808
314,809
314,810
var1
var2
var3
var4
Целые
Символьный тип
Вещественные числа
Целые
Переменные программы
адрес var1: 314 800
адрес var2: 314 802
адрес var3: 314 803
адрес var4: 314 807

Указатели

1. Операция получения адреса &

Мы можем получить адрес переменной, используя **операцию получения адреса &:**

&<имя переменной>

Пример:

```
int var1 = 11;          // определим
int var2 = 22;      //переменные
int var3 = 33;
// напечатаем адреса этих переменных
   cout << &var1 << endl
        << &var2 << endl
        << &var3 << endl;
```

Указатели

Реальные адреса, занятые переменными в программе, зависят от многих факторов, таких, как компьютер, на котором запущена программа, размер оперативной памяти, наличие другой программы в памяти и т. д. По этой причине вы, скорее всего, получите совершенно другие адреса при запуске этой программы
(вы даже можете не получить одинаковые адреса, запустив программу несколько раз подряд).

Указатели

На экране, например:

```
0x8f4ffff4 - адрес переменной var1
0x8f4ffff2 - адрес переменной var2
0x8f4ffff0 - адрес переменной var3
```

ЗАМЕЧАНИЕ: адреса переменных — это не то же самое, что их значение.

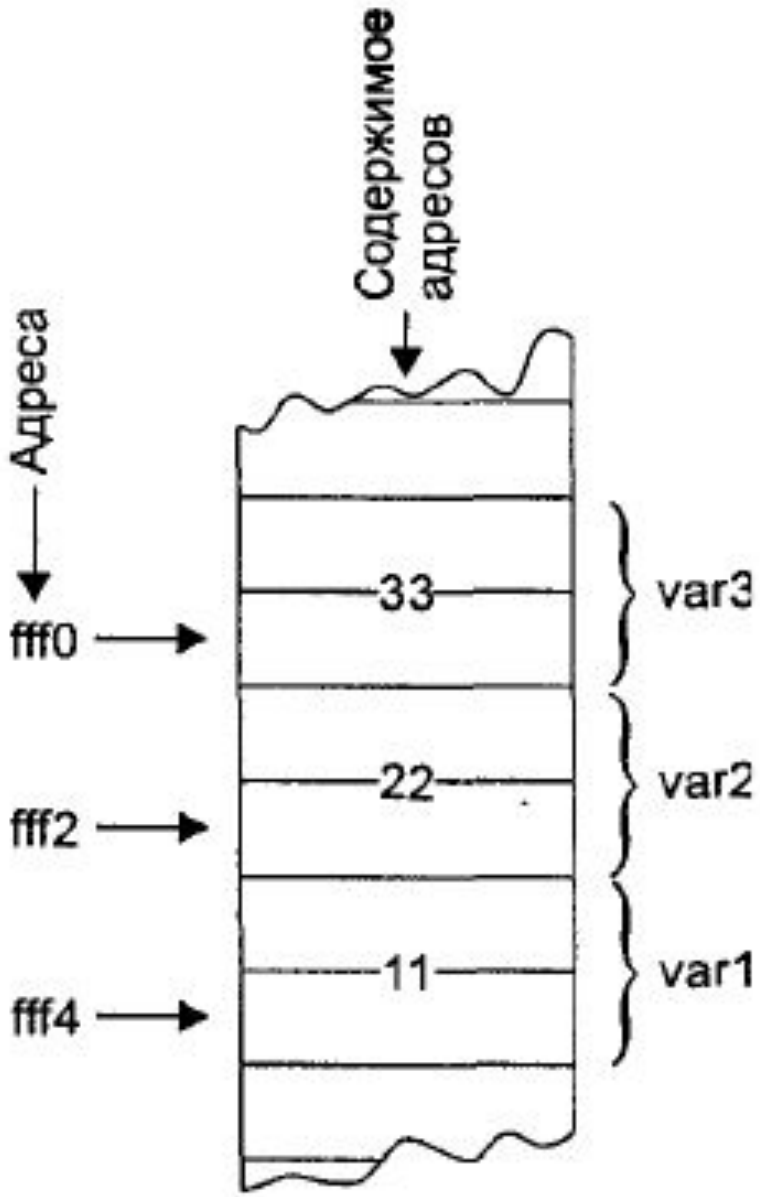


Рис. Адреса и содержимое переменных

Указатели

2. Переменные указатели

Указатель - это переменная, содержащая адрес памяти, где расположены другие объекты (переменные, функции и т.п.).

Тип переменной-указателя: она не того же типа, что и переменная, адрес которой хранит, т.е. указатель на `int` ***не имеет типа*** ***`int`***.

Указатели

Синтаксис определения указателя:

```
<класс_хранения> <спецификация_типа>
*<имя_указателя>;
```

Упрощенный синтаксис:

```
<тип> *<имя_указателя>;
```

где * обозначает указатель на…

Например:

```
char *ptr; // ptr – указатель
           // на char,
/*то есть эта переменная может содержать
в себе адрес переменной типа char*/
```

Указатели

ЗАМЕЧАНИЕ: можно писать

```
<тип>*  идентификатор;
```

Однако принято * устанавливать перед именем переменной-указателя

```
// три переменных указателя
char* ptr1, * ptr2, * ptr3;
char* ptr1, ptr2, ptr3; //не верно!

char *ptr1, *ptr2, *ptr3;
```

Указатели

3. Технология применения указателей

1)Объявить указатель

Например,

```
int *ip; // ip – указатель на int
```

ЗАМЕЧАНИЕ: Компилятору нужны сведения о том, какого именно типа переменная, на которую указывает указатель, поэтому **адрес, который помещается в указатель, должен быть того же типа, на который ссылается указатель.**

Указатели

2) Присвоить указателю значение адреса

Унарный оператор & выдает **адрес объекта, расположенного в памяти** - его операндом не может быть ни выражение, ни константа.

```
<имя указателеля> = &<имя переменной>;
```

```
int x;
ip=&x; //помещаем в ip адрес переменной x
```

Говорят, что ***ip*** указывает на ***x***

или ***ip*** ссылается на ***x***

Указатели

3) Применить доступ к переменным по указателю

Унарный оператор * - операция разыменования означает взять значение переменной, на которую указывает указатель.

```
а) *<имя_указателя>
```

Т.е. примененный к указателю он выдает объект, на который данный указатель ссылается.

```
б) *<имя_указателя> = <выражение>;
в)<имя_переменой> = *<имя указателя>[в составе выражения];
```

Указатели

```
cout << *ip; /* а)то же самое, что
cout<<x; Т.е. показываем содержимое
переменной через указатель*/

*ip = 10; //б) то же самое, что x = 10;
int y;
y = *ip; //в) то же самое, что y = x;
y = *ip + 15; //то же самое, что y=x+15;

int z[10];
ip = &z[0]; //ip теперь указывает на z[0]
```

Указатели

Доступ к значению переменной, хранящейся по адресу, с использованием операции разыменования называется **непрямым доступом** или **разыменованием указателя**.

```
int v;   // определим переменную v типа int
int* p; // определим переменную типа указатель на int
p = &v; // присвоим переменной p значение адреса переменной v
v = 3;   // присвоим v значение 3
*p = 3; // сделаем то же самое, но через указатель
```

Указатели

4. Арифметические операции над указателями

Если ***ip*** ссылается на x целого типа, то **** ip*** можно использовать в любом месте, где допустимо применение ***x.***

```
*ip=*ip+1; // увеличивает на 1 то, на
     // что ссылается ip, т.е. x + 1
```

ЗАМЕЧАНИЕ: Унарные операторы * и & имеют **более высокий приоритет**, чем арифметческие операторы.

Указатели

```
++*ip; // увеличивает x на 1
(*ip)++; // скобки необходимы, т.к.
если их не будет, увеличится значение
самого указателя, а не того, на что он
ссылается
```

ЗАМЕЧАНИЕ: Унарные операторы * и ++ имеют **одинаковый приоритет**.

Если iq – указатель на целое, то можно

```
iq=ip; /*копирует содержимое ip в iq,
чтобы ip и iq ссылались на один и тот
же объект*/
```

Указатели

Указатели можно использовать как операнды в арифметических операциях:

если **y** - указатель, то унарная операция **y++**; увеличивает его значение; теперь оно является адресом следующего элемента.

