Составные (агрегированные) типы данных

Наумов Д.А., доц. каф. КТ

Операционные системы и системное программное обеспечение, 2020

Содержание лекции

Массивы

Массив – это набор переменных одного типа, имеющих одно и то же имя. Доступ к конкретному элементу массива осуществляется с помощью **индекса**.

В языке C все массивы располагаются в отдельной непрерывной области памяти. Первый элемент массива располагается по самому меньшему адресу, а последний — по самому большому. В языке C массивы и указатели тесно связаны. Объявление массива:

```
<<ElementsType>> <<ArrayName>>[<<ElementsCount>>];
```

Массив целых чисел длиной 10 элементов:

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

По сути, это блок из десяти последовательных объектов с именами `a[0], a[1], ..., a[9];`

Запись `a[i]` обозначает i-й элемент массива. Индексация элементов массива в языке C всегда начинается с нуля.

Представление массивов в памяти

Объем памяти, необходимый для хранения массива, непосредственно определяется его типом и размером. Для одномерного массива количество байтов памяти вычисляется следующим образом:

`количество_байтов = sizeof(базовый_тип) × длина_массива`

Во время выполнения программы на C не проверяется ни соблюдение границ массивов, ни их содержимое. В область памяти, занятую массивом, может быть записано что угодно, даже программный код.

Программист должен сам, где это необходимо, ввести проверку границ индексов. Элементы одномерного массива хранятся в непрерывной области памяти в порядке индексации. Ниже показано, как хранится в памяти массив a, начинающийся по адресу 1000 и объявленный как `char a[8]`:

Элемент	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
Адрес	1000	1001	1002	1003	1004	1005	1006	1007

Пример

Следующий пример программы компилируется без ошибки, однако при выполнении происходит нарушение границы массива `count` и разрушение соседних участков памяти:

```
int count[10], i;
/* здесь нарушена граница массива count */
for(i=0; i<100; i++)
    count[i] = i;
```

Что произойдет при запуске такой программы?

Двухмерные массивы

Стандартом C определены многомерные массивы. Простейшая и самая распространенная форма многомерного массива - двухмерный массив. Двухмерный массив - это массив одномерных массивов.

Объявление двухмерного массива `d` с размерами 10 и 20 выглядит следующим образом:

```
int d[10][20];
```

В языке C каждый индекс заключен в свои квадратные скобки.

Аналогично обращению к элементу одномерного массива, обращение к элементу с индексами 1 и 2 двухмерного массива `d` выглядит так:

```
d[1][2]
```

Инициализация массивов

В языке С массивы при объявлении можно инициализировать. Общая форма инициализации массива аналогична инициализации переменной:

```
тип имя_массива[размер1]...[размерN] = список_значений;
```

- Список_значений представляет собой список констант, разделенных запятыми.
- Типы констант должны быть совместимыми с типом массива.
- Первая константа присваивается первому элементу массива, вторая - второму и так далее.
- После закрывающейся фигурной скобки точка с запятой обязательна.

В следующем примере массив целых из 10 элементов инициализируется числами от 1 до 10:

```
int i[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Здесь элементу `i[0]` присваивается 1, а `i[9]` - 10.

Инициализация многомерных массивов

Многомерные массивы инициализируются так же, как и одномерные. В следующем примере массив sqrs инициализируется числами от 1 до 5 и их квадратами:

```
int sqrs[5][2] = {
    1, 1,
    2, 4,
    3, 9,
    4, 16,
    5, 25
};
```

Инициализируя многомерный массив, для улучшения наглядности элементы инициализации каждого измерения можно заключать в фигурные скобки.

```
int sqrs[5][2] = {
    {1, 1},
    {2, 4},
    {3, 9},
    {4, 16},
    {5, 25}
};
```

Содержание лекции

Указатели

Указатель – это переменная, значением которой является адрес некоторого объекта (обычно другой переменной) в памяти компьютера.

Например, если одна переменная содержит адрес другой переменной, то говорят, что первая переменная указывает (ссылается) на вторую.

Указательная переменная

Объявление указателя состоит из имени базового типа, символа * и имени переменной.

Общая форма объявления указателя следующая:

