Производные (пользовательские) типы данных

В языке С существует пять способов создания пользовательских типов:

- структура (structure) – агрегированный (или составной) тип данных, позволяющий объединить группу разнотипных переменных под одним именем
- битовое поле (bit-field) – вариант структуры, позволяющий работать с отдельными битами
- объединение (union) – позволяет использовать одну и ту же область памяти для хранения нескольких переменных разного типа
- перечисление (enumeration) – список именованных целочисленных констант
- ключевое слово typedef – присваивание существующему типу нового имени

Структуры

Структура объединяет под одним именем логически связанные данные разных типов

С помощью структур удобно размещать в смежных полях связанные между собой элементы информации

Объявление структуры создает шаблон, который можно использовать для создания ее объектов или переменных (то есть экземпляров этой структуры)

Переменные, из которых состоит структура, называются *членами, элементами* или *полями*

Структуры: определение структурного типа данных

Структурный тип данных определяется следующим описанием:

```
struct имя_типа_структуры {
   тип_элемента имя_элемента; /*Описание элементов
    структуры*/
   …
   тип_элемента имя_элемента;
};
```

Пример:

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

Объявление переменной типа структура

Структурная переменная описывается с помощью структурного типа данных.

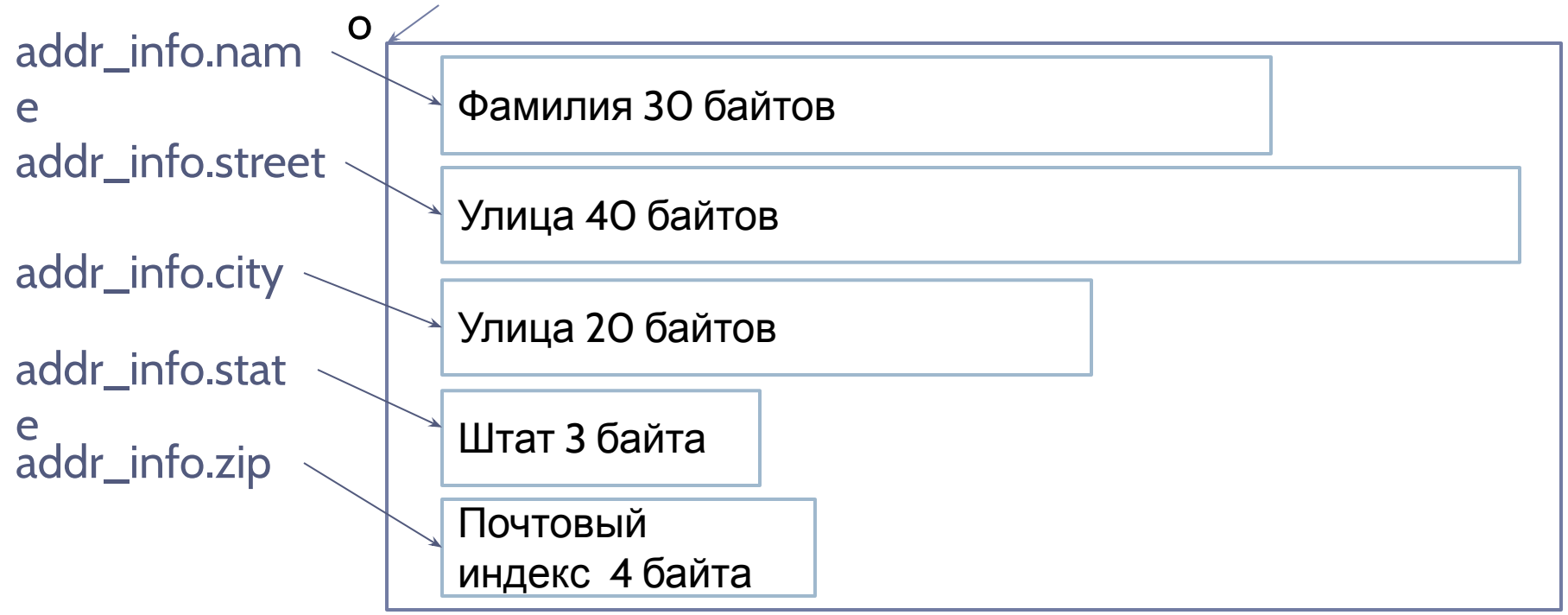
```
struct имя_типа_структуры
    имя_структурной_переменной;
```