Указатели и целые числа можно складывать:

если **y** - указатель, **n** - целое число, то

```
y+n;  /* задает адрес n-го объекта, на который указывает y*/
```

Указатели

Любой адрес можно проверить на равенство (==) или неравнство (!=) со специальным значением NULL, которое позволяет определить ничего не адресующий указатель.

Указатели можно сравнивать с помощью операций отношения (<, >, <=, >=, != и ==).

ЗАМЕЧАНИЕ: в сравнении должны участвовать указатели, которые адресуются к данным одного и того же типа.

Указатели

5. Примеры использования указателей:

♦ доступ к элементам массива;

♦ передача аргументов в функцию, от которой требуется изменить эти аргументы;

♦ передача в функции массивов и строковых переменных;

♦ выделение памяти;

♦ создание сложных структур, таких, как связный список.

Указатели

Указатели и массивы

Связь между массивами и указателями

а) доступ к элементу массива через операцию индексирования или через указатель

По определению *имя массива* - *это адрес его нулевого элемента*.

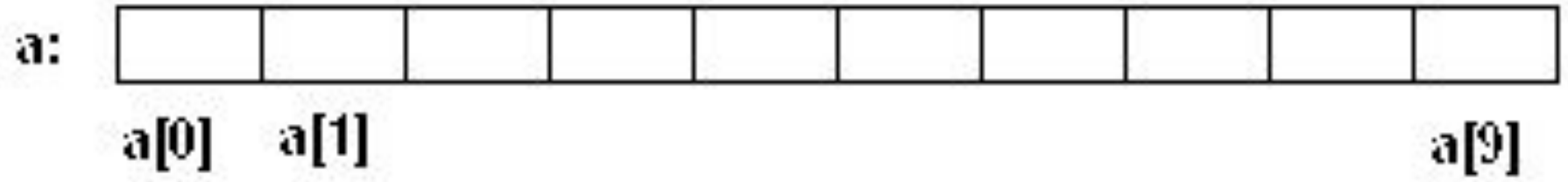
Указатели

1 способ: доступ к элементу мас. через индексацию

- Объявление массива

```
int a[10];
```

определяет массив `a` размера 10, т.е. блок из 10 последовательных объектов с именами `a[0], a[1], ..., a[9]`



- Обращение к элементу массива в выражениях

  `a[i]` – отсылает к `i`-му элементу массива.

- Присваивание

```
int x =  a[0]; // копирование a[0] в x
int y =  a[i]; // копирование a[i] в y
```

Указатели

2 способ: доступ к элементу масс. через указатель

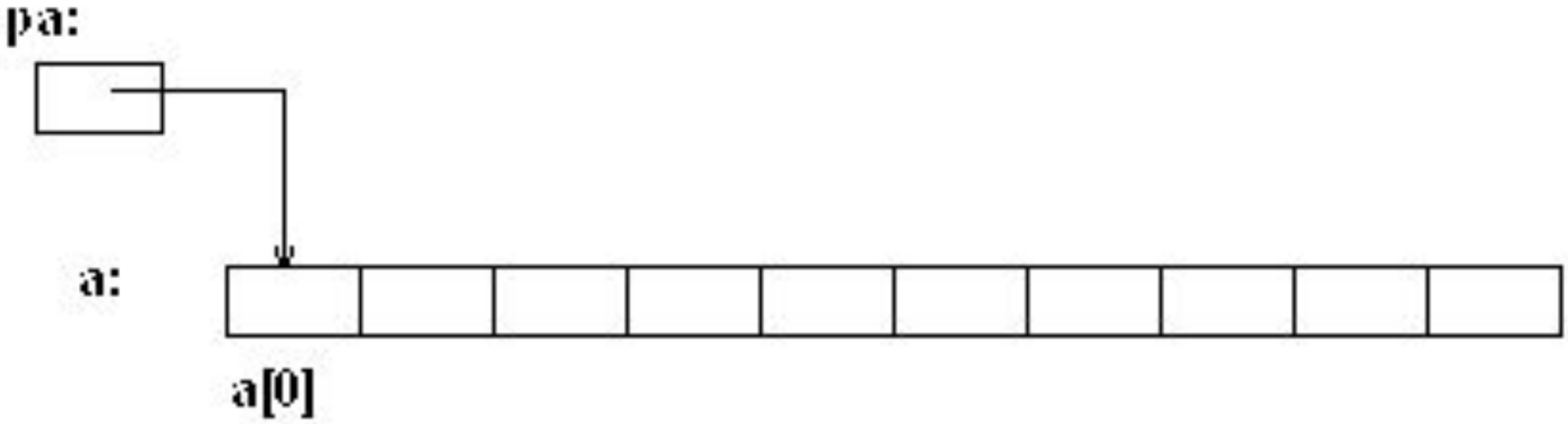
- Объявление массива и указателя

```
int a[10];
int *pa;  // pa есть указатель на int
```

- Присвоить указателю значение адреса эл-та

```
pa = &a[0];
```

Так, **pa** будет указывать на нулевой элемент **a**, т.е. **pa** будет содержать адрес элемента **a[0]**



Указатели

ЗАМЕЧАНИЕ: Если `pa` указывает на некоторый элемент массива, то `pa+1` по определению указывает на следующий элемент, `pa+i` - на `i`-ый элемент после `pa`, а `pa-i` - на `i`-ый элемент перед `pa`.

- Обращение к элементу массива в выражениях

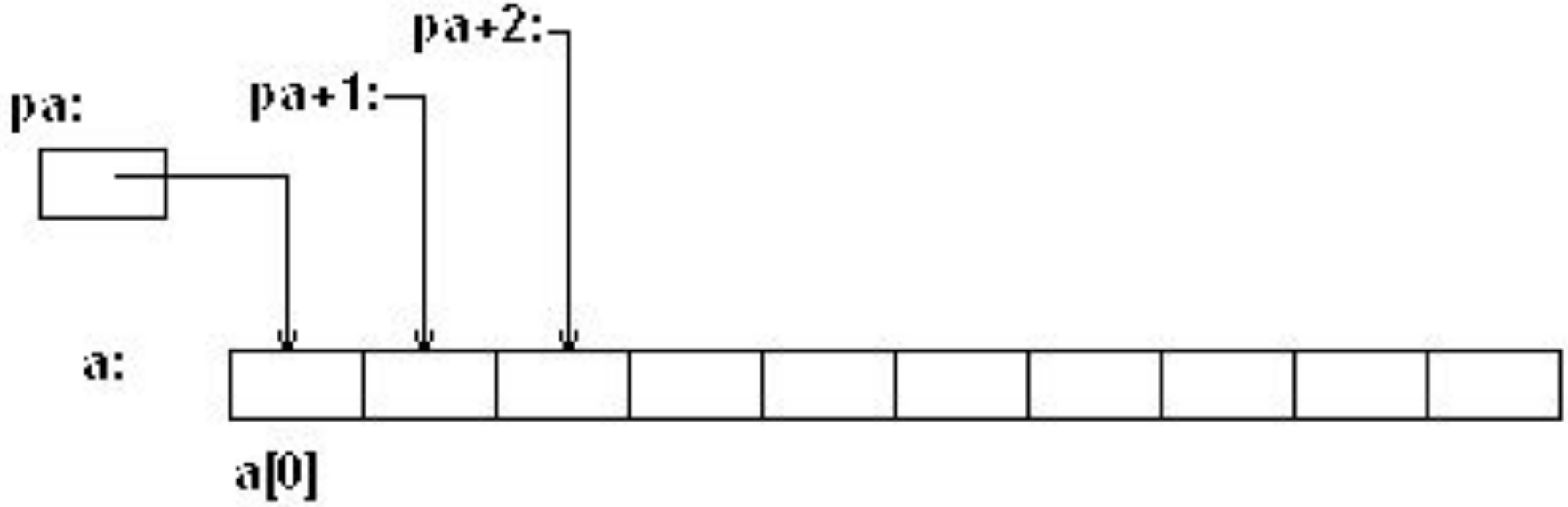
  `*(pa+i)` – отсылает к `i`-му элементу массива, если `pa` указывает на первый элемент.

Указатели

ЗАМЕЧАНИЕ: Таким образом, если `pa` указывает на `a[0]`, то `*(pa+1)` есть содержимое `a[1]`, `pa+i` – адрес `a[i]`, а `*(pa+i)` - содержимое `a[i]`.

