Основы программирования на языке C++

Преподаватель:
Маслов Алексей Владимирович,
доцент кафедры общей физики

Содержание

1. Типы данных, операторы, выражения [см. лек. 1]

2. Управление (инструкции, блоки, циклы, переключатели) [см. лек. 2]

3. Массивы, структуры, указатели

4. Функции и структура программы, шаблоны функций

5. Интерфейс (ввод-вывод)

6. Работа с памятью, пространства имен

7. Классы и объекты

8. .....

Массивы, структуры и указатели

Массивы

Массив – это набор данных, относящихся к одному и тому же типу.

Объявление массива должно описывать три аспекта:

- тип значений каждого элемента;
- имя массива;
- количество элементов в массиве.

```
short months[12];          // создает массив из 12 элементов типа short
const int N=3;             // переменная с неизменяющимся значением
short months2[N],  months3[3*N];
```

```
имяТипа имяМассива[размерМассива];
```

размерМассива, т.е. количество элементов, должно быть целочисленной константой, значением const либо константным выражением (например, 5+7).

months[0] - это первый элемент массива months

months[11] - его последний элемент.

```
int m=3;
double arr[m]; // запрещено стандартом, так как m – переменная,
               // но может поддерживаться компилятором g++. Компилирование
               // с опцией «-pedantic-errors» дает
               // «error: ISO C++ forbids variable length array»
               // C99 (gcc компилятор) позволяет декларирование массивов
               // переменной длины, но не их инициализацию.
```

Инициализация массивов

Инициализация осуществляется только при объявлении массива.
Нельзя присваивать один массив другому:

```
int cards[4] = {3, 6, 8, 10}; // все в порядке
int hand[4];                  // все в порядке
hand[4] = {5, 6, 7, 9};       // не допускается
hand = cards;                 // не допускается
```

Можно указывать меньше значений, чем в массиве объявлено элементов:

```
float hotelTips[5] = {5.0, 2.5}; // оставшиеся будут 0
long totals[500] = {0};  // все элементы 0
long totals[500]={1};    // totals[0]=1, все другие - 0
short things [] = {1, 5, 3, 8}; // массив из 4 элементов
int N = sizeof(things)/sizeof(short); // количество элементов в массиве
```

Массивы в C++11

Можно отбросить знак =

```
double earnings[3] {1.2e4, 1.6e4, 1.1e4}; // допускается в C++11
```

Можно использовать пустые фигурные скобки:

```
unsigned int counts[10] = {}; // все элементы = 0
float balances[100] {};           // все элементы = 0
```

Защита от сужения (например, от double к int) при инициализации (ошибка при компиляции):

```
long plifs[ ] = {25, 92, 3.0};       // не разрешено в C++, работает в C
char slifs[4] {'h', 'i', 1122011, '\0'};   // не разрешено
char tlifs[4] {'h', 'i', 112, '\0'};          // разрешено
```

Строки в C и в C++

Строка в C – это последовательность символов, сохраненная в расположенных последовательно байтах памяти и оканчивающаяся нулевым символом '\0'.

Многие функции используют этот символ.

Функции из <cstring> (string.h): strlen, strcmp, strchr, strcpy, strcat и т.д.

В C++ вводится также класс string (строка). Объекты этого класса наделены определенными свойствами (например, =, +, += и т.д.)

```
char dog[8] = { 'b', 'e', 'a', 'u', 'x', ' ', 'I', 'I'};  // это не строка, а просто массив
char cat[8] = {'f', 'a', 't', 'e', 's', 's', 'a', '\0'};   // а это – строка
```

Более простой способ инициализации массива – строковой константой (строковым литералом):

```
char mystr[100] = "Some text"; // добавляет '\0' до конца массива
int s1=sizeof(mystr); // =100
int s2=strlen(mystr);  // = 9, без '\0'
char fish[ ] = "Bubbles";  // компилятор подсчитает количество элементов
char str[3]="abc"; // выдаст ошибку
```

Нужно обеспечить достаточный размер массива, чтобы в него поместились все символы строки, включая нулевой.

Конкатенация (присоединение) строковых постоянных (строковых литералов) (string constants, string literals)

Эквивалентные записи:

```
cout << "Hello, " "World!\n";

cout << "Hello, World!\n" ;

cout << "Hello, "
        "World!\n";
```

Чтение строк

cin – это объект входного потока.

Операция >> читает входной поток и записывает его в значения переменных. Результат операции >> зависит от типа переменной.

Здесь рассмотрим только тип char[].

char ch1[100];

cin >> ch1; // записывает до первого пробельного символа (пробел, табуляция,
// новая строка). Игнорирует начальные пробелы.
// Оставляет конечный пробельный символ в потоке ввода.

```
#include <iostream>

using namespace std;

#define N 100

int main(void){
  char s1[N], s2[N];

  cout << "Enter a string: ";

  cin >> s1;
  cin >> s2;

  cout << "s1=" << s1 << endl;
  cout << "s2=" << s2 << endl;
}
```

```
Enter a string: Hello, World!
s1=Hello,
s2=World!
```

Построчное чтение ввода в char *str

В классе istream, элементом которого является cin, есть функции-члены, предназначенные для строчно-ориентированного ввода:
getline(char *str, int n) и **get(char *str, int n)**.

