Массивы
Строки
Пользовательские типы

При использовании простых переменных каждой области памяти для хранения данных соответствует свое имя. Если с группой величин одинакового типа требуется выполнять однообразные действия, им дают одно имя, а различают по порядковому номеру. Это позволяет компактно записывать множество операций с помощью циклов.
Конечная именованная последовательность однотипных величин называется *массивом*.

Описание массива в программе отличается от описания простой переменной наличием после имени квадратных скобок, в которых задается количество элементов массива (размерность):

```
float a [10]; // описание массива из 10 вещественных чисел
```

Элементы массива нумеруются с нуля. При описании массива используются те же модификаторы (класс памяти, const и инициализатор), что и для простых переменных. Инициализирующие значения для массивов записываются в фигурных скобках. Значения элементам присваиваются по порядку. Если элементов в массиве больше, чем инициализаторов, элементы, для которых значения не указаны, обнуляются:

```
int b[5] = {3, 2, 1}; // b[0]=3, b[1]=2, b[2]=l, b[3]=0, b[4]=0
```

Размерность массива вместе с типом его элементов определяет объем памяти, необходимый для размещения массива, которое выполняется на этапе компиляции, поэтому размерность может быть задана только целой положительной константой или константным выражением. Если при описании массива не указана размерность, должен присутствовать инициализатор, в этом случае компилятор выделит память по количеству инициализирующих значений.

В дальнейшем будет показано, что размерность может быть опущена также в списке формальных параметров.

Для доступа к элементу массива после его имени указывается номер элемента (индекс) в квадратных скобках.

Пример: сумма *элементов массива*.

```
#include <iostream.h>
int main(){
const int n = 10;
int i, sum;
int marks[n] = {3, 4, 5, 4, 4};
for (i = 0, sum = 0; i<n; i++) sum += marks[i];
cout « "Сумма элементов; " « sum;
return 0; }
```

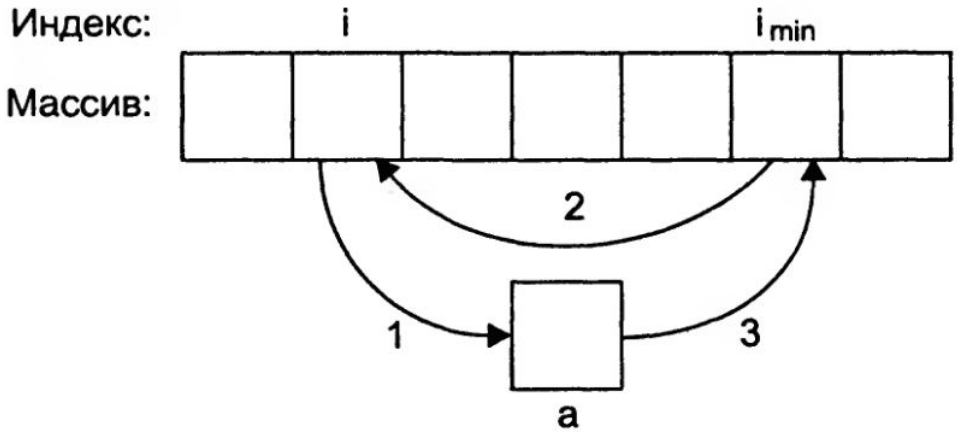
Размерность массивов предпочтительнее задавать с помощью именованных констант, как это сделано в примере, поскольку при таком подходе для ее изменения достаточно скорректировать значение константы всего лишь в одном месте программы. (! Последний элемент массива имеет номер, на единицу меньший заданной при его описании размерности.)

Пример. *Сортировка целочисленного массива методом выбора*

Алгоритм: выбирается наименьший элемент массива и меняется местами с первым элементом, затем рассматриваются элементы, начиная со второго, и наименьший из них меняется местами со вторым элементом, и так далее n-1 раз (при последнем проходе цикла при необходимости меняются местами предпоследний и последний элементы массива).

```
#include <iostream.h>
int main(){ const int n = 20; // количество элементов массива
int b[n]; int i; // описание массива
for (i = 0; i<n; i++) cin » b[i]; // ввод массива
for (i = 0; i<n-1; i++){ // n-1 раз ищем наименьший элемент
// принимаем за наименьший первый из рассматриваемых элементов
int imin = i;
// поиск номера минимального элемента из неупорядоченных:
for (int j = i + 1; j<n; j++) // если нашли меньший элемент, запоминаем его номер:
if (b[j] < b[imin]) imin = j;
int a = b[i]; b[i] = b[imin]; // обмен элементов с номерами
b[imin] = a; // i и imin
}
// вывод упорядоченного массива:
for (i =0: i<n; i++)cout « b[i] « ' ';
return 0; }
```