- Присваивание

```
int x = *pa; //  копирование a[0] в x

int y = *(pa + i); //копирование a[i] в y
```



Указатели

б) Доступ к элементу массива через указатель на элемент или через указатель, выступающий в роли имени массива, с использованием арифметики с указателями

По определению *имя массива* - *это адрес его нулевого элемента*, тогда

1) `pa=&a[0];` **`эквивалентно`** `pa=a;`
2) `a[i]` **`эквивалентно`** `*(a+i)`

ЗАМЕЧАНИЕ: встречая запись `a[i]`, компилятор сразу преобразует ее в `*(a+i)`

3) `&a[i]` **`эквивалентно`** `a+i`

Замечание: это адрес `i`-го элемента после `a`

4) `pa[i]` **`эквивалентно`** `*(pa+i)`

Указатели

в) Различие между именем массива и указателем, выступающим в роли имени массива

Указатель - это переменная, поэтому можно написать `pa = a` или `pa++`

Но имя массива является константой, и записи типа `a = pa` или `a++` не допускаются.

Указатели

Указатели и функции

Передача аргументов функции может быть произведена тремя путями:

- по значению
- по ссылке
- по указателю

ЗАМЕЧАНИЕ: Если функция предназначена для изменения переменной в вызываемой программе, то эта переменная не может быть передана по значению, так как функция получает только копию переменной - следует использовать передачу переменной по ссылке и по указателю.

Указатели

1. Передача простой переменной
а) передача аргументов по ссылке

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

int main()
{
  void centimize(double &); // прототип функции

  double  var = 10.0;       // значение переменной var равно 10 (дюймов)
  cout << "var = " << var << "дюймов" << endl;

  centimize(var);           // переведем дюймы в сантиметры
  cout << "var = " << var << "сантиметров" << endl;

  return 0;
}
//////////////////////////////////////////////////////////
void centimize(double & v)
{
  v *= 2.54;                // v – это то же самое, что и var
}
```

Указатели

На экране:

```
var = 10 дюймов
var = 25.4 сантиметров
```

ЗАМЕЧАНИЕ: В функцию передается сама переменная var: вызов функции

```
centimize(var);
```

Функция `centimize()` умножает первоначальное значение переменной на 2.54. Для ссылки на переменную функция просто использует имя аргумента v; v и var — это различные имена одного и того же.

Указатели

б) передача аргументов по указателю

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

int main()
{
  void centimize(double *); // прототип функции

  double  var = 10.0;        // значение переменной var равно 10 (дюймов)
  cout << "var = " << var << "дюймов" << endl;

  centimize(&var);           // переведем дюймы в сантиметры
  cout << "var = " << var << "сантиметров" << endl;

  return 0;
}
//////////////////////////////////////////////////////////
void centimize(double * ptrd)
{
  *ptrd *= 2.54;             // *ptrd – это то же самое, что и var
}
```

Указатели

На экране получаем тот же результат.

ЗАМЕЧАНИЕ: В функцию передается адрес переменной var: вызов функции

```
centimize(&var);
```

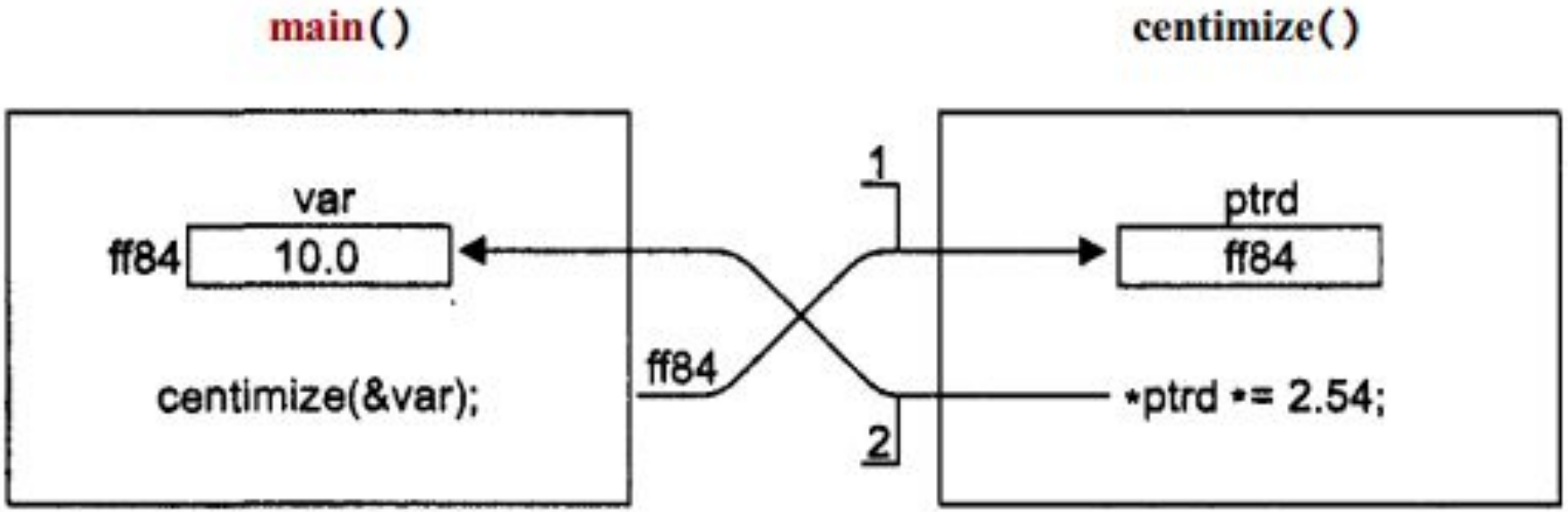
Т.к. функция `centimize()` получает адрес, то она может использовать операцию разыменования `*ptrd` для доступа к значению, расположенному по этому адресу:

```
*ptrd *= 2.54; // умножаем содержимое переменной
               // по адресу ptrd на 2.54
```

Это то же самое, что

```
*ptrd = *ptrd * 2.54;
```

Указатели



1. main() передает адрес переменной var в ptrd в centimize()
2. centimize() использует этот адрес для доступа к var

Рис. Передача в функцию по указателю

ЗАМЕЧАНИЕ: Ссылка — это псевдоним переменной, а указатель — это адрес переменной.

Указатели

2. Передача массивов по указателю

а) передача массива по значению

```
const int MAX = 5; // кол-во элементов
void centimize(double []);/* прототип ф-и
перевода всех элементов массива в см */
int main()
{
   double varray[MAX] =
   { 10.0, 43.1, 95.9, 58.7, 87.3 };
   centimize(varray); /*вызов ф-ии */
   for(int j = 0; j < MAX; j++)
   cout <<  varray[j] << " см" << endl;
   return 0;
}
```

Указатели

```
void centimize(double arr[])//опр-е ф-и
{
   for(int j = 0; j < MAX; j++)
       arr[j] *= 2.54;
}
```

ЗАМЕЧАНИЕ: при вызове функции передается имя массива:

```
centimize(varray);
```

В функции `centimize()` поочередно присваиваются новые значения всем элементам массива

```
arr[j] *= 2.54;
//т.е. arr[j] = arr[j] * 2.54;
```

Указатели

б) передача массива по указателю

```
const int MAX = 5; // кол-во элементов
void centimize(double *);/* прототип ф-и
перевода всех элементов массива в см */
int main()
{
   double varray[MAX] =
   { 10.0, 43.1, 95.9, 58.7, 87.3 };
   centimize(varray); /*вызов ф-ии */
   for(int j = 0; j < MAX; j++)
   cout <<  varray[j] << " см" << endl;
   return 0;
}
```

Указатели

```
void centimize(double * ptrd)//опр-е ф-и
{
   for(int j = 0; j < MAX; j++)
      *ptrd++ *= 2.54;
}
```

ЗАМЕЧАНИЕ: т.к. имя массива является его адресом то не используется операция взятия адреса & при вызове функции:

```
centimize(varray); // перед-ся адрес м-ва
```

В функции `centimize()` адрес массива присваивается переменной `ptrd`: используется операция увеличения для указателя `ptrd`, чтобы указать на все элементы массива по очереди:

```
*ptrd++ *= 2.54;
```