Функция getline () читает целую строку (включая пробелы), используя символ новой строки '\n' (клавиша <Enter>) для обозначения конца ввода.

```
cin.getline(ch1, 20); // читает полную строку в массив ch1, предполагая,
                      // что строка состоит не  более чем из 19 символов.
                      // Добавляет '\0' к ch1 и убирает '\n' из потока ввода.
                      // Наличие более 19 символов в строке приводит к ошибке
```

Вместо того, чтобы прочитать и отбросить '\n', get() оставляет его на входе:

```
cin.get(s1, ArSize);  // чтение первой строки. Добавлает '\0' к s1.
                       // Оставляет неуместившиеся символы в потоке ввода.
cin.get ();            // чтение символа '\n' новой строки
cin.get(s2, Arsize);  // чтение второй строки
```

Конкатенация (объединение) операций:

```
cin.get(name, ArSize).get(); // конкатенация функций-элементов
cin.getline(name1, ArSize).getline(name2, ArSize); // сначала в name1
```

Комбинирование операций ввода

Комбинирование с cin:

Функция cin.get(char *, int n) прекращает ввод, если считана нулевая строка, то есть из потока не удалось считать символ. Если перед использованием cin.get(char *, int n) использовалась cin, то это приведет завершению программы.

Функция cin.getline(char *, int n) прекращает ввод, если было прочитано n-1 символов до обнаружения '\n'. Если перед использованием cin.getline(char *, int n) использовалась cin, то считывается пустая строка (и добавляется '\0') и убирается '\n' из потока ввода.

Перед использованием cin.get(char *, int n) или cin.getline(char *,int n) после cin нужно убрать символ '\n' из потока ввода, например, с помощью cin.get(). Если перед '\n' были другие пробельные символы, то их тоже надо удалить.

Класс string в C++

Класс string -- новый тип данных, имитирующий строки языка (письменного и разговорного). При письме мы не рассматриваем строки как массивы, как это делается в типе char[].

Класс string является частью пространства имён std.

```
std::string some_string="Hello!";
std::cout << "some_string=" << some_string << std::endl;
```

Пространство имён (namespace) - есть некоторый список, который имеет имя и состоит из имён объектов, функций, переменных.

```
#include <iostream>
#include <string>  // Обеспечение доступа к классу string
int main(void) {
  using namespace std;
  char charr1[20];              // создание пустого массива
  char charr2[20] = "jaguar"; // создание инициализированного массива
  string str1;                  // создание пустого объекта типа string
  string str2 = "panther";    // создание инициализированного объекта
  ........
```

Объект string обычно можно использовать так же, как символьный массив char[].

- Объект string можно инициализировать строкой в стиле C.
- Чтобы записать символы с клавиатуры в объект string, можно использовать cin.
- Для отображения объекта string можно применять cout.
- Можно использовать обозначения массивов, т.е. [], для доступа к индивидуальным символам, хранящимся в объекте string.

Главное отличие между объектами string и символьными массивами: объект string объявляется как обычная переменная, а не массив.

```
char c1[]="one";
string s1="two";
cout << strlen(c1); // длина строки c1;
cout << strlen(s1); // ошибка, так как s1 - не массив, а объект.
```

Чтение строк:

```
cin >> s1;            // чтение до пробельного
getline(cin, s1);     // чтение строки с пробелами
```

Присваивание, конкатенация и добавление

Объект string можно присвоить другому:

```
char charr1[20];                  // создание пустого массива
char charr2[20] = "jaguar"; // создание и инициализация

string str1; // создание пустого объекта string
string str2 = "panther"; // создание инициализированной строки

charr1 = charr2; // НЕПРАВИЛЬНО, присваивание массивов
                          // не разрешено

str1 = str2; // ПРАВИЛЬНО, присваивание объектов
                    // допускается
```