Процесс обмена элементов массива с номерами i и imin через буферную переменную a на i-м проходе цикла проиллюстрирован на рис. Цифры около стрелок обозначают порядок действий.



Идентификатор массива является константным указателем на его нулевой элемент.

Например, для массива из предыдущего листинга имя b — это то же самое, что &b[0], а к i-му элементу массива можно обратиться, используя выражение *(b+i). Можно описать указатель, присвоить ему адрес начала массива и работать с массивом через указатель. Следующий фрагмент программы копирует все элементы массива a в массив b:

```
int a[100],b[100];
int *pa = a;  // или int *p = &a[0];
int *pb = b;
for(int i = 0; i<100; i++)
   *pb++ = *pa++; // или pb[i] = pa[i];
```

Динамические массивы создают с помощью операции new, при этом необходимо указать тип и размерность, например:

```
int n = 100;
float *p = new float [n];
```

В этой строке создается переменная-указатель на float, в динамической памяти отводится непрерывная область, достаточная для размещения 100 элементов вещественного типа, и адрес ее начала записывается в указатель p. Динамические массивы нельзя при создании инициализировать, и они не обнуляются.

Преимущество динамических массивов состоит в том, что размерность может быть переменной, то есть объем памяти, выделяемой под массив, определяется на этапе выполнения программы. Доступ к элементам динамического массива осуществляется точно так же, как к статическим, например, к элементу номер 5 приведенного выше массива можно обратиться как p[5] или *(p+5).

Альтернативный способ создания динамического массива — использование функции malloc библиотеки C:

```
int n = 100;
float *q = (float *) malloc(n * sizeof(float));
```

Операция преобразования типа, записанная перед обращением к функции malloc, требуется потому, что функция возвращает значение указателя типа void*, а инициализируется указатель на float.

Память, зарезервированная под динамический массив с помощью new [], должна освобождаться оператором delete [], а память, выделенная функцией malloc — посредством функции free, например:

```
delete [ ] p; free (q);
```

При несоответствии способов выделения и освобождения памяти результат не определен. Размерность массива в операции delete не указывается, но квадратные скобки обязательны.

Многомерные массивы задаются указанием каждого измерения в квадратных скобках, например, оператор

int matr [6][8];

задает описание двумерного массива из 6 строк и 8 столбцов. В памяти такой массив располагается в последовательных ячейках построчно. Многомерные массивы размещаются так, что при переходе к следующему элементу быстрее всего изменяется последний индекс. Для доступа к элементу многомерного массива указываются все его индексы, например, matr[i][j], или более экзотическим способом: *(matr[i]+j) или *(*(matr+i)+j).
Это возможно, поскольку matr[i] является адресом начала i-й строки массива. При инициализации многомерного массива он представляется либо как массив из массивов, при этом каждый массив заключается в свои фигурные скобки (в этом случае левую размерность при описании можно не указывать), либо задается общий список элементов в том порядке, в котором элементы располагаются в памяти:

```
int mass2 [][2] = { {1, 1}, {0, 2}, {1, 0} };
int mass2 [3][2] = {1, 1, 0, 2, 1, 0};
```

Пример. *В целочисленной матрице найти номер строки, которая содержит наибольшее количество элементов, равных нулю*

```
#include <stdio.h>
int main(){ const int nstr = 4, nstb = 5; // размерности массива
 int i, j; int b[nstr][nstb];  // описание массива
for (i =0; i<nstr; i++) // ввод массива
for (j = 0; j<nstb; j++) scanf(“%d“, &b[i][j]);
int istr = -1, MaxKol = 0;
for (i = 0; i<nstr: i++){ // просмотр массива по строкам
int Kol = 0:
for (j = 0; j<nstb; j++) if (b[i][j] == 0)Kol++;
if (Kol > MaxKol){istr = i; MaxKol = Kol;}
}
printf(“ Исходный массив :\n");
for (i =0; i<nstr; i++){
for (j = 0; j<nstb; j++)printf(“%d “, b[i][j]);
printf("\n");}
if (istr == -1)printf("Нулевых элементов нет");
else printf("Номер строки; %d“, istr);
return 0;}
```

Номер искомой строки хранится в переменной istr, количество нулевых элементов в текущей (i-й) строке — в переменной Kol, максимальное количество нулевых элементов — в переменной MaxKol.