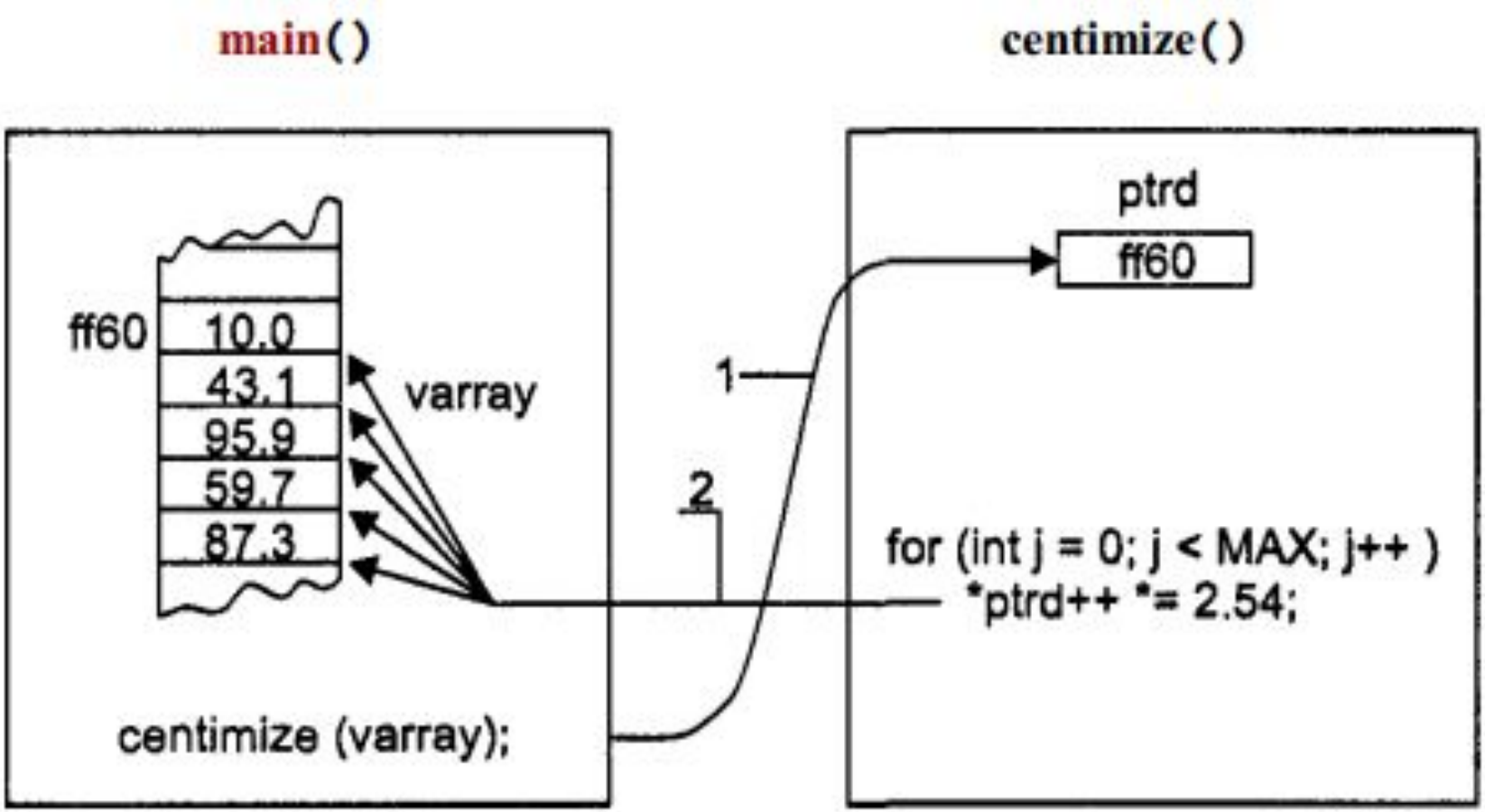
Указатели

Выражение *ptrd++ интерпретируется как *(ptrd++) и увеличивает указатель, а не то, на что он указывает. Таким образом, сначала увеличивается указатель, а затем к результату применяется операция разыменования.

На экране:

```
25.4 сантиметров
109.474 сантиметров
243.586 сантиметров
151.638 сантиметров
221.742 сантиметров
```

Указатели



1. **main() передает адрес varray в ptrd функции centimize()**
2. **centimlze( ) использует этот адрес для доступа ко всем элементам массива по очереди**

Рис. Доступ к массиву из функции

Указатели

Как узнать, что в выражении *ptrd++ увеличивается указатель, а не его содержимое? Другими словами, как компилятор интерпретирует это выражение: как *(ptrd++), что нам и нужно, или как (*ptrd)++?

Здесь * (при использовании в качестве операции разыменования) и ++ имеют одинаковый приоритет. Однако операции одинакового приоритета различаются еще и другим способом: ***ассоциативностью.*** **Ассоциативность определяет, как компилятор начнет выполнять операции, справа или слева. В группе операций, имеющих правую ассоциативность,**

Указатели

Сортировка элементов массива с использованием указателей для доступа к элементам массива

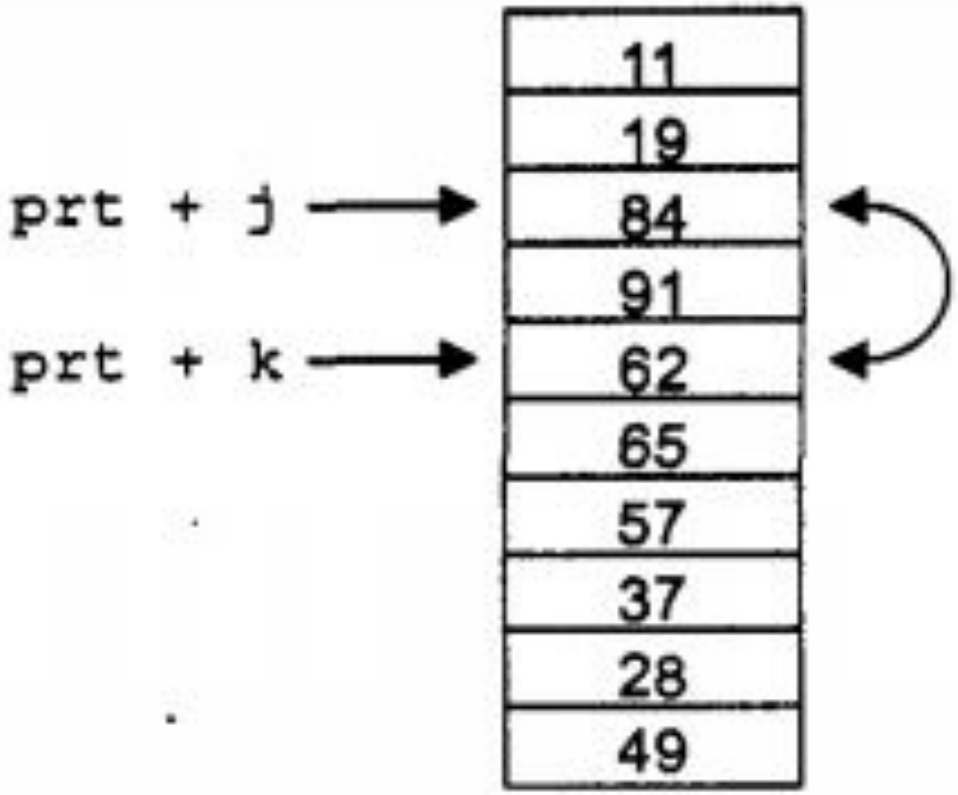
```
int main()
{
  void bsort(int*, int);     // прототип функции
  const int N = 10;          // размер массива
  // массив для сортировки
  int arr[N] = { 37, 84, 62, 91, 11, 65, 57, 28, 19, 49 };
  bsort(arr, N);

  for(int j = 0; j < N; j++)
    cout << arr[j] << " ";
  cout << endl;

  return 0;
}
```

Указатели

```
void bsort(int* ptr, int n)
{
  void order(int*, int*);           // прототип функции
  int j, k;                         // индексы в нашем массиве

  for(j = 0; j < n - 1; j++)        // внешний цикл
    for(k = j + 1; k < n; k++)      // внутренний цикл
      order(ptr + j, ptr + k);      // сортируем элементы
}
////////////////////////////////////////////////////////////
void order(int* numb1, int* numb2) // сортировка двух чисел
{
  if(*numb1 > *numb2)  // если первое число больше, то меняем
  {                                // их местами
    int temp = *numb1;
    *numb1   = *numb2;
    *numb2   = temp;
  }
}
```

Указатели

ЗАМЕЧАНИЕ: Адрес массива и номера его элементов передаются в функцию bsort()



Указатели

Указатели на строки

Строки — это просто массивы элементов типа **`char`**.
Таким образом, доступ через указатели может быть применен к элементам строки так же, как и к элементам массива.