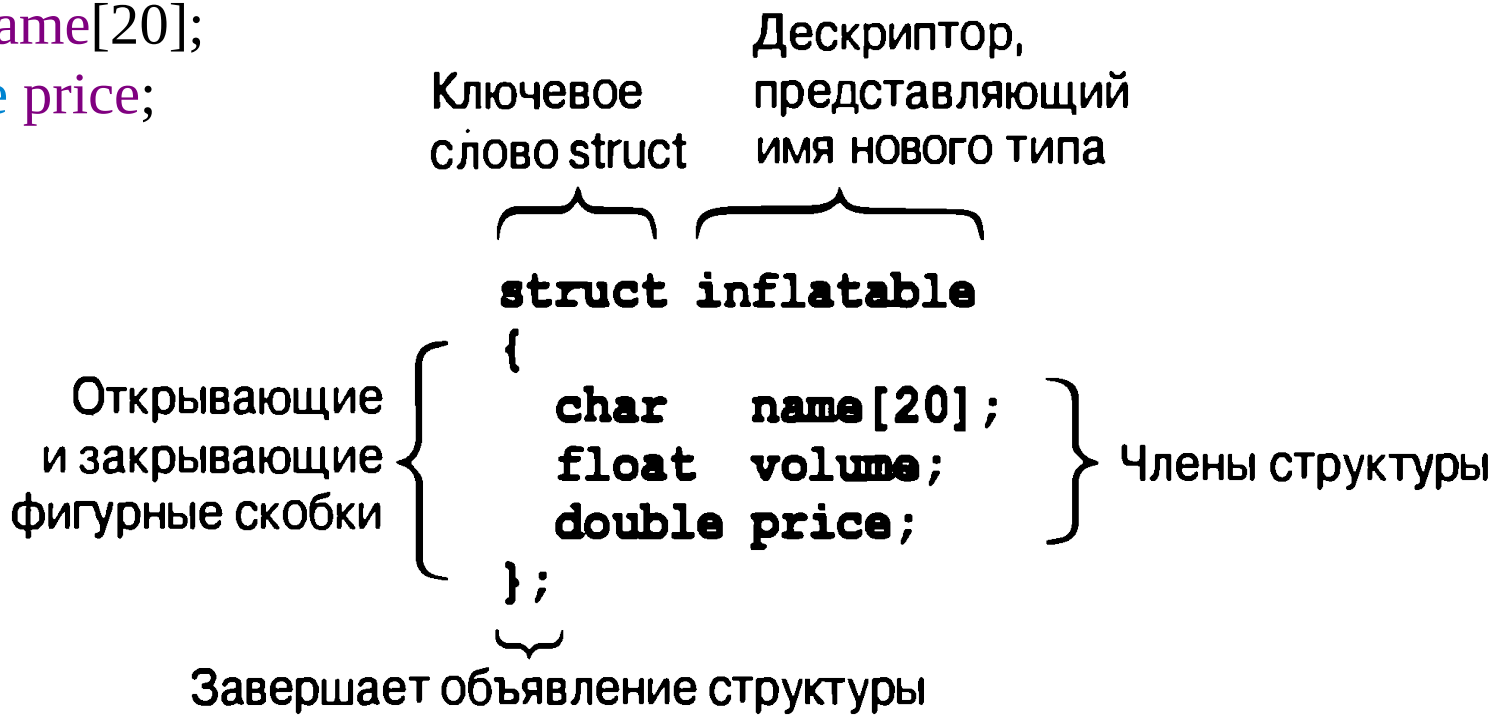
Упрощение комбинирования строк:

```
str3 = str1 + str2; // присвоить str3 объединение строк
str1 += str2;          // добавить str2 в конец strl
```

Структуры

Структура — тип данных, который может объединять элементы более чем одного типа. Дла работы с какой-либо структурой надо сначала ее описать, то есть перечислить из каких элементов она состоит.

```
// объявление структуры, ничего пока не создается
struct toy{
    char name[20];
    double price;
};
```



Работа со структурами

После определения структуры можно создавать переменные этого типа:

```
struct toy t1; // в C, t1 – структурная переменная типа toy
toy t1;        // t1 – структурная переменная типа toy
```

Переменная t1 имеет тип toy, для доступа к ее отдельным элементам (members, членам) используется операция (.)

t1.name – элемент структуры по имени name
t1.price – элемент по имени price.

Элемент t1.price является переменной типа double и может быть использован точно так же, как любая другая переменная типа double.

Короче говоря, toy является структурой, но toy.price относится к типу double.

Работа со структурами

```
struct toy{  // внешнее объявление структуры
      char name[20];
      double price;
};

int main (void) {
      .............
      toy t1 = { "Winnie-the-Pooh", 29.99}; // создание переменной, инициализация
      toy t2  { "Piglet", 9.99}; // The  = sign is optional
      toy t3 {} ; // все элементы =0

      struct toy t4; // keyword struct is optional in C++, but required in C.

       t2 = t1;                                  //  присваивание, работает начиная с C90
       memcpy(&t2, &t1, sizeof(toy));  // можно использовать функцию memcpy()
         .............
}
```

Работа со структурами

Комбинирование определения формы структуры с созданием структурных переменных.

```
struct perks {
      int key_number;
      char car[12];
} mr_smith, ms_jones; // две переменных типа perks
```

Одновременная инициализация:

```
struct perks {
      int key_number;
      char car[12];
} mr_glitz = { 7, "Packard" };
```

Массивы структур

```
struct toy{ // объявление структуры
      char name[20];
      double price;
};

toy gifts[100];                          // массив из 100 структур toy
cin >> gifts[0].price;                   // элемент price первой структуры
cout << gifts[99].price << endl; // элемент price последней структуры

// инициализация массива структур
toy  my_toys[2] = {{"Bambi",  21.99}, {"Godzilla", 65.99}};
```