Примеры:

```
struct  addr addr_info[7];/* массив структур */
struct addr addr_info;      /* одна структурная переменная */
struct addr *addr_info;     /* указатель на структурную переменную */
```

Выделение памяти под объекты структуры

- Когда объявляется переменная-структура, компилятор автоматически выделяет количество памяти, достаточное, чтобы разместить все ее члены



Определение типа структура и одновременное объявление переменных

Одновременно с определением структурного типа данных (структуры) можно объявить несколько ее экземпляров – структурных переменных:

```
struct имя_типа_структуры {
   тип_элемента имя_элемента; /*Описание элементов
    структуры*/
   …
   тип_элемента имя_элемента;
} имена_структурных_переменных;
```

Пример:

```
struct dinner {
   char *place;
   float cost;
   struct dinner *next;
} Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday;
```

Доступ к компонентам структуры

Доступ к отдельным членам структуры осуществляется с помощью оператора . (который обычно называют *оператором точка* или *оператором доступа к члену структуры*).

Например, в следующем выражении полю zip в уже объявленной переменной-структуре addr_info присваивается значение ZIP-кода, равное 12345:

```
addr_info.zip = 12345;
```

Этот отдельный член определяется именем объекта (в данном случае addr_info), за которым следует точка, а затем именем самого этого члена (в данном случае zip). В общем виде использование оператора точка для доступа к члену структуры выглядит таким образом:

```
имя-объекта.имя-члена
```

Поэтому, чтобы вывести ZIP-код на экран, напишите следующее:

```
printf("%d", addr_info.zip);
```

Будет выведен ZIP-код, который находится в члене zip переменной-структуры addr_info.

Доступ к компонентам структуры

Точно так же в вызове gets() можно использовать массив символов addr_info.name:

```
gets(addr_info.name);
```

Таким образом, в начало name передается указатель на символьную строку.

Так как name является массивом символов, то чтобы получить доступ к отдельным символам в массиве addr_info.name, можно использовать индексы вместе с name.

Например, с помощью следующего кода можно посимвольно вывести на экран содержимое addr_info.name:

```
for(t=0; addr_info.name[t]; ++t) putchar(addr_info.name[t]);
```

Обратите внимание, что индексируется именно name (а не addr_info).

Помните, что addr_info — это имя всего объекта-структуры, а name — имя элемента этой структуры. Таким образом, если требуется индексировать элемент структуры, то индекс необходимо

Присваивание структур

- Информация, которая находится в одной структуре, может быть присвоена другой структуре того же типа при помощи единственного оператора присваивания

```
#include <stdio.h>
int main(void)
{
 struct {
 int a;
int b;
} x, y;
x.a = 10;
y = x; /* присвоение одной структуры другой */
printf("%d", y.a);
return 0;
}
```

Массивы структур

Структуры часто образуют массивы

Чтобы объявить массив структур, вначале необходимо определить тип данных структура, а затем объявить переменную массива этого же типа

Например:

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

Чтобы получить доступ к определенной структуре, необходимо указывать имя массива с индексом

Например:

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

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

Например, в результате выполнения следующего выражения первому символу члена name, находящегося в третьей структуре из addr_list, присваивается значение 'X'.

```
addr_list[2].name[0] = 'X';
```

Передача структур функциям: передача членов структур функциям

При передаче функции члена структуры передается его значение, притом не играет роли то, что значение берется из члена структуры

Например:

```
struct fred{
  char x;
  int y;
  float z;
  char s[10];
} mike;
```

Каждый член этой структуры передается функции следующим образом:

```
func(mike.x);    /* передается символьное значение x */
func2(mike.y);   /* передается целое значение y */
func3(mike.z);   /* передается значение с плавающей точкой z */
func4(mike.s);   /* передается адрес строки s */
func(mike.s[2]); /* передается символьное значение s[2] */
```

Передача структур функциям: передача членов структур функциям

Если же нужно передать *адрес* отдельного члена структуры, то перед именем структуры должен находиться оператор &