Также для описания строковых данных применяются так называемые **динамические массивы**, работа с которыми производится с помощью указателей.
Например: **`char* str = “строка”;`**

Указатели

Практически все функции работы со строками возвращают или имеют в качестве параметров данные типа char*.

При этом в качестве фактических параметров им могут передаваться обычные (статические) символьные массивы.

Указатели

1) Указатели на строковые константы

```
int main()
{
  char str1[] = "Определение через массив";
  char* str2 = "Определение через указатель";

  cout << str1 << endl; // покажем наши строки
  cout << str2 << endl;

 // str1++;              // так делать нельзя
  str2++;               // а так можно

  cout << str2 << endl; // значение str2 немного изменилось

  return 0;
}
```

Указатели

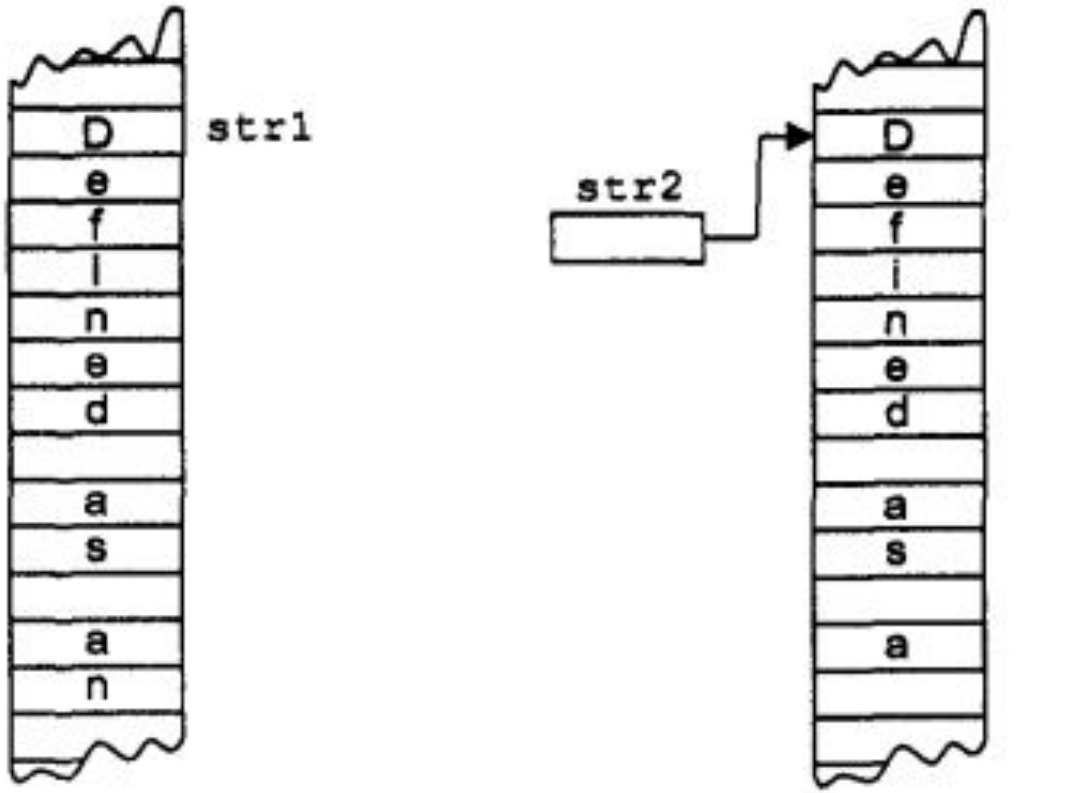
Замечание: str1 — это адрес, то есть указатель-константа, а str2 — указатель-переменная. Поэтому str2 может изменять свое значение, а str1 нет, что показано в программе.

Мы можем увеличить str2, так как это указатель, но после этой операции он уже больше не будет показывать на первый элемент строки.

На экране:

```
Определение через массив
Определение через указатель
Определение через указатель
```

Указатели



Строка, определенная как массив

Строка, определенная как указатель

```
char str1[] = "Def....
```

```
char* str2 = "Def...
```

Рис. Строки как массив и как указатель

Указатели

2) Строки как аргументы функций

```
int main()
{
  void dispstr(char*);  // прототип функции
  char str[] = "У бездельников всегда есть свободное время.";

  dispstr(str);

  return 0;
}
void dispstr(char* ps)
{
  while(*ps)              // пока не встретим конец строки
    cout << *ps++;        // будем посимвольно выводить ее на экран
  cout << endl;
}
```

Указатели

ЗАМЕЧАНИЕ: Адрес массива str использован как аргумент при вызове функции dispstr().
Этот адрес является константой, но так как он передается по значению, то в функции dispstr() создается его копия.
Это будет указатель ps. Он может быть изменен, и функция увеличивает его, выводя строку на дисплей.
Выражение *ps++ возвращает следующий знак строки. Цикл повторяется до появления знака конца строки ('\0'). Так как он имеет значение 0, которое интерпретируется как
false, то в этот момент цикл заканчивается.

Указатели

3) Копирование строк с использованием указателей

Указатели можно использовать не только для получения значений элементов массива, но и для вставки значений в массив.

```
int main()
{
  void copystr(char*, const char*); // прототип функции
  char* str1 = "Поспешишь - людей насмешишь!";
  char str2[80];                    // пустая строка

  copystr(str2, str1);    // копируем строку str1 в str2
  cout << str2 << endl;   // и показываем результат

  return 0;
}
```

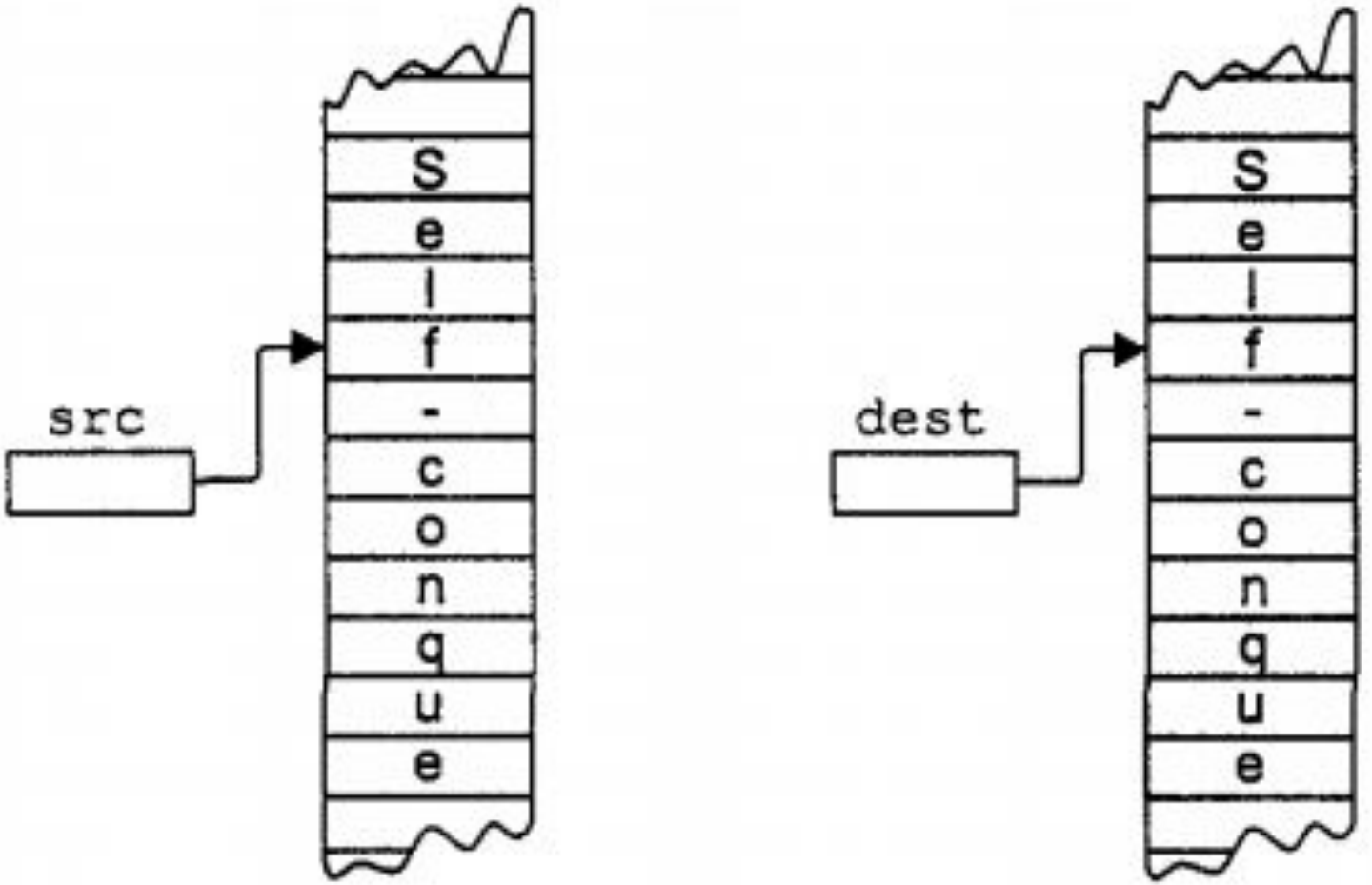
Указатели

```
void copystr(char* dest, const char* src)
{
  while(*src)          // пока не встретим конец строки
    *dest++ = *src++; // копируем ее
  *dest = '\0';        // заканчиваем строку
}
```