Указатели (pointers)

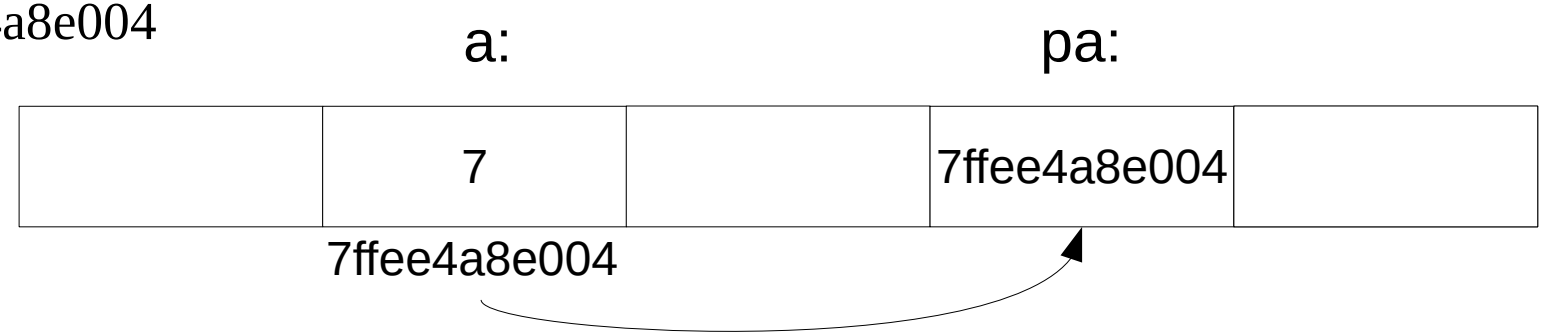
Указатели – переменные, значения которых – адреса памяти.

Операция & (ampersand, амперсанд) – взятие (получение) адреса

Операция * (asterisk, star, астериск, звёздочка) – получение/записывание значения (разыменование) по адресу.

```
int a = 7; // создание переменной
int *pa;  // создание указателя pa, значение которого — это адрес переменной
          // типа int.  Здесь * – определяет тип (указатель), а не дает значение
int *pa = &a; // создание и инициализация
pa=&a;       // получаем адрес переменной a
*pa = *pa + 1;  // изменяем значение переменной через указатель.
cout << "address of a= " <<  &a << endl; // вывод адреса на экран
cout << "value of pa= " <<  pa << endl; // вывод адреса на экран
cout << "value of *pa= " <<  *pa << endl;  // вывод значения, записанного по адресу
```

```
address of a= 0x7ffee4a8e004
value of pa= 0x7ffee4a8e004
value of *pa= 8
```



Объявление и инициализация указателей

Объявление указателя должно задавать тип данных указываемого значения.

```
int *p_updates;  // p_update указвыет на int
int* p1;         // так тоже можно
int * p4;        // так тоже можно
int *p2, *p3;    // объявление двух указателей
int *p5, b;      // объявление указателя и обычной переменной
int* p6, c;      // объявление указателя и обычной переменной
int d, *p7;      // объявление обычной переменной и указателя

int var1 = 5;
int * pt = &var1; // объявление и инициализация
```

Указатель должен иметь значение (заданное через инициализацию или присвоение), прежде чем к нему можно применять операцию разыменования (*, dereferencing). Значение указателя должно соответствовать используемому участку памяти.

```
long *fellow;        // создать указатель на long
*fellow = 223323;    // поместить значение в неизвестное место
```

segmentation fault (ошибка сегметации, адреса памяти)

Указатели и числа

Указатели – это не целочисленные типы

```
int *pt;
pt = 0xB8000000; // несоответствие типов
```

Надо делать приведение типа:

```
int *pt;
pt = (int *) 0xB8000000; // типы соответствуют, но так делать не надо
```

Размер указателя в памяти:

sizeof(int)=4

sizeof(long)=8

sizeof(int *)=8

sizeof(long *)=8

Указатели типа void

```
void *v; // указатель на неопределенный (произвольный) тип данных
```

Размер:

sizeof(void *) = 8

```
int a=87;
int *ip = &a;  // записали адрес переменной a
cout << *ip; // распечатает 87, то есть значение переменной a
void *vp = (void *) &a;  // адрес переменной a без спецификации типа
cout << *vp; // ошибка при компиляции
void *p = &p; // так тоже можно! Значение p равно ее адресу.
 cout << "p=" << p << endl; // распечатает адрес
 cout << "*(char * ) vp="<< *(char * ) vp << endl; // распечатает W (=87)
```

Выделение/освобождение памяти операциями new/delete

Указатели используют для работы с неименованными областями памяти, выделяемыми в процессе выполнения программы.

Память, выделяемая операцией new, находится в области, называемой кучей или свободным хранилищем (heap or free store).