```
тип *имя;
```

Здесь **тип** – это базовый тип указателя, им может быть любой правильный тип. Имя определяет имя переменной-указателя.

Базовый тип указателя определяет тип объекта, на который указатель будет ссылаться. Фактически указатель любого типа может ссылаться на любое место в памяти. Однако выполняемые с указателем операции существенно зависят от его типа. Например, если объявлен указатель типа `int *`, компилятор предполагает, что любой адрес, на который он ссылается, содержит переменную типа `int`, хоть это может быть и не так. Следовательно, объявляя указатель, необходимо убедиться, что его тип совместим с типом объекта, на который он будет ссылаться.

Операции для работы с указателями

В языке C определены две операции для работы с указателями: * и &. Унарная операция возвращает адрес объекта. Например, операция

```
m = &count;
```

присваивает переменной m адрес переменной count. Можно сказать, что адрес - это номер первого байта участка памяти, в котором хранится переменная.

Предположим, переменная count хранится в ячейке памяти под номером 2000, а ее значение равно 100. Тогда переменной m будет присвоено значение 2000.

Унарная операция * называется операцией ссылки по указателю (indirection) или разыменования (dereferencing). Применяя ее к указателю, получаем объект, на который он указывает. Например, если m содержит адрес переменной count, то оператор

```
q = *m;
```

присваивает переменной q значение переменной count. Таким образом, q получит значение 100, потому что по адресу 2000 расположена переменная count, которая имеет значение 100.

Операции для работы с указателями

```
int x, *p;
x = 3;
p = &x;
*p = 5;
```

Чему равно значение переменной x?

Указательные выражения

В выражениях с указателями можно использовать операции присваивания, преобразования типов, а так же сложения и вычитания.

Указатель можно использовать в правой части оператора присваивания для присваивания его значения другому указателю. Если оба указателя имеют один и тот же тип, то выполняется простое присваивание, без преобразования типа:

```
int x = 99;
int *p1, *p2;
p1 = &x;
p2 = p1;
```

Оба указателя (p1 и p2) ссылаются на x. Допускается присваивание указателя одного типа указателю другого типа.Однако для этого необходимо выполнить явное преобразование типа указателя.

Преобразование типа указателя

Указатель можно преобразовать к другому типу. Эти преобразования бывают двух видов: с использованием указателя типа `void *` и без его использования.

В языке С допускается присваивание указателя типа `void *` указателю любого другого типа (и наоборот) без явного преобразования типа указателя.

Тип указателя `void *` используется, если тип объекта неизвестен. Например, использование типа `void *` в качестве параметра функции позволяет передавать в функцию указатель на объект любого типа, при этом сообщение об ошибке не генерируется.
Также он полезен для ссылки на произвольный участок памяти, независимо от размещенных там объектов.

В отличие от `void *`, преобразования всех остальных типов указателей должны быть всегда явными (т.е. должна быть указана операция приведения типов).
Однако следует учитывать, что преобразование одного типа указателя к другому может вызвать непредсказуемое поведение программы.

Преобразование типа указателя

В следующей программе делается попытка присвоить значение x переменной y посредством указателя p. При компиляции программы сообщение об ошибке не генерируется, однако результат работы программы неверен.

```
#include <stdio.h>
int main(void) {
    double x = 100.1, y;
    int *p;
    /* В следующем операторе указателю на целое p
    (присваивается значение, ссылающееся на double. */
    p = (int *) &x;
    /* Следующий оператор работает не так, как ожидается. */
    y = *p;
    /* Следующий оператор не выведет число 100.1. */
    printf("Значение x равно: %f (Это не так!)", y);
    return 0;
}
```

Адресная арифметика

В языке C допустимы только две арифметические операции над указателями: сложение и вычитание.

Предположим, текущее значение указателя p1 типа int * равно 2000. Предположим также, что переменная типа int занимает в памяти 4 байта.

Тогда после операции увеличения

```
p1++;
```

указатель p1 принимает значение 2004, а не 2001. То есть, при увеличении на 1 указатель p1 будет ссылаться на следующее целое число.

Это же справедливо и для операции уменьшения. Например, если p1 равно 2000, то после выполнения оператора

```
p1--;
```

значение p1 будет равно 1996.

Адресная арифметика

Операции адресной арифметики подчиняются следующим правилам:

- После выполнения операции увеличения над указателем, данный указатель будет ссылаться на следующий объект своего базового типа.
- После выполнения операции уменьшения - на предыдущий объект.

Операции адресной арифметики не ограничены увеличением (инкрементом) и уменьшением (декрементом).

Например, к указателям можно добавлять целые числа или вычитать из них целые числа. Выполнение оператора