ЗАМЕЧАНИЕ: функция copystr() копирует str1 в str2: **`*dest++ = *src++;`**

Значение src помещается по адресу, на который указывает dest. Затем оба указателя увеличиваются и на следующей итерации передается следующий символ. Цикл прекращается, когда в src будет найден символ конца строки; в этой точке в dest присваивается значение null, работа функции завершена.

Указатели



***dest++ = *src++;**

Рис. Движение указателей по строкам

Указатели

4) Библиотека строковых функций

Многие из библиотечных функций для строк имеют строковые аргументы, которые определены с использованием указателей.

Синтаксис библиотечной фунции strcpy();

```
char* strcpy(char* dest, const char* src);
```

Функция имеет два аргумента типа char*.
Функция strcpy() возвращает указатель на char; это адрес строки dest.

Указатели

Функции модуля **stdlib.h**

В данном модуле содержится ряд функций, предназначенных для преобразования строковых данных в числовые.

```
int atoi(char* str);
```
Преобразует строку str в десятичное число.

```
char* itoa(int v, char* str, int baz);
```
Преобразует целое v в строку str.
При изображении числа используется основание baz (2≤baz ≤ 36). Для отрицательного числа и baz=10 первый символ – «минус» (–).

Указатели

```
long atol(char* str);
```

Преобразует строку str в длинное десятичное число.

```
char* ltoa(long v, char* str, int baz);
```

Преобразует длинное целое v в строку str. При изображении числа ис- пользуется основание baz (2≤baz ≤ 36).

```
char* ultoa(unsigned long v, char* str,
int baz);
```

Преобразует беззнаковое длинное целое v в строку str.

Указатели

```
double atof(char* str);
```

Преобразует строку str в вещественное число типа double.

```
char* gcvt(double v, int ndec, char* str);
```

Преобразует вещественное число v типа double в строку str.
Количество значащих цифр задается параметром ndec

Указатели

Функции модуля **`string.h`**

В данном модуле описаны функции, позволяющие производить различные действия над строками: преобразование, копирование, сцепление, поиск и т.п.

```
char* strcat(char* sp,char* si);
```

Добавляет строку si к строке sp (конкатенация строк).

```
char* strncat(char* sp,char* si,int kol);
```

Добавляет kol символов строки si к строке sp (конкатенация).

Указатели

```
char* strdup(const char* str);
```
Выделяет память и переносит в нее копию строки str.

```
char* strset(char* str,int c);
```
Заполняет строку str заданным символом c.

```
char* strnset(char* str,int c,int kol);
```
Заполняет первые kol символов строки str заданным символом c.

```
unsigned strlen(char* str);
```
Вычисляет длину строки str.

Указатели

```
char* strcpy(char* sp,char* si);
```

Копирует байты строки si в строку sp.

```
char* strncpy(char* sp,char* si,int
kol);
```

Копирует kol символов строки si в строку sp («хвост» отбрасывается или дополняется пробелами).

```
int strcmp(char* str1,char* str2);
```

Сравнивает строки str1 и str2.
Результат отрицателен, если str1str2 (сравнение беззнаковое).

Указатели

```
int strncmp(char* str1,char* str2,int kol);
```

Аналогично предыдущему, но сравниваются только первые kol символов.

```
int stricmp(char* str1,char* str2,int kol);
```

Аналогично предыдущему, но при сравнении не делается различия регистров.

```
char* strchr(char* str,int c);
```

Ищет в строке str первое вхождение символа c.

Указатели

```
char* strrchr(char* str,int c);
```

Ищет в строке str последнее вхождение символа c.

```
char* strstr(const char* str1,const
char* str2);
```

Ищет в строке str1 подстроку str2.
Возвращает указатель на тот эле- мент в строке str1, с которого начинается подстрока str2.

```
char* strpbrk(char* str1,char* str2);
```

Ищет в строке str1 первое появление любого из множества символов, входящих в строку str2.

Указатели

```
int strspn(char* str1,char* str2);
```

Определяет длину первого сегмента строки str1, содержащего только символы из множества символов строки str2.

```
int strcspn(char* str1,char* str2);
```

Определяет длину первого сегмента строки str1, содержащего симво- лы, не входящие во множество символов строки str2.

```
char* strtok(char* str1,const char*
str2);
```

Ищет в строке str1 лексемы, выделенные символами из второй строки.

Указатели

```
char* strlwr(char* str);
```

Преобразует буквы верхнего регистра в строке в соответствующие буквы нижнего регистра.

```
char* strupr(char* str);
```

Преобразует буквы нижнего регистра в строке str в буквы верхнего регистра.

Указатели

Функции модуля ctype.h

Данный модуль содержит функции проверки и преобразования символов, которые могут быть полезны при посимвольной обработке строк.

int isalnum(int c);
Дает значение не ноль, если c – код буквы или цифры (A-Z, a-z, 0-9), и ноль – в противном случае.

int isalpha(int c); Дает значение не ноль, если c – код буквы (A-Z, a-z), и ноль – в противном случае.

Указатели

Дает значение не ноль, если с – управляющий символ с кодами 0x00- 0x1F или 0x7F, и ноль – в противном случае.

int isdigit (int c);
Дает значение не ноль, если с – цифра (0-9) в коде ASCII, и ноль – в противном случае.

int isgraph(int c);
Дает значение не ноль, если с – видимый (отображаемый) символ с ко- дом 0x21-0x7E, и ноль – в противном случае.

int islower(int c);

Указатели

Дает значение не ноль, если c – символ-разделитель (соответствует iscntrl или isspace), и ноль – в противном случае.

int isspace(int c);
Дает значение не ноль, если c – обобщенный пробел: пробел, символ табуляции, символ новой строки или новой страницы, символ возврата каретки (0x09-0x0D, 0x20), и ноль – в противном случае.

int isupper (int c);
Дает значение не ноль, если c – код символа в верхнем регистре (A-Z), и ноль – в противном

Указатели

Преобразует целое число с в символ кода ASCII, обнуляя все биты, кроме младших семи. Результат от 0 до 127.

int tolower(int c);
Преобразует код буквы с к нижнему регистру, остальные коды не из- меняются.

int toupper(int c);
Преобразует код буквы с к верхнему регистру, остальные коды не из- меняются.

Указатели

5) Модификатор const и указатели

Варианты объявления указателей:

1. Указатель на константу

```
const int* cptrInt;
```

ЗАМЕЧАНИЕ: нельзя изменять значение переменной, на которую указывает указатель `cptrInt`, но можно изменять значение самого указателя `cptrInt`.

Указатели

2. Константный указатель

```
int* const ptrcInt;
```

ЗАМЕЧАНИЕ: нельзя изменять значение самого указателя `cptrInt`, но можно изменять значение того, на что `cptrint` указывает.

Можно использовать `const` в обеих позициях и сделать константами как сам указатель, так и то, на что он указывает.

Указатели

В дополнение к описанию функции strcpy():

```
char* strcpy(char* dest, const char* src);
```

В объявлении функции strcpy() показано, что параметр const char* scr определен так, что функция не может изменять строку, на которую указывает scr.
Это не значит, что указатель scr не может быть изменен. Для того чтобы указатель стал константой, нужно при его объявлении указать char* const scr.