В языке C++ используют new/delete:

```
int *ps = new int[10];  // выделить память, без инициализации
.............................      // использовать память
delete [] ps;                  // освободить память, указатель не удаляется.
```

Массив ps – динамический массив.

В языке C используют malloc/free (можно использовать и в C++):

```
int *ps  = malloc(10*sizeof(int));// выделить память, без инициализации
.............................                    // использовать память
free(ps);          // освободить память, сам указатель не удаляется,
                   // его значение не меняется.
```

Создание динамических массивов

Объявление массива внутри функции означает, что массив встраивается в программу во время компиляции (статическое связывание).

```
int some_array[20];
```

Динамическое связывание означает, что массив будет создан во время выполнения программы. Такой массив называется динамическим массивом. Операция delete должна получить значение, которое было возвращено операцией new.

```
int * psome = new int [10] ; // получение блока памяти из 10
                             // элементов типа int
char *pc = new char; // присваивание адреса выделенной памяти char
                     // указателю pc
….
delete [] psome; // освобождение динамического массива
delete pc;       // освобождение памяти
```

Имена массивов и указатели

C/C++ интерпретирует имя массива как адрес первого элемента.

```
double wages[3]={1000.0, 2000.0, 3000.0};
double *pw = wages;   // создаем указатель  на double и
                      // инициализируем его
pw++;                 // можно, так как указатель --  переменная
wages++;              // нельзя, так как имя массива -- не переменная
sizeof(pw)      дает 8, то есть размер указателя
sizeof(wages)  дает 24, то есть размер массива
```

Для wages, как и любого другого массива, выполняется:

wages эквивалентно &wages[0] // адрес первого элемента массива

pw эквивалентно &pw[0];

Имена массивов и указатели

C/C++ выполняет следующие преобразования при компиляции:

имя_массива[i] превращается в * (имя_массива + i)

имя_указателя[i] превращается в *(имя_указателя + i)

Т.е. имена указателей и имена массивов можно использовать одинаковым образом. Нотация квадратных скобок [] применима и там, и там. К обоим можно применять операцию разыменования (dereferencing operator, *).

```
int a[3]={5,6,7};
int *p=a;
*p=4;    // changes a[0] to 4
p[1]=8; // changes a[1] to 8
```

В большинстве выражений каждое имя представляет адрес. Единственное отличие состоит в том, что значение указателя изменить можно, а имя массива является константой:

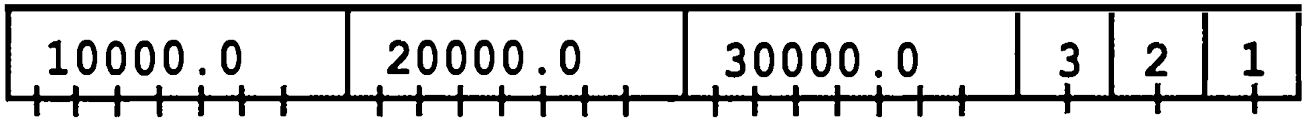
имя_указателя = имя_указателя + 1; // правильно

имя массива = имя массива + 1; // не допускается

Арифметика указателей

Тип указателя определяет, на сколько байт надо сместиться в памяти для сдвига на один элемент (изменения указателя на 1).

```
double wages[3] = {10000.0, 20000.0, 30000.0};
short stacks[3] = {3, 2, 1};
double * pw = wages;
short * ps = &stacks[0];
```



Адрес: 100 108 116 124 126 128

pw (**pw + 1**) **ps** (**ps + 1**)

pw указывает на тип **double**, поэтому добавление к нему 1 изменяет его величину на 8 байт

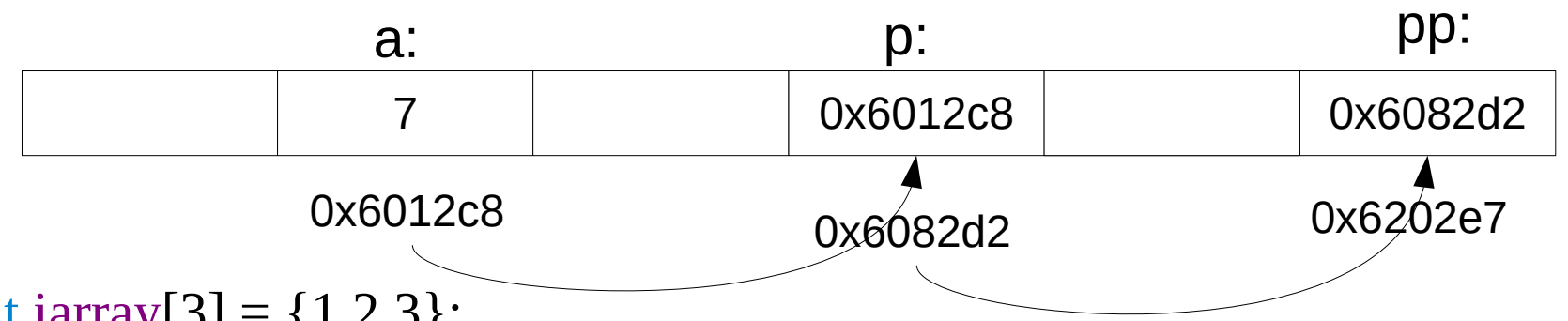
ps указывает на тип **short**, поэтому добавление к нему 1 изменяет его величину на 2 байта

sizeof(double) -> 8
sizeof(short) -> 2

Указатели на указатели

Указатель на указатель – переменная, значение которой содержит адрес памяти, по которому записан указатель.

```
int     a=7;
int   *p=&a;   // указатель на int
int **pp=&p; // указатель на указатель int *
```