```
p1 = p1 + 12;
```

"передвигает"указатель `p1` на 12 объектов в сторону увеличения адресов.

Инициализация указателей

После объявления нестатического локального указателя до первого присвоения он содержит неопределенное значение. (Глобальные и статические локальные указатели при объявлении неявно инициализируются нулем.) Если попытаться использовать указатель перед присвоением ему нужного значения, то скорее всего он мгновенно разрушит программу или всю операционную систему. Это очень досадная ошибка.

При работе с указателями большинство программистов придерживаются следующего важного соглашения: указатель, не ссылающийся в текущий момент времени должным образом на конкретный объект, должен содержать нулевое значение. Нуль используется потому, что C гарантирует отсутствие чего-либо по нулевому адресу. Следовательно, если указатель равен нулю, то это значит, во-первых, что он ни на что не ссылается, а во-вторых — что его сейчас нельзя использовать.

Инициализация указателей

Указателю можно задать нулевое значение, присвоив ему 0. Например, следующий оператор инициализирует p нулем:

```
char *p = 0;
```

Дополнительно к этому во многих заголовочных файлах языка C, например, в <stdio.h> определен макрос NULL, являющийся нулевой указательной константой. Поэтому в программах на C часто можно увидеть следующее присваивание:

```
p = NULL;
```

Однако равенство указателя нулю не делает его абсолютно "безопасным". Использование нуля в качестве признака неподготовленности указателя - это только соглашение программистов, но не правило языка C.

Указатели и аргументы функций

В языке Pascal существует особый вид параметров функции – var-параметры. Переменным, переданным как var-параметры в функции может быть присвоено новое значение, и это новое значение переменной сохранится после выхода из функции.

В языке C аргументы передаются в функции только по значению, и не существует прямого способа изменить переменную, переданную через параметр функции. Добиться нужного эффекта можно передавая в функцию не саму переменную, а указатель на нее.

В следующем примере показана функция swap, выполняющая обмен значений двух целочисленных переменных:

```
void swap(int *px, int *py) /* обмен местами *px и *py */
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
```

Функция swap вызывается как swap(&a, &b);.

Указатели и массивы

Понятия указателей и массивов тесно связаны. Рассмотрим следующий фрагмент программы:

```
char str[80], *p1;
p1 = str;
```

Здесь `p1` указывает на первый элемент массива `str`. Обратиться к пятому элементу массива `str` можно с помощью любого из двух выражений:

```
str[4]
*(p1+4)
```

Массив начинается с нуля. Поэтому для пятого элемента массива `str` нужно использовать индекс 4. Можно также увеличить `p1` на 4, тогда он будет указывать на пятый элемент.

Как и объекты любых других типов, указатели могут быть собраны в массив. В следующем операторе объявлен массив из 10 указателей на объекты типа `int`:

```
int *x[10];
```

Для присвоения, например, адреса переменной `var` третьему элементу массива указателей, необходимо написать:

```
x[2] = &var;
```

Указатели на функции

Указатели на функции – очень мощное средство языка С. Функция располагается в памяти по определенному адресу, который можно присвоить указателю в качестве его значения. Адресом функции является ее точка входа. Именно этот адрес используется при вызове функции. Так как указатель хранит адрес функции, то она может быть вызвана с помощью этого указателя. Он позволяет также передавать ее другим функциям в качестве аргумента.

Описание указателя на функцию и вызов функции по указателю показаны ниже. Предполагается, что ранее была описана функция `max2`, принимающая два аргумента типа `int` и возвращающее значение типа `int`.

```
int a = -5, b = 3, c;
int (*p)(int, int);/* указатель на функцию */
p = max2;
c = (*p)(a,b);/* вызов функции по указателю */
```

При описании указателя на функцию и ее вызове имя функции и знак * необходимо брать в круглые скобки `(*p)`.

Сложные объявления с указателями

В декларациях обычно используется имя (идентификатор) и один из модификаторов *, [ ] и ( ), причем разрешается использовать более одного модификатора в одной декларации. Для раскрытия этих деклараций применяются следующие правила:

1. Чем ближе модификатор стоит к идентификатору, тем выше его приоритет.
2. Приоритет ( ) и [ ] выше, чем приоритет *.
3. Приоритет повышается заключением в скобки ().