Указатели

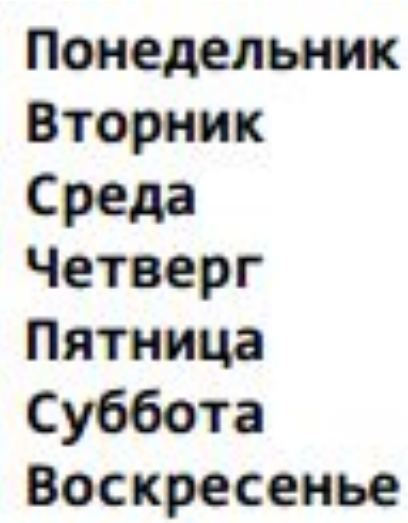
6) Массивы указателей на строки

Массив указателей — это то же, что и массив переменных типа int или float.

```
const int DAYS = 7;

int main()
{
  // массив указателей на строки
  char* arrptrs[DAYS] = { "Понедельник", "Вторник", "Среда",
 "Четверг", "Пятница", "Суббота", "Воскресенье" };

  for(int j = 0; j < DAYS; j++)    // покажем все строки
    cout << arrptrs[j] << endl;

  return 0;
}
```

Указатели

На экране:



ЗАМЕЧАНИЕ: В том случае, если строки не являются частью массива, то C++ размещает их в памяти друг за другом, чтобы не было пустого пространства. Но для поиска строк создается массив, содержащий указатели на них. Сами строки — это массивы элементов типа char, поэтому массив указателей на строки является массивом указателей на char. А, т.к. адресом строки является адрес ее первого элемента, то эти адреса и хранит массив указателей на строки

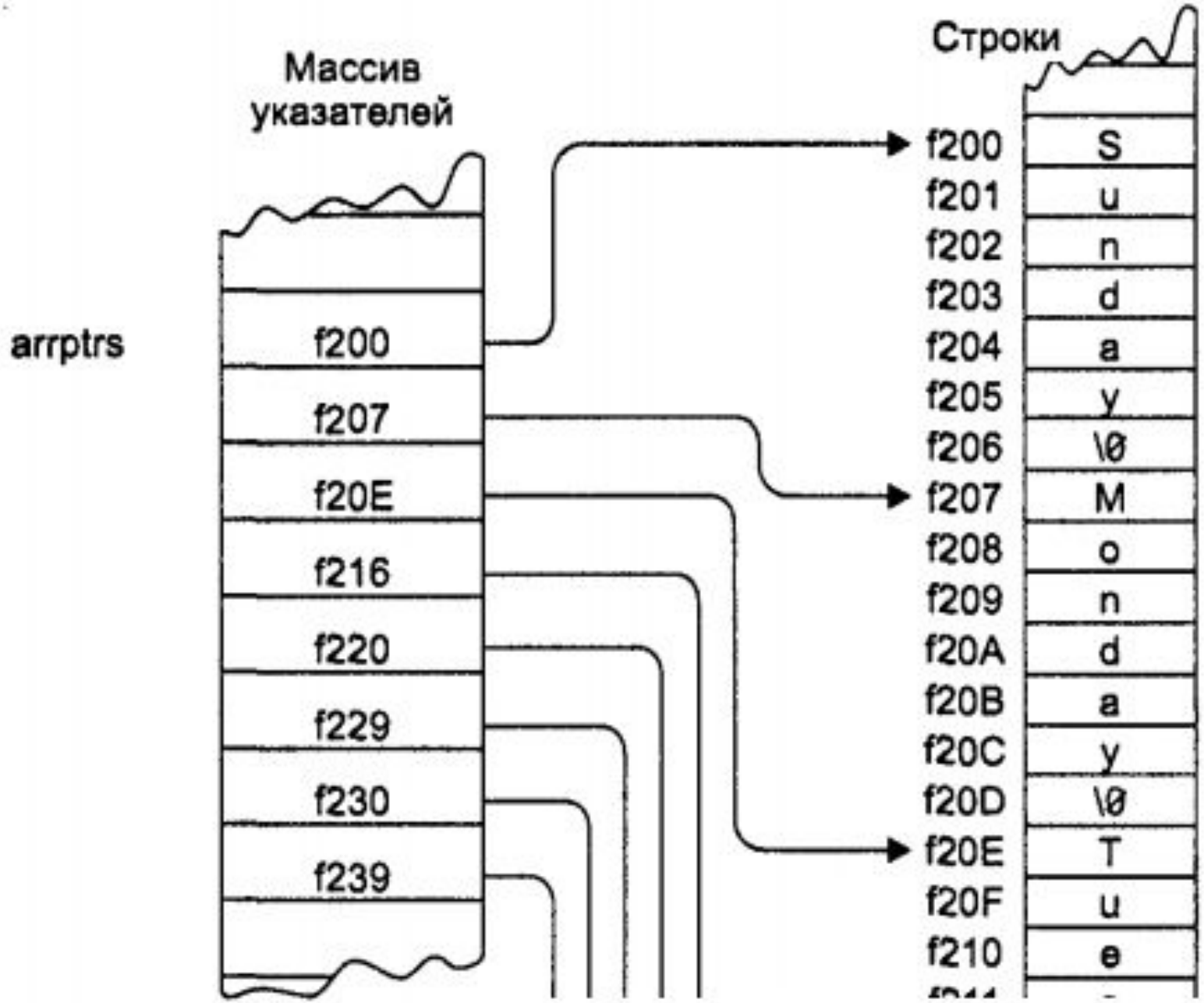


Рис. Массив указателей и строки

Указатели

Указатели и структуры

Можно определять указатели на структуры:
имя_структуры *имя_указателя на структуру;
Доступ к элементам структуры обеспечивает ***оператор стрелка (->)***. Формат соответствующего выражения следующий:
имя_указателя->имя_элемента_структуры
Оператор стрелка, состоящий из знака минус (-) и знака больше (>), записанных без пробела, обеспечивает доступ к элементам структуры через указатель на объект.

Указатели

Указатель на void

Это тип указателя, который может указывать на любой тип данных.

```
void *<имя_указателя>;
```

Такие указатели предназначены для использования в определенных случаях, например, передача указателей в функции, которые работают независимо от типа данных, на который указывает указатель.

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

int main()
{
  int intvar;             // целочисленная переменная
  float flovar;           // вещественная переменная

  int* ptrint;            // указатель на int
  float* ptrflo;          // указатель на float
  void* ptrvoid;          // указатель на void

  ptrint = &intvar;       // так можно: int* = int*
// ptrint = &flovar;       // так нельзя: int* = float*
// ptrflo = &intvar;       // так нельзя: float* = int*
  ptrflo = &flovar;       // так можно: float* = float*

  ptrvoid = &intvar;      // так можно: void* = int*
  ptrvoid = &flovar;      // так можно: void* = float*

  return 0;
}
```

Указатели

ЗАМЕЧАНИЕ: нужно знать при написании программы, насколько большой массив нужен. Поэтому подход с вводом размера массива в процессе выполнения программы работать не будет:

```
int size;
cin >> size; // получим желаемый размер
       // массива
int arr[size]; // ошибка, размер
   // массива должен быть константой!
```

Правильно:

```
const int size;
```

Указатели

Управление памятью: операции new и delete

Способы объявления массивов:

1) Объявление массива без учета памяти с помощью задания размера массива как константы:

```
const int size = 100;
int arr1[size];
//или
int arr1[100]; /* зарезервирована
память для 100 элементов типа int*/
```

Указатели

2) Объявление массива с помощью **операции new -** универсальной операции, получающей память у операционной системы и возвращающей указатель на начало выделенного блока.
При этом происходит создание **динамического массива.**

Указатели

Операторы управления свободной памятью **new** и **delete**

Свободная память - это предоставляемая системой область памяти для объектов, время жизни которых напрямую управляется программистом. Программист создает объект с помощью ключевого слова **new**, а уничтожает его, используя **delete**.

Указатели

Примеры:

1) Простые типы

```
int *p;
p=new int (5); //выделили память и
//инициализировали
```

ЗАМЕЧАНИЕ: указателю на целое p присваивается адрес ячейки памяти, полученный при размещении целого объекта. Место в памяти, на которое указывает p, инициализируется значением 5.

Такой способ обычно не используется для простых типов вроде **int**

Указатели

2) Одномерные Динамические массивы

```
int *q;
q=new int [10]; //получаем массив от
//q[0] до q[9]
```

ЗАМЕЧАНИЕ: использование примера с указателем q на массив встречается значительно чаще.