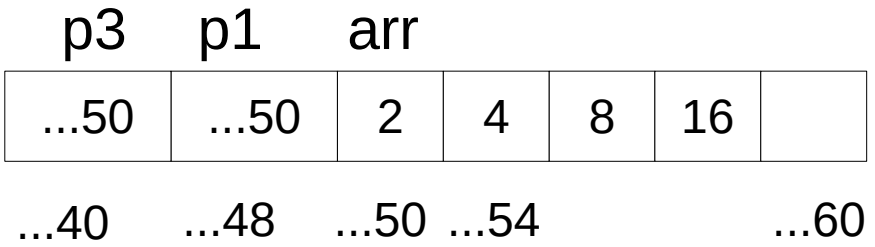



```
int iarray[3] = {1,2,3};
int *ip = iarray;
int **ipp = &ip;

// вывести второй элемент массива
 printf("a[1]=%d\n", iarray[1]);
 printf("a[1]=%d\n", ip[1]);      // *(ip+1)
 printf("a[1]=%d\n", *(ip+1));
 printf("a[1]=%d\n", 1[ip]);      // тоже работает! *(ip+1)
 printf("a[1]=%d\n", (*ipp)[1]);
```

Пример:

```
int arr[4]={2, 4, 8, 16};
int *p1, *p2;
int (*p3)[4]; // pointer to an array of 4 integers

p1=arr;   // pointer to arr[0]
// p2=&arr; так нельзя
p3=&arr;  // pointer to the whole array

printf("         arr[0]=%d\n\n",   arr[0]);
printf("   sizeof(int)=%d bytes\n", sizeof(int));
printf("   sizeof(arr)=%d bytes\n", sizeof(arr));
printf(" sizeof(int *)=%d bytes\n\n", sizeof(int *));

printf("    &arr[0]=%p\n", &arr[0]);
printf("       arr=%p\n", arr);
printf("        p1=%p\n",  p1);
printf("       &p1=%p\n", &p1);
printf("        p3=%p\n",  p3);
printf("       &p3=%p\n\n", &p3);

p3++; p1++;
// arr++; а так нельзя
printf("        p1=%p\n",  p1);
printf("        p3=%p\n",  p3);
```

```
         arr[0]=2

  sizeof(int)=4 bytes
  sizeof(arr)=16 bytes
sizeof(int *)=8 bytes

    &arr[0]=0x7ffe02880550
       arr=0x7ffe02880550
        p1=0x7ffe02880550
       &p1=0x7ffe02880548
        p3=0x7ffe02880550
       &p3=0x7ffe02880540

        p1=0x7ffe02880554
        p3=0x7ffe02880560
```