```
char **argv
int (*daytab)[13]
int *daytab[13]
void *comp()
void (*comp)()
char (*(*x()) []) ()
char (*(*x[3]) ()) [5]
```

```
char **argv
/* argv: указатель на указатель на char */

int (*daytab)[13]
/* daytab: указатель на массив[13] типа int */

int *daytab[13]
/* daytab: массив[13] указателей на int */

void *comp()
/* comp: функция, возвращающая указатель на void */

void (*comp)()
/* comp: указатель на функцию, возвращающую void */

char (*(*x()) []) ()
/* x: функция, возвращающая указатель на массив []
 * указателей на функцию, возвращающую char */

char (*(*x[3]) ()) [5]
/* x: массив [3] указателей на функцию, возвращающую
 * указатель на массив[5] типа char */
```

Содержание лекции

Динамическое выделение памяти

Указатели используются для динамического выделения памяти компьютера для хранения данных. Динамическое распределение означает, что программа выделяет память для данных во время своего выполнения.

Память, выделяемая в C функциями динамического распределения данных, находится в т.н. динамически распределяемой области памяти (heap). Динамически распределяемая область памяти — это свободная область памяти, не используемая программой, операционной системой или другими программами. Размер динамически распределяемой области памяти заранее неизвестен, но как правило в ней достаточно памяти для размещения данных программы.

Основу системы динамического распределения в C составляют функции `malloc()` и `free()`. Эти функции работают совместно. Функция `malloc()` выделяет память, а `free()` — освобождает ее. В программу, использующую эти функции, должен быть включен заголовочный файл `<stdlib.h>`.

Функция malloc()

Прототип функции malloc() следующий:

```
void *malloc(size_t <<number_of_bytes>>);
```

Здесь «number_of_bytes» – размер памяти, необходимой для размещения данных. (Тип size_t определен в <stdlib.h> как некоторый целый без знака.)

Функция malloc() возвращает указатель типа void *, поэтому его можно присвоить указателю любого типа.

При успешном выполнении malloc() возвращает указатель на первый байт непрерывного участка памяти, выделенного в динамически распределяемой области памяти. Если в динамически распределяемой области памяти недостаточно свободной памяти для выполнения запроса, то память не выделяется и malloc() возвращает нуль. При выполнении следующего фрагмента программы выделяется непрерывный участок памяти объемом 1000 байтов:

```
char *p;
p = malloc(1000); /* выделение 1000 байтов */
```

После присвоения указатель p ссылается на первый из 1000 байтов выделенного участка памяти.

Функция malloc()

В следующем примере выделяется память для 50 целых чисел. Для повышения мобильности (переносимости программы с одной машины на другую) используется оператор sizeof.

```
int *p;
p = malloc(50 * sizeof(int));
```

Поскольку динамически распределяемая область памяти не бесконечна, при каждом размещении данных необходимо проверять, состоялось ли оно. Если malloc() не смогла по какой-либо причине выделить требуемый участок памяти, то она возвращает нуль:

```
p = malloc(100);
if(!p) {
    printf("Нехватка памяти.\n");
    exit(1);
}
```

Функция free()

Функция free() противоположна функции malloc() в том смысле, что она возвращает системе участок памяти, выделенный ранее с помощью функции malloc().

Функция free() имеет следующий прототип:

```
void free(void *p)
```

Здесь p — указатель на участок памяти, выделенный перед этим функцией malloc().

Функцию free() нельзя вызывать с неправильным аргументом, это разрушит всю систему распределения памяти.

Пример - динамический массив

```c
#include <stdlib.h>
int main(void)
{
    int *a;
    int i, n = 80;

    a = malloc(n*sizeof(int));
    if(!a) {
        printf("Требуемая память не выделена.\n");
        exit(1);
    }

    for(i = 0;i < n;i++)
        a[i]=i;

    free(a);
    return 0;
}
```

Трудности при работе с указателями

Указатели являются очень мощным средством языка C. В то же время неумелое использование указателей может привести к появлению в программе нестабильных, труднообнаруживаемых ошибок. Это привело к тому, что в языках-потомках C/C++ (Java, C) было принято решение отказаться от непосредственной работы с указателями.

Классический пример ошибки – неинициализированный указатель.

```
/* Это программа содержит ошибку. */
int main(void) {
    int x, *p;
    x = 10;
    *p = x; /* ошибка, p не инициализирован */
    return 0;
}
```

Эта программа присваивает значение 10 некоторой неизвестной области памяти.

Содержание лекции

Строки

В языке программирования C отсутствует специальный тип для представления строк, подобный типу string в языке Pascal.

В C строки хранятся как массив символов. При этом размер массива должен быть как минимум на 1 больше числа символов в строке, т.к. после последнего символа строки хранится значение 0, обозначающее конец строки. Отсутствие в конце строки завершающего нуля почти наверняка приведет к аварийному завершению программы!

При объявлении строки можно выполнить ее инициализацию строковым литералом:

```
char s[8] = "example";
```

Данная строка приведет к созданию массива со следующими элементами:

Элемент	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
Значение	'e'	'x'	'a'	'm'	'p'	'l'	'e'	0

Операции со строками

Операции со строками в языке C реализованы с помощью функций, находящихся в заголовочном файле <string.h>:

Имя функции	**Выполняемое действие**
strcpy(s1, s2)	Копирование s1 в s2
strcat(s1, s2)	Конкатенация (присоединение) s2 в конец s1
strlen(s1)	Возвращает длину строки s1
strcmp(s1, s2)	Возвращает 0, если s1 и s2 совпадают, отрицательное значение, если s1 < s2 и положительное, если s1 > s2
strchr(s1, ch)	Возвращает указатель на первое вхождение символа ch в строку s1
strstr(s1, s2)	Возвращает указатель на первое вхождение строки s2 в строку s1

Пример работы со строками

```c
#include <stdio.h>
#include <string.h>
int main(void) {
    char s1[80], s2[80];
    gets(s1);
    gets(s2);
    printf("Длина: %d %d\n", strlen(s1), strlen(s2));

    if(strcmp(s1, s2)==0)
        printf("Строки равны\n");

    strcat(s1, s2);
    printf("%s\n", s1);
    strcpy(s1, "Проверка.\n");
    printf(s1);
    if(strstr("Привет", "ив"))
        printf(" найдено ив ");

    return 0;
}
```

Содержание лекции

Структуры

Структура в языке **C** представляет собой аналог записи в языке **Pascal**. Объявление структуры имеет следующий синтаксис:

```
struct <<StructureName>> { //имя типа структуры
    <<Fields Definition>> //описание полей
};
```

В конце описания структуры ставится точка с запятой. Описание переменной структурного типа в языке C имеет вид:

```
struct StructureName VariableName;
```

Необходимость указания ключевого слова `struct` при объявлении структурной переменной является особенностью языка C и отличает его от других языков программирования – C++, Pascal и др.

Доступ к значениям полей структуры осуществляется с помощью операции “.”.

Пример работы со структурой

```
struct addr {
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    unsigned long int zip;
};

struct addr addr_info;

addr_info.zip = 390005;
```

Структуры

При объявлении структуры можно сразу же объявить одну или несколько переменных структурного типа.

Например:

```
struct addr {
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    unsigned long int zip;
} addr_info, binfo, cinfo;
```

Массивы структур

Структуры часто образуют массивы. Чтобы объявить массив структур, вначале необходимо определить структуру (то есть определить агрегатный тип данных), а затем объявить переменную массива этого же типа.

Например:

```
struct addr addr_list[100];
```

Это выражение создаст 100 наборов переменных, каждый из которых организован так, как определено в структуре addr.

Чтобы получить доступ к определенной структуре, нужно указать имя массива с индексом. Например, выведем значение ZIP-кода из третьей структуры:

```
printf("%d", addr_list[2].zip);
```

Как и в других массивах переменных, в массивах структур индексирование начинается с 0.

Передача структур функциям

Над структурами в целом одного типа можно выполнять операции присваивания. Одной из разновидностей такой операции является передача в функцию структурной переменной как аргумента.

В следующем примере вводится структура, описывающая координаты точки и функция, вычисляющая координаты середины отрезка:

```
struct point {
    float x;
    float y;
};

struct point segment_middle(struct point p1, struct point p2)
{
    struct point result;
    result.x = (p1.x + p2.x) / 2;
    result.y = (p1.y + p2.y) / 2;
    return result;
}
```

Указатели на структуры

В языке C указатели на структуры имеют некоторые особенности.

Чтобы объявить указатель на структуру нужно так же, как и в случае обычных указателей, поместить символ звездочки * перед именем переменной. Например:

```
struct addr *addr_pointer;
```

Чтобы получить по указателю доступ к полям структуры используется операция стрелка “->”:

```
addr_pointer->zip = 390000;
```

Операции обращения к полям точка и стрелка имеют более высокий приоритет, чем остальные. Поэтому в выражении `++p->len` инкрементируется `len`, а не `p`.

Указатели на структуры

В языке С указатели на структуры используются в двух случаях:

- при передаче структуры функции по указателю,
- при организации динамических структур данных – списков, очередей, стеков и т.п.

Во втором случае в структуру включается поле – указатель на структуру такого же типа. Заметим, что структура не может содержать в качестве поля структуру такого же типа, а содержать указатель на структуру – может.

Далее описывается возможная структура записи, соответствующей элементу списка:

```
struct list_item {
    int data[50];
    struct list_item *next;
};
```

Объединения

Объединение – это переменная, которая может содержать объекты различных типов и размеров (но не одновременно); при этом удовлетворение требований к размеру и выравниванию возлагается на компилятор.

С помощью объединений можно работать с данными различных типов в пределах одного участка памяти, не привнося в программу элементы низкоуровневого, машинно-зависимого программирования.

Объявление объединения (начинается с ключевого слова `union`) похоже на объявление структуры и в общем виде выглядит так:

```
union <<Tag>>{
<<type1>> <<Name1>>;
<<type2>> <<Name2>>;
<<type3>> <<Name3>>;
} <<VariableNamesList>>;
```

Объединения

Например:

```
union u_type {
    int i;
    char ch;
};
```

Это объявление не создает никаких переменных. Чтобы объявить переменную, ее имя можно поместить в конце объявления или написать отдельный оператор объявления. Чтобы с помощью только

что написанного кода объявить переменную-объединение, которая называется `cnvt` и имеет тип `u_type`, можно написать следующий оператор:

```
union u_type cnvt;
```

В `cnvt` одну и ту же область памяти занимают целая переменная `i` и символьная переменная `ch`. Конечно, `i` занимает 2 байта (при условии, что целые значения занимают по 2 байта), а `ch` — только 1.

Битовые поля

В языке C имеется встроенная поддержка битовых полей, которая дает возможность получать доступ к единичному биту. Битовые поля могут быть полезны по разным причинам, а именно:

- Если память ограничена, то в одном байте можно хранить несколько булевых переменных (принимающих значения ИСТИНА и ЛОЖЬ);
- Некоторые устройства передают информацию о состоянии, закодированную в байте в одном или нескольких битах;
- Для некоторых процедур шифрования требуется доступ к отдельным битам внутри байта.

Битовые поля

Хотя для решения этих задач можно успешно применять побитовые операции, битовые поля могут придать вашему коду больше упорядоченности.

Битовое поле может быть членом структуры или объединения. Оно определяет длину поля в битах.

Общий вид определения битового поля такой:

```
<<Type>> <<FieldName>> : <<Length>>
```

Здесь `тип` означает тип битового поля, а `длина` — количество бит, которые занимает это поле.

Тип битового поля может быть `int`, `signed` или `unsigned`.

Битовые поля

Например, информацию в байте состояния модема можно представить с помощью следующего битового поля:

```
struct status_type {
    unsigned delta_cts: 1;
    unsigned delta_dsr: 1;
    unsigned tr_edge: 1;
    unsigned delta_rec: 1;
    unsigned cts: 1;
    unsigned dsr: 1;
    unsigned ring: 1;
    unsigned rec_line: 1;
} status;
```

Для обращения к битовому полю используется тот же способ, что и для обращения к элементу структуры любого другого типа. Вот, например, фрагмент кода, выполняющий сброс поля `ring`:

```
status.ring = 0;
```

Перечисления

Перечисление — это набор именованных целых констант.
Перечисление в общем виде выглядит так:

```
enum тег {список_перечисления} список_переменных;
```

Здесь тег и список_переменных не являются обязательными. (Но хотя бы что-то одно из них должно присутствовать.)

```
enum day_of_week { mo, tu, we, th, fr, sa, su };
```

Тег перечисления можно использовать для объявления переменных данного перечислимого типа.

```
enum day_of_week today = mo;
```

При задании перечислений одной или нескольким константам можно задать связанные с ними значения:

```
enum day_of_week { mo=1, tu=2, we=3, th=4, fr=5, sa=6, su=0 };
```

Ключевое слово typedef

Новые имена типов данных можно определять, используя ключевое слово `typedef`. На самом деле таким способом новый тип данных не создается, а всего лишь определяется новое имя для уже существующего типа. Этот процесс может помочь сделать машинно-зависимые программы более переносимыми. Если вы для каждого машинно-зависимого типа данных, используемого в вашей программе, определяете данное вами имя, то при компиляции для новой среды придется менять только операторы `typedef`. Такие выражения могут помочь в самодокументировании кода, позволяя давать понятные имена стандартным типам данных. Общий вид декларации `typedef` (оператора `typedef`) такой:

```
typedef тип новое_имя;
```

где `тип` – это любой тип данных языка C, а `новое_имя` – новое имя этого типа. Новое имя является дополнением к уже существующему, а не его заменой.