Массив просматривается по строкам, в каждой из них подсчитывается количество нулевых элементов

(переменная Kol обнуляется перед просмотром каждой строки). Наибольшее количество и номер соответствующей строки запоминаются.

Для создания динамического многомерного массива необходимо указать в операции new все его размерности (самая левая размерность может быть переменной), например:

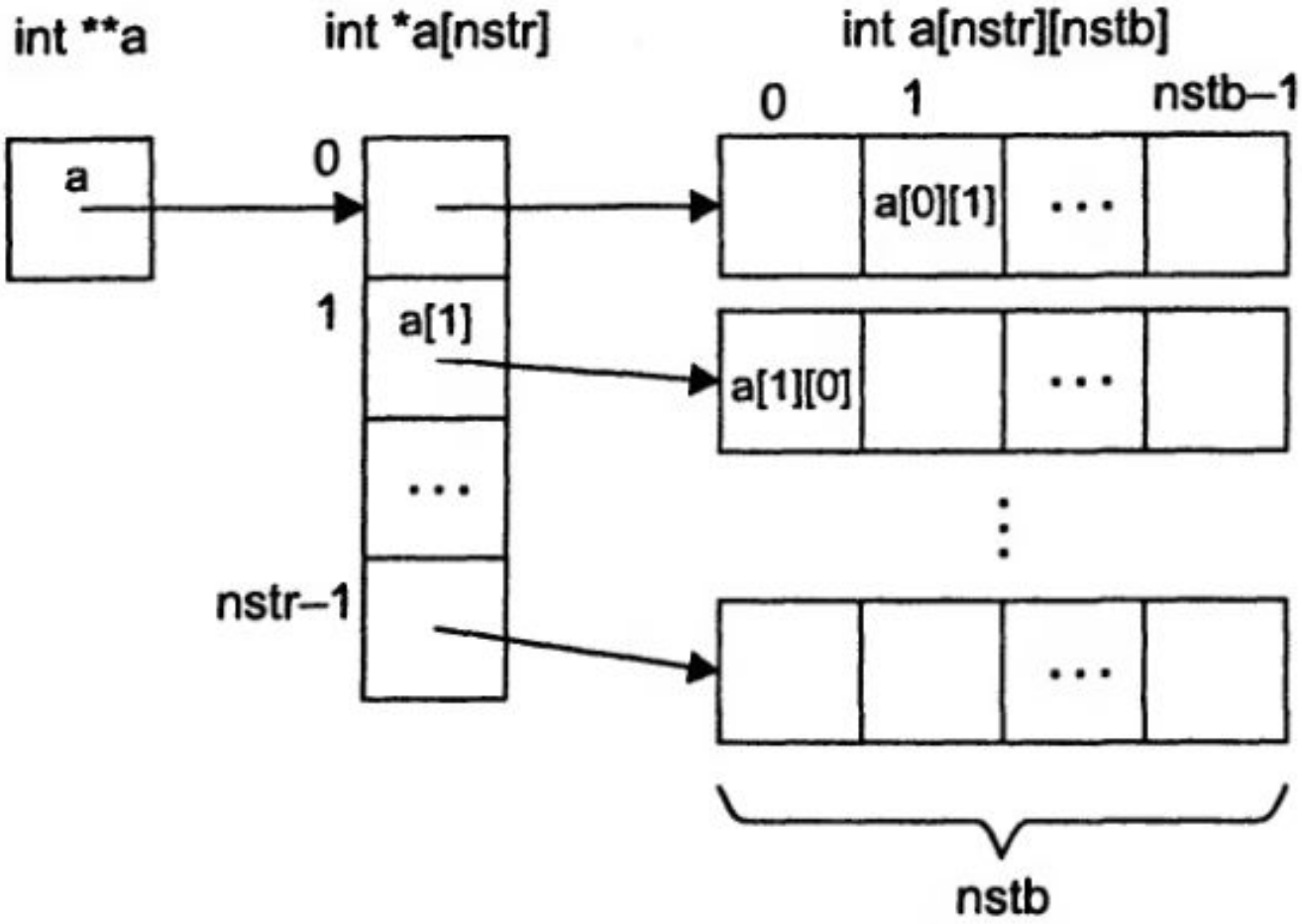
```
int nstr = 5; int ** m = (int **) new int [nstr][10];
```