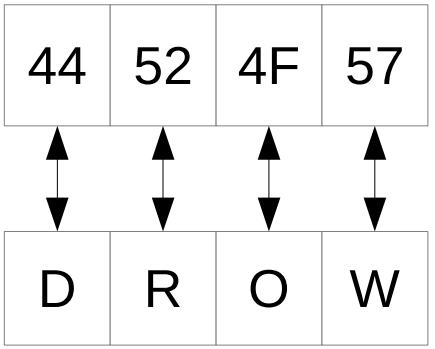


Представление чисел в памяти

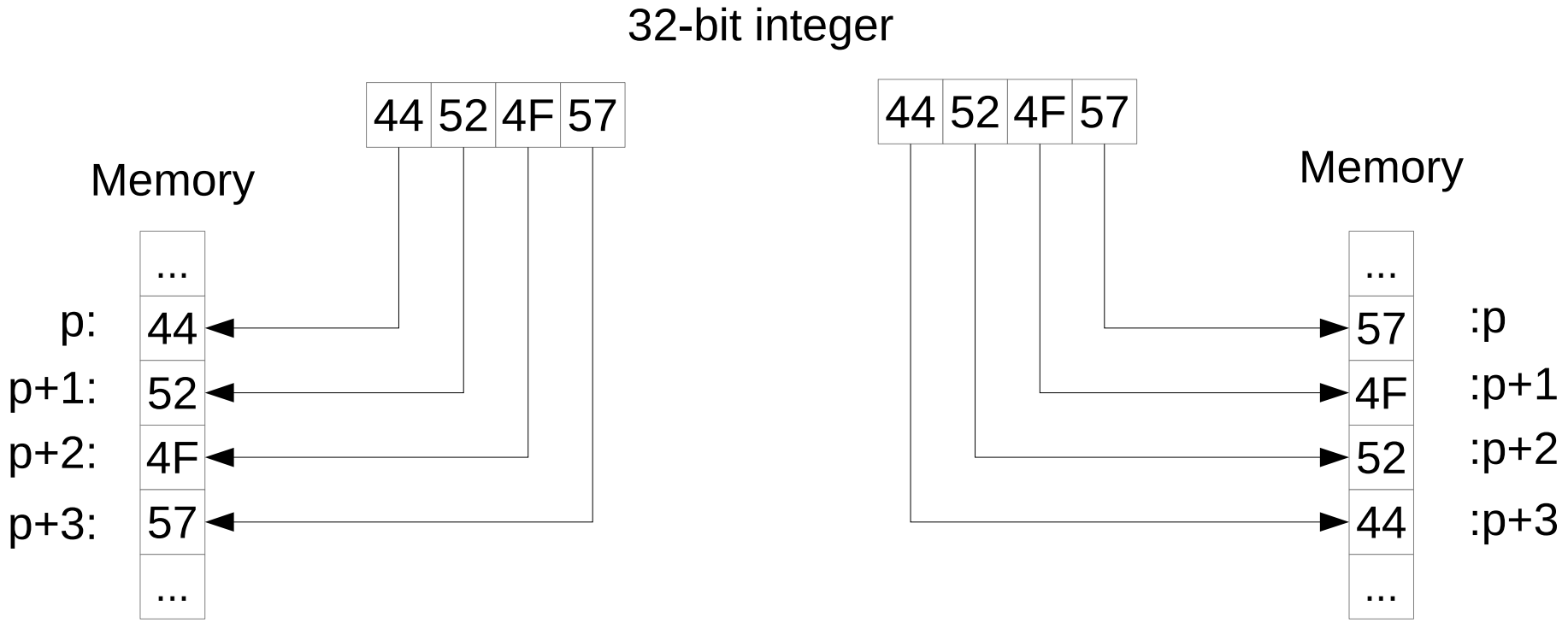
```
int main(void){

   unsigned int var = 0b0100'0100'0101'0010'0100'1111'0101'0111;
   // unsigned int var = 1'146'244'951;
   // unsigned int var = 0x44524F57;

   cout << hex << "hex: var = " << var << endl;
   cout << dec << "dec: var = " << var << endl;

   char *cp = (char*) &var;

   printf("cp[0]=%c\n", cp[0]);
   printf("cp[1]=%c\n", cp[1]);
   printf("cp[2]=%c\n", cp[2]);
   printf("cp[3]=%c\n", cp[3]);

   return 0;
}
```

```
hex: var = 44524f57
dec: var = 1146244951
cp[0]=W
cp[1]=O
cp[2]=R
cp[3]=D
```

ASCII:



Представление чисел в памяти (порядок следования байт или endianness)



Big-endian (BE):
Самый значимый по значению байт сохраняется в памяти первым по порядку

Это сетевой порядок байт.

Little-endian (LE)
Самый малый по значению байт сохраняется в памяти первым по порядку

Указатели и строки

```
char flower[10] = "rose";
cout << flower << "s are red\n"; // печатает roses are red
```

Имя массива – это адрес его первого элемента, т.е. flower в операторе cout представляет адрес элемента char, содержащего символ 'r'.

Объект cout предполагает, что адрес char – это адрес строки, поэтому печатает символ, расположенный по этому адресу, и затем продолжает печать последующих символов, пока не встретит нулевой символ '\0'.

Строка в кавычках, как и имя массива, служит адресом первого элемента.

```
int a[3]={1,2,3};
cout << a; // печатает адрес массива
```

Указатели и строки (примеры)

```
char flower[10] = "rose";            // создали строку
char *p=flower;                      // создали указатель
cout << flower << "s are red\n"; // печатает roses are red
cout << p << "s are red\n";          // печатает roses are red
cout << *p << '\n';                  // печатает r
cout << &p << '\n';                  // печатает адрес указателя
cout << &flower << '\n';             // печатает адрес массива flower
cout << (void *) flower << '\n';   // печатает адрес массива flower
cout << (void *) p << '\n';          // печатает значение p = адрес массива flower
```

```
char x='a';
char *p=&x;
cout << "x=" << p << endl;  // печатает мусор, так как рассматривает p как строку
cout << "x=" << *p << endl; // печатает a
cout <<  "p=" << (void *) p;  // печатает значение указателя = адрес x
```

Указатели и операции ++, --

```
int arr[5] = {2, 4, 8, 16, 32};
int *pt = arr;   // pt указывает на arr[0], т.е. на 2
++pt;            // pt указывает на arr[1], т.е. на 4
```