```
int *arr;
int size;
cin >> size; // получим желаемый размер
        // массива
arr = new int[size];  //выделяем память
//под введенное количество целых чисел
```

Указатели

Формы оператора **new** в C++:

new имя_типа;

new имя_типа (инициализатор);

new имя_типа [выражение];

Производимые эффекты:

Во-первых, выделяется надлежащий объем свободной памяти для хранения указанного типа.
Во-вторых, возвращается базовый адрес объекта (в качестве значения оператора **new**).
Когда память недоступна, оператор **new** возвращает значение 0 (NULL). Следовательно, мы можем контролировать процесс успешного выделения памяти оператором **new**.

Указатели

Оператор **delete** уничтожает объект, созданный с помощью **new**, отдавая тем самым распределенную память для повторного использования.

Формы оператора **delete**:

delete выражение;

delete [ ] выражение;

ЗАМЕЧАНИЕ: Первая форма используется, если соответствующее выражение **new** размещало не массив. Во второй форме присутствуют пустые квадратные скобки, показывающие, что изначально размещался массив объектов. Оператор **delete** не возвращает значения, поэтому можно сказать, что возвращаемый им тип - **void**.

Указатели

При работе с динамически задаваемыми массивами часто забывают освобождать память, захваченную для массива. Память следует вновь возвращать в распоряжение операционной системы, то есть освобождать с помощью операции delete. Правда, при завершении работы функции main автоматически уничтожаются все переменные, созданные в программе, и указатели сегментов памяти получают свои исходные значения. Однако при разработке сложных многомодульных комплексов программ следует помнить о том, что выделенная память «повисает», становится недоступной операционной системе при выходе из области

Указатели

Чтобы освободить память, выделенную для одной переменной d, например, с помощью оператора **double *d = new double;**, достаточно в конце функции или блока, где использовалась переменная d, записать

delete d;

Если был размещен массив переменных, например, **float *p = new float[200]**, то в современных версиях компиляторов следует освобождать память оператором

delete [] p;.

Указатели

Здесь квадратные скобки указывают компилятору на то, что освобождать следует то количество ячеек, которое было захвачено в последней операции **new** в применении к указателю **p**. Явно указывать это число не нужно. Хотя компилятор Visual C++ 6.0 при попытке освободить память, занятую массивом, операцией **delete** без скобок не выдает сообщений об ошибке и, по-видимому, функционирует верно, однако для обеспечения надежности следует соблюдать условия стандарта. Заметим, что операция **delete** игнорирует нулевые указатели, поэтому проверка на неравенство нулю указателя перед

Указатели

Пример использования операторов для динамического распределения памяти.

Создать динамический массив, размер запросить у пользователя. Проверить возможность выделения указанного количества памяти. Заполнить массив случайными числами от 0 до 20.
Подсчитать произведение элементов массива, попадающих в диапазон от 5 до 15.

Указатели

```
#include <iostream>
#include <stdlib.h>

using namespase std;
int main()
{
int *p;    //указатель, в котором будет
      // храниться адрес массива
int size; //переменная – размер массива
cout<< "Vvedite razmer massiva\n";
cin>>size;  //запросили размер
p=new int[size]; //выделяем память под
   // введенное количество целых чисел
```

Указатели

```
if(p==NULL)
{
   cout<<"Nedostatochno pamjati\n";
return 0;
}
                  //если памяти недостаточно,
//выводим сообщение на экран и
//завершаем программу
```

Указатели

```
for(int n=0;n<size;n++)
{
   p[n]=rand()%20;
//заполняем массив случайными числами
//и выводим на экран
     cout<<p[n]<<"\n";
}
```

Указатели

```
    long rez=1;
//переменная, в которой будем хранить
//произведение;
    for(n=0;n<size;n++)
    {
          if(p[n]>5&&p[n]<15)
//если элемент попадает в заданный
//диапазон, добавляем в произведение
          rez*=p[n];
   }
    cout<<"proizvedenie chisel ot 5 do 15
ravno"<<rez<<"\n"; //выводим рез.
    delete[] p;  //освобождаем уже не нужный
нам участок памяти
    return 0;
}
```

Указатели

```
    long rez=1;
//переменная, в которой будем хранить
//произведение;
    for(n=0;n<size;n++)
    {
          if(p[n]>5&&p[n]<15)
//если элемент попадает в заданный
//диапазон, добавляем в произведение
          rez*=p[n];
   }
    cout<<"proizvedenie chisel ot 5 do 15
ravno"<<rez<<"\n"; //выводим рез.
    delete[] p;  //освобождаем уже не нужный
нам участок памяти
    return 0;
}
```

Указатели

3) Многомерные динамические массивы

Указатели на многомерные массивы - это массивы массивов, т.е. такие массивы, элементами которых являются массивы. При объявлении таких массивов в памяти выделяются участки для хранения:

- значения переменной (имя массива), которая является указателем на массив из указателей (количество элементов массива равно количеству строк двумерного массива)
- массива указателей, содержащих адреса элементов в строках двумерного массива (одномерных массивов, размер которых равен количеству столбцов двумерного массива)

Указатели

Принцип организации динамического двухмерного массива `a[m][n]` (схема выделения памяти):

Адрес массива	Адреса массивов адресов	Серия отдельных одномерных массивов		
a	a[0]	a[0][0]	a[0][1]	a[0][n-1]
	a[1]	a[1][0]	a[1][1]	a[1][n-1]
	...	...	...	...
	a[m-1]	a[m-1][0]	a[m-1][1]	a[m-1][n-1]

Указатели

ЗАМЕЧАНИЕ:
Отличие описанной схемы от схемы статического двухмерного массива состоит в том, что теперь для адресов a, a[0], a[1], … a[n - 1] должно быть отведено реальное физическое пространство памяти.

В то время как для статического двухмерного массива выражение вида a, a[0], a[1], … a[n - 1] были всего лишь возможными конструкциями для ссылок на реально существующие элементы массива, но сами эти указатели не существовали как объекты в памяти компьютера.

Указатели

Например, объявление вещественного массива `a[4][3]` порождает в программе три разных объекта:

- указатель с идентификатором `a`
- безымянный массив из четырех указателей
- безымянный массив из двенадцати чисел типа `float`.

ЗАМЕЧАНИЕ: выделение и освобождение памяти для динамических матриц должно выполняться отдельно для матриц целиком, которые являются массивом строк, и отдельно для каждой строки, которые состоят из массивов элементов.

Указатели

Алгоритм выделения памяти:

1. Определяем переменную `a` как адрес массива адресов:
   ```
   float **a;//указатель для мас-ва указателей
   ```
2. Выделяем область памяти для массива из `m` указателей на тип `float` и присваиваем адрес начала этой памяти указателю `a`:
   ```
   a = new float* [m];
   ```
3. В цикле проходим по массиву адресов `a[]`, присваивая каждому указателю `a[i]`, адрес вновь выделяемой памяти под массив из `n` чисел типа `float`.

Указатели

Например:

```
int** a;
a = new float* [m];  //выделяем память

for (int i=0;i<m;i++)
{
   a[i] = new int[n];
     //заполняем случайными числами
   for (int j=0;j<n;j++)
        a[i][j] = rand()%10-5;
}
```

Указатели

По аналогии с массивами, можно получать **доступ к элементам матриц через указатели**:

```
a[i][j] == *(*(a+i)+j) == *(a[i]+j)
==  (*(a+i))[j]
```

ЗАМЕЧАНИЕ: Следует учитывать, что с точки зрения синтаксиса языка СИ указатель `a` и указатели `a[i]`-е являются константами и их значения нельзя изменять во время выполнения программы.

Указатели

Алгоритм освобождения памяти:

```
for (int i=0;i<m;i++)
   delete a[i]; //массив m-указателей
delete []a; //указатель на массив
        //указателей
```

Указатели

Передача матриц в качестве формальных параметров функций

Самым оптимальным методом передачи матриц в качестве формальных параметров функций является использование указателей и динамических массивов:

```
void func(int** x,int m,int n)…
```

Сама матрица может формироваться в главной функции программы как совокупность одномерных динамических массивов строк матрицы и динамического массива указателей на эти массивы-строки.