Более универсальный и безопасный способ выделения памяти под двумерный массив, когда обе его размерности задаются на этапе выполнения программы:

```
int nstr, nstb;
cout « " Введите количество строк и столбцов :";
cin » nstr » nstb;
int **a = new int *[nstr]; // 1
for(int i = 0; i<nstr; i++) // 2
a[i] = new int [nstb]; // 3
…
```

В операторе 1 объявляется переменная типа «указатель на указатель на int» и выделяется память под массив указателей на строки массива (количество строк — nstr). В операторе 2 организуется цикл для выделения памяти под каждую строку массива. В операторе 3 каждому элементу массива указателей на строки присваивается адрес начала участка памяти, выделенного под строку двумерного массива.

Каждая строка состоит из nstb элементов типа int



Освобождение памяти из-под массива с любым количеством измерений выполняется с помощью операции delete [ ]. Указатель на константу удалить нельзя.

СТРОКИ

Строка представляет собой массив символов, заканчивающийся нуль-символом. Нуль-символ — это символ с кодом, равным 0, что записывается в виде управляющей последовательности '\0'. По положению нуль-символа определяется фактическая длина строки. Строку можно инициализировать строковым литералом

```
char str[10] = "Vasia";
// выделено 10 элементов с номерами от 0 до 9
// первые элементы - 'V', 'a', 's', 'i', 'a', '\0'
```

В этом примере под строку выделяется 10 байт, 5 из которых занято под символы строки, а шестой — под нуль-символ. Если строка при определении инициализируется, её размерность можно опускать (компилятор сам выделит соответствующее количество байт):

```
char str[] = "Vasia"; // выделено и заполнено 6 байт
```

Оператор char *str = "Vasia" создает не строковую переменную, а указатель на строковую константу, изменить которую невозможно

(к примеру, оператор str[1] ='o' не допускается).

Знак равенства перед строковым литералом означает инициализацию, а не присваивание. Операция присваивания одной строки другой не определена (поскольку строка является массивом) и может выполняться с помощью цикла или функций стандартной библиотеки. Библиотека предоставляет возможности копирования, сравнения, объединения строк, поиска подстроки, определения длины строки и т. д., а также содержит специальные функции ввода строк и отдельных символов с клавиатуры и из файла.

Заголовочный файл <string.h> (<cstring>) — функции работы со строками в стиле C

memchr	Ищет первое вхождение символа в блок памяти
memcmp	Сравнивает блоки памяти
memcpy	Копирует блок памяти
memmove	Переносит блок памяти
memset	Заполняет блок памяти символом
strcat	Складывает строки
strchr	Ищет символ в строке
strcmp	Сравнивает строки
strcoll	Сравнивает строки с учетом установленной локализации
strcpy	Копирует одну строку в другую
strcspn	Ищет один из символов одной строки в другой
strerror	Возвращает указатель на строку с описанием ошибки
strlen	Возвращает длину строки
strncat	Складывает одну строку с n символами другой

strncmp	Сравнивает одну строку с n символами другой
strncpy	Копирует первые n символов одной строки в другую
strpbrk	Ищет один из символов одной строки в другой
strrchr	Ищет символ в строке
strspn	Ищет символ одной строки, отсутствующий в другой
strstr	Ищет подстроку в строке
strtok	Выделяет из строки лексемы
strxfrm	Преобразует строки на основе текущей локализации
wcscat	Складывает строки
wcschr	Ищет символ в строке
wcscmp	Сравнивает строки
wcscoll	Сравнивает строки с учетом установленной локализации