Префиксный инкремент, префиксный декремент и операция разыменования имеют одинаковый приоритет и действуют справа налево.

```
int b = *++pt; // изменить указатель и затем получить значение;
               // т.е. агг[2] , или 8
++*pt;       // инкремент указываемого значения, т.е. изменение 8 на 9,
             // без изменения указателя, pt указывает на arr[2]
```

Постфиксный инкремент и декремент имеют приоритет, более высокий, чем приоритет префиксных форм.

```
(*pt)++;     // инкремент указываемого значения, т.е. изменение 9 на 10
b = *pt++;  // разыменование исходного указателя, т.е. 10, затем инкремент
            // указателя  (++ действует на pt, а не на *pt); тоже что и b=*(pt++);
            // после операции pt указывает на arr[3], т. е. на 16
```

Двумерные массивы

Двумерный массив – массив, каждый элемент которого является массивом.

```
int maxtemps[4][5]; // массив из 4-х элементов, каждый из
                    // которых является массивом из 5-и элементов
```

Массив maxterms, представленный в виде таблицы

		0	1	2	3	4
maxtemps[0]	0	maxtemps[0][0]	maxtemps[0][1]	maxtemps[0][2]	maxtemps[0][3]	maxtemps[0][4]
maxtemps[1]	1	maxtemps[1][0]	maxtemps[1][1]	maxtemps[1][2]	maxtemps[1][3]	maxtemps[1][4]
maxtemps[2]	2	maxtemps[2][0]	maxtemps[2][1]	maxtemps[2][2]	maxtemps[2][3]	maxtemps[2][4]
maxtemps[3]	3	maxtemps[3][0]	maxtemps[3][1]	maxtemps[3][2]	maxtemps[3][3]	maxtemps[3][4]

```
int maxtemps[4][5] = {// двумерный массив
      {96, 100, 87, 101, 105},  // значения для maxtemps[0 ]
      {96,   98, 91, 107, 104},  // значения для maxtemps[1]
      {97, 101, 93, 108, 107},  // значения для maxtemps[2]
      {98, 103, 95, 109, 108}   // значения для maxtemps [3]
};

for (int row = 0; row < 4; row++ ) {
      for (int col = 0; col < 5; ++col)
            cout << maxtemps [row] [col] << "\t";
      cout << endl;
}
```

Создание динамических структур, указатели на структуры

```
struct toy{ // объявление структуры
   char name[20];
   double price;
};
toy *ps = new toy; // создание безымянной структуры типа toy
```

У нас нет доступа к ее элементам через имя, поскольку эта структура безымянна. Для доступа к элементам используется адрес и операция ->

ps->price означает элемент (член) price структуры, на которую указывает ps.

Синтаксически более сложный подход: (*ps).price

```
strcpy(ps->name, "Mickey Mouse"); // нельзя ps->name="Mickey Mouse";
ps->price= 12.99;
toy t1={"Donald Duck", 10.99};
toy *ps2=&t1; // указатель на структуру t1
double x = ps2->price;   // получение значения элемента price
double y = (*ps2).price; // тоже
double z = t1.price;       // тоже
```

Хранение, диапазон доступа и связывание данных

Как правило, программы состоят из большего количества функций и различных типов данных разбитых по файлам.

Функции имеют глобальный диапазон (область видимости) , то есть функции из одного файла могут вызываться функциями из другого файла. Если перед функцией стоит ключевое слово static, то она видна только в этом файле, то есть может вызываться функциями из того же файла.

Переменные делятся:

1) По типу хранения (для процессоров с одним ядром):
автоматические, статические, динамические

2) По диапазону доступа (видимости):
локальные (видны только в блоке, где определены) и глобальные (определены вне функций и видны везде).

3) По связыванию:
Глобальные переменные бывают со внешним связыванием (могут использоваться функциями в различных файлах) и внутренним связыванием (видны только внутри файла, определены со словом static)

Способы хранения данных в памяти

1) Автоматическое хранилище [automatic storage]: Обычные переменные, объявленные внутри функции, используют автоматическое хранение и называются автоматическими переменными. Они создаются автоматически при вызове содержащей их функции и уничтожаются при ее завершении. Автоматические переменные также создаются внутри блока, то есть части кода, ограниченной {}. Такие переменные обычно хранятся в стеке (stack).

2) Статическое хранилище [static storage] – это хранилище, которое существует в течение всего времени выполнения программы, то есть память выделяется в начале работы программы и освобождается при ее завершении. Два способа сделать переменные статическими: (а) объявление их вне функций, (б) использование при объявлении переменной внутри функции ключевого слова static:

```
static double fee = 56.50;
```

3) Динамическое хранилище (свободное хранилище или куча) [dynamic storage (free store or heap)] -- область памяти, отделенная от статической и динамической, которая управляется с помощью операций new (выделение памяти) и delete (освобождение). Время жизни данных при этом не привязывается жестко к времени жизни программы или функции. Вместо пары new и delete можно использовать функции malloc и free.

4) Хранилище потока (thread storage) (начиная с C++11) – используется для процессоров с несколькими ядрами.