Например:

```
func(&mike.x);    /* передается адрес символа x */
func2(&mike.y);   /* передается адрес целого y */
func3(&mike.z);   /* передается адрес члена z с плавающей точкой */
func4(mike.s);    /* передается адрес строки s */
func(&mike.s[2]); /* передается адрес символа в s[2] */
```

Обратите внимание, что оператор & стоит непосредственно перед именем структуры, а не перед именем отдельного члена

Передача целых структур функциям

Когда в качестве аргумента функции используется структура, то для передачи целой структуры используется обычный способ вызова по значению

При использовании структуры в качестве аргумента тип аргумента должен соответствовать типу параметра

При объявлении параметров, являющихся структурами, объявление типа структуры должно быть глобальным, чтобы структурный тип можно было использовать во всей программе

```
#include <stdio.h>
 struct struct_type {/* Определение типа структуры. */
  int a, b;
  char ch;
} ;
 void f1(struct struct_type parm);
 int main(void)
{
  struct struct_type arg;
   arg.a = 1000;
   f1(arg);
   return 0;
```

Указатели на структуры

В языке С указатели на структуры имеют некоторые особенности

Объявление указателя на структуру

Как и другие указатели, указатель на структуру объявляется с помощью звездочки *, которую помещают перед именем переменной структуры

Например:

```
struct addr *addr_pointer;
```

Использование указателей на структуры

Указатели на структуры используются главным образом в двух случаях:

- когда структура передается функции с помощью вызова по ссылке
- когда создаются связанные друг с другом списки и другие структуры с динамическими данными, работающие на основе динамического размещения

Передача структуры функции по ссылке

Проблема: при передаче любых (кроме самых простых) структур функциям, имеется один большой недостаток: при выполнении вызова функции, чтобы поместить структуру в стек, необходимы существенные ресурсы

Решение проблемы: передача структуры по ссылке, то есть передача функции указатель на структуру. В этом случае в стек попадает только адрес структуры. В результате вызовы функции выполняются очень быстро. В некоторых случаях этот способ имеет еще и второе преимущество: передача указателя позволяет функции модифицировать содержимое структуры, используемой в качестве аргумента

Получение адреса переменной типа структура

Чтобы получить адрес переменной-структуры, необходимо перед ее именем поместить оператор &

Например:

```
struct bal {
  float balance;
  char name[80];
} person;
 struct bal *p;  /* объявление указателя на структуру */
```

адрес структуры person можно присвоить указателю p:

```
p = &person;
```

Доступ к членам структуры через указатель

Чтобы с помощью указателя на структуру получить доступ к ее членам, необходимо использовать оператор стрелка ->.

Например, как можно сослаться на поле balance:

```
p->balance
```

Оператор ->, который обычно называют *оператором стрелки*, состоит из знака "минус", за которым следует знак "больше". Стрелка применяется вместо оператора точки тогда, когда для доступа к члену структуры используется указатель на структуру.

Программа-имитатор таймера

```
#include <stdio.h>
#define DELAY 128000
struct my_time {
int hours;
int minutes;
int seconds;
} ;
void display(struct my_time *t);
void update(struct my_time *t);
void delay(void);
int main(void){
struct my_time systime;
systime.hours = 0; systime.minutes = 0; systime.seconds = 0;
for(;;) {
update(&systime);
display(&systime);
}
return 0;
}
void update(struct my_time *t) {
t->seconds++;
if(t->seconds==60) {
t->seconds = 0;
```

Массивы и структуры внутри структур

Членом структуры может быть также составной тип: массивы и структуры

Члены структуры, которые являются массивами, можно считать такими же членами структуры, как и члены простых типов

```
struct x {
  int a[10][10]; /* массив 10 x 10 из целых значений */
  float b;
} y;
```

Целый элемент с индексами 3, 7 из массива a, находящегося в структуре y, обозначается таким образом:

```
y.a[3][7]
```

Когда структура является членом другой структуры, то она называется *вложенной*

Например, в следующем примере структура address вложена в emp:

```
struct emp {
  struct addr address; /* вложенная структура */
  float wage;
} worker;
```

В следующем фрагменте кода элементу zip из address присваивается значение 93456.

```
worker.address.zip = 93456;
```

Объединения

Объединение — это место в памяти, которое используется для хранения переменных, разных типов. Объединение дает возможность интерпретировать один и тот же набор битов не менее, чем двумя разными способами. Объявление объединения (начинается с ключевого слова union) похоже на объявление структуры и в общем виде выглядит так:

```
union тег {
  тип имя-члена;
  тип имя-члена;
  ...
} переменные-этого-объединения;
```

Например:

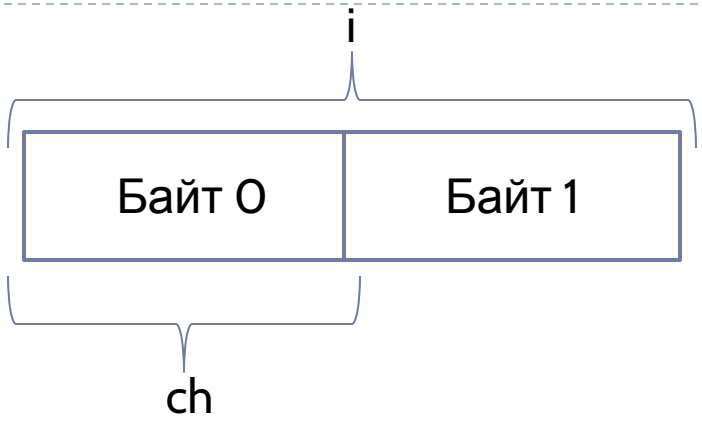
```
union u_type {
  int i;
  char ch;
};
```

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

Например:

```
union u_type cnvt;
```

Размещение в памяти переменной типа объединение




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

- В cnvt одну и ту же область памяти занимают целая переменная i и символьная переменная ch
- i занимает 2 байта (при условии, что целые значения занимают по 2 байта), а ch — только 1
- На рис. показано, каким образом i и ch пользуются одним и тем же адресом. В любом месте программы хранящиеся в cnvt данные можно обрабатывать как целые или символьные
- При объявлении экземпляра объединения, компилятор автоматически выделяет память, достаточную для хранения наибольшего члена объединения

Доступ к члену объединения

- Для получения доступа к члену объединения используйте тот же синтаксис, что и для структур: операторы точки и стрелки
- При работе непосредственно с объединением следует пользоваться точкой. А при получении доступа к объединению с помощью указателя нужен оператор стрелка

Например:

```
cnvt.i = 10;
```

В следующем примере функции func1 передается указатель на cnvt:

```
void func1(union u_type *un) {
un->i = 10; /* присвоение cnvt значение 10 с помощью
   указателя */
}
```

Битовые поля

В языке С 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.

Хотя для решения этих задач можно успешно применять побитовые операции, битовые поля могут придать вашему коду больше упорядоченности (и, возможно, с их помощью удастся достичь большей эффективности).

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

Общий вид определения битового поля такой:

тип имя : длина;

Здесь *тип* означает тип битового поля, а *длина* — количество бит, которые занимает это поле. Тип битового поля может быть int, signed или unsigned

Пример

```
struct emp {
struct addr address;
float pay;
unsigned lay_off:    1; /* временно уволенный или
   работающий */
unsigned hourly:     1; /* почасовая оплата или оклад */
unsigned deductions: 3; /* налоговые (IRS) удержания */
};
```

Здесь определены данные о работнике, для которых выделяется только один байт, содержащий информацию трех видов: статус работника, на окладе ли он, а также количество удержаний из его зарплаты.

Без битового поля эта информация занимала бы 3 байта.

Перечисления

Перечисление — это набор именованных целых констант. Перечисления довольно часто встречаются в повседневной жизни

Например, перечисление, в котором приведены названия монет, используемых в Соединенных Штатах:

penny (пенни, монета в один цент), nickel (никель, монета в пять центов), dime (монета в 10 центов), quarter (25 центов, четверть доллара), half-dollar (полдоллара), dollar (доллар)

Перечисления определяются во многом так же, как и структуры

Началом объявления перечислимого типа служит ключевое слово enum.

Перечисление в общем виде выглядит так:

enum *тег* {*список перечисления*} *список переменных*;

Здесь *тег* и *список переменных* не являются обязательными

Пример

```
enum coin { penny, nickel, dime, quarter, half_dollar, dollar};
```

Тег перечисления можно использовать для объявления переменных данного перечислимого типа

```
enum coin money;
```

С учетом этих объявлений совершенно верными являются следующие операторы:

```
money = dime;