wcscpy	Копирует одну строку в другую
wcscspn	Ищет один из символов одной строки в другой
wcslen	Возвращает длину строки
wcsncat	Складывает одну строку с n символами другой
wcsncmp	Сравнивает одну строку с n символами другой
wcsncpy	Копирует первые n символов одной строки в другую
wcspbrk	Ищет один из символов одной строки в другой
wcsrchr	Ищет символ в строке
wcsspn	Ищет символ одной строки, отсутствующий в другой
wcsstr	Ищет подстроку в строке
wcstok	Выделяет из строки лексемы
wcstrxfrm	Преобразует строки на основе текущей локализации
wmemcpy	Копирует блок памяти
wmemmove	Переносит блок памяти
wmemset	Заполняет блок памяти символом

Пример. Программа запрашивает пароль не более трех раз.

```
#include <stdio.h>
#include <string.h>
int main(){
char s[80], passw[] = "kuku"; // passw - эталонный пароль.
                              // Можно описать как *passw = "kuku";
 int i, k = 0;
 for (i = 0; !k && i<3; i++){ printf("\nвведите пароль:\n");
              gets(s); // функция ввода строки
           if (strstr(s,passw))k = 1; // функция сравнения строк
}
 if (k) printf("\nпароль принят");
else printf(“n\пароль не принят");
return 0; }
```

При работе со строками часто используются указатели.

Процесс копирования строки src в строку dest.
Очевидный алгоритм имеет вид:

```
char src[10], dest[10];
 for (int i = 0; i<=strlen(src); i++) dest[i] = src[i];
```

Длина строки определяется с помощью функции strlen, которая вычисляет длину, выполняя поиск нуль-символа. Таким образом, строка фактически просматривается дважды. Более эффективным будет использовать проверку на нуль-символ непосредственно в программе.

Увеличение индекса можно заменить инкрементом указателей (для этого память под строку src должна выделяться динамически, а также требуется определить дополнительный указатель и инициализировать его адресом начала строки dest):

```
#include <iostream.h>
int main(){
char *src = new char [10];
char *dest = new char [10], *d = dest;
cin » src;
while ( *src != 0) *d++ = *src++;
*d = 0; // завершающий нуль
cout « dest;
return 0; }
```

В цикле производится посимвольное присваивание элементов строк с одновременной инкрементацией указателей. Результат операции присваивания — передаваемое значение, которое, собственно, и проверяется в условии цикла, поэтому можно поставить присваивание на место условия, а проверку на неравенство нулю опустить (при этом завершающий нуль копируется в цикле, и отдельного оператора для его присваивания не требуется). В результате цикл копирования строки принимает вид: `while ( *d++ = *src++);`

Оба способа работы со строками (через массивы или указатели) приемлемы и имеют свои плюсы и минусы, но в общем случае лучше пользоваться функциями библиотеки или определенным в стандартной библиотеке C++ классом string, который обеспечивает индексацию, присваивание, сравнение, добавление, объединение строк и поиск подстрок, а также преобразование из C-строк, то есть массивов типа char, в string, и наоборот.

Типы данных, определяемые пользователем

В реальных задачах информация, которую требуется обрабатывать, может иметь достаточно сложную структуру. Для ее адекватного представления используются типы данных, построенные на основе простых типов данных, массивов и указателей.

Язык C++ позволяет программисту определять свои типы данных и правила работы с ними. Исторически для таких типов сложилось наименование, вынесенное в название главы, хотя правильнее было бы назвать их типами, определяемыми программистом.

Переименование типов (typedef)

Для того чтобы сделать программу более ясной, можно задать типу новое имя с помощью ключевого слова typedef:

```
typedef тип новое_имя [ размерность ];
```

В данном случае квадратные скобки являются элементом синтаксиса. Размерность может отсутствовать. Примеры:

```
typedef unsigned int UINT;
typedef char Msg[100];
typedef struct{ char fio[30];
int date, code;
double salary;} Worker;
```

Введенное таким образом имя можно использовать таким же образом, как и имена стандартных типов:

```
UINT i, j; // две переменных типа unsigned
int Msg str[10]; // массив из 10 строк по 100 символов
Worker staff[100]; // массив из 100 структур
```

Кроме задания типам с длинными описаниями более коротких псевдонимов, typedef используется для облегчения переносимости программ: если машинно-зависимые типы объявить с помощью операторов typedef, при переносе программы потребуется внести изменения только в эти операторы.

Перечисления(enum)

При написании программ часто возникает потребность определить несколько именованных констант, для которых требуется, чтобы все они имели различные значения (при этом конкретные значения могут быть не важны).

Для этого удобно воспользоваться перечисляемым типом данных, все возможные значения которого задаются списком целочисленных констант.

Формат: enum [имя_типа] { список_констант };

Имя типа задается в том случае, если в программе требуется определять переменные этого типа. Компилятор обеспечивает, чтобы эти переменные принимали значения только из списка констант. Константы должны быть целочисленными и могут инициализироваться обычным образом. При отсутствии инициализатора первая константа обнуляется, а каждой следующей присваивается на единицу большее значение, чем предыдущей:

```
enum Err {ERR_READ, ERR_WRITE, ERR_CONVERT};
Err error;
…
switch (error){
case ERR_READ: /* операторы */ break;
case ERR_WRITE: /* операторы */ break;
case ERR_CONVERT: /* операторы */ break; }
```

Константам ERR_READ, ERR_WRITE, ERR_CONVERT присваиваются значения 0, 1 и 2 соответственно.

Другой пример:

```
enum {two = 2, three, four, ten = 10, eleven, fifty = ten + 40};
```

Константам three и four присваиваются значения 3 и 4, константе eleven —11.

Имена перечисляемых констант должны быть уникальными, а значения могут совпадать. Преимущество применения перечисления перед описанием именованных констант и директивой #define состоит в том, что связанные константы нагляднее; кроме того, компилятор при инициализации констант может выполнять проверку типов. При выполнении арифметических операций перечисления преобразуются в целые. Поскольку перечисления являются типами, определяемыми пользователем, для них можно вводить собственные операции.

Структуры (struct)

В отличие от массива, все элементы которого однотипны, структура может содержать элементы разных типов.
В языке C++ структура является видом класса и обладает всеми его свойствами.
Во многих случаях достаточно использовать структуры так, как они определены в языке C:

```
struct [ имя_типа ] {
    тип_1 элемент_1;
    тип_2 элемент_2;
    …
    тип_n элемент_n;
} [ список_описателей ];
```

Элементы структуры называются полями структуры и могут иметь любой тип, кроме типа этой же структуры, но могут быть указателями на него.

Если отсутствует имя типа, должен быть указан список описателей переменных, указателей или массивов. В этом случае описание структуры служит определением элементов этого списка:

```
// Определение массива структур и указателя на структуру:
        struct {
            char fio[30];
            int date, code;
            double salary; }
        staff[100], *ps;
```

Если список отсутствует, описание структуры определяет новый тип, имя которого можно использовать в дальнейшем наряду со стандартными типами, например:

```
struct Worker{
// описание нового типа Worker
char fio[30];
int date, code;
double salary; }; // описание заканчивается тчк с запятой
// опр. массива типа Worker и указателя на тип Worker:
Worker staff[100]. *ps;
```

Имя структуры можно использовать сразу после его объявления (определение можно дать позднее) в тех случаях, когда компилятору не требуется знать размер структуры, например:

```
struct List;// объявление структуры List
struct Link{
        List *p; // указатель на структуру List
        Link *prev, *succ; // указатели на структуру Link
 };
struct List { /* определение структуры List */};
```

Это позволяет создавать связные списки структур.

Для инициализации структуры значения ее элементов перечисляют в фигурных скобках в порядке их описания:

```
struct{
    char fio[30];
    int date, code;
    double salary;
}worker = {"Страусенко“, 31,215, 3400.55};
```

При инициализации массивов структур следует заключать в фигурные скобки каждый элемент массива (учитывая, что многомерный массив — это массив массивов):

```
struct complex{
    float real, im; }
compl [2][3] = {
{{1, 1}, {1, 1}, {1, 1}}, // строка 1, то есть массив compl[0]
{{2, 2}, {2, 2}, {2, 2}}  // строка 2, то есть массив compl[1]
};
```

Для переменных одного и того же структурного типа определена **операция присваивания**, при этом происходит поэлементное копирование.

Структуру можно передавать в функцию и возвращать в качестве значения функции. Другие операции со структурами могут быть определены пользователем. Размер структуры не обязательно равен сумме размеров ее элементов, поскольку они могут быть выровнены по границам слова.

Доступ к полям структуры выполняется с помощью операций выбора **.** (точка) при обращении к полю через имя структуры и **->** при обращении через указатель, например: Worker worker, staff[100], *ps;

• • •

```
worker.fio = "Страусенко“;
staff[8].code = 215;
ps->salary = 0.12;
```

Если элементом структуры является другая структура, то доступ к ее элементам выполняется через две операции выбора:

```
struct A {int a; double x;};
struct B {A a; double x;} x[2];
 x[0].a.a = 1;
x[1].x = 0.1;
```

Как видно из примера, поля разных структур могут иметь одинаковые имена, поскольку у них разная область видимости. Более того, можно объявлять в одной области видимости структуру и другой объект (например, переменную или массив) с одинаковыми именами, если при определении структурной переменной использовать слово struct, но это создаёт дополнительные трудности.

Битовые поля

это особый вид полей структуры. Они используются для плотной упаковки данных, например, флажков типа «да/нет». Минимальная адресуемая ячейка памяти — 1 байт, а для хранения флажка достаточно одного бита. При описании битового поля после имени через двоеточие указывается длина поля в битах (целая положительная константа):

```
struct Options{
    bool centerX:1;
    bool centerY:1;
    unsigned int shadow:2;
    unsigned int palette:4;
};
```

Битовые поля могут быть любого целого типа. Имя поля может отсутствовать, такие поля служат для выравнивания на аппаратную границу. Доступ к полю осуществляется обычным способом — по имени. Адрес поля получить нельзя, однако в остальном битовые поля можно использовать точно так же, как обычные поля структуры. Следует учитывать, что операции с отдельными битами реализуются гораздо менее эффективно, чем с байтами и словами, так как компилятор должен генерировать специальные коды, и экономия памяти под переменные оборачивается увеличением объема кода программы. Размещение битовых полей в памяти зависит от компилятора и аппаратуры.

Объединения (union)

представляет собой частный случай структуры, все поля которой располагаются по одному и тому же адресу. Формат описания такой же, как у структуры, только вместо ключевого слова struct используется слово union. Длина объединения равна наибольшей из длин его полей. В каждый момент времени в переменной типа объединение хранится только одно значение, и ответственность за его правильное использование лежит на программисте. Объединения применяют для экономии памяти в тех случаях, когда известно, что больше одного поля одновременно не требуется:

```
#include <iostream.h>
int main(){
enum paytype {CARD, CHECK};
paytype ptype: union payment{ char card[25]; long check; } info;
/* присваивание значений info и ptype */
switch (ptype){
case CARD: cout « "Оплата по карте: " « info.card; break;
case CHECK: cout « "Оплата чеком: " « info.check; break; } return 0; }
```

Объединение часто используют в качестве поля структуры, при этом в структуру удобно включить дополнительное поле, определяющее, какой именно элемент объединения используется в каждый момент. Имя объединения можно не указывать, что позволяет обращаться к его полям непосредственно:

```
#include <iostream.h>
int main(){
enum paytype {CARD, CHECK};
struct{ paytype ptype; union{ char card[25]; long check; }; } info; ...
/* присваивание значения info */
switch (info.ptype){
case CARD: cout « "Оплата по карте: " « info.card; break;
case CHECK: cout « "Оплата чеком: " « info.check; break; }
return 0; }
```

Объединения применяются также для разной интерпретации одного и того же битового представления (но, как правило, в этом случае лучше использовать явные операции преобразования типов). В качестве примера рассмотрим работу со структурой, содержащей битовые поля:

```
struct Options{
        bool centerX:1;
        bool centerY:1;
        unsigned int shadow:2;
        unsigned int palette:4;
};
union{ unsigned char ch; Options bit; }option = {0xC4};
cout « option.bit.palette;
option.ch &= 0xF0; // наложение маски
```

По сравнению со структурами на объединения налагаются некоторые ограничения.

□ объединение может инициализироваться только значением его первого элемента;

□ объединение не может содержать битовые поля;

□ объединение не может содержать виртуальные методы, конструкторы, деструкторы и операцию присваивания;

□ объединение не может входить в иерархию классов.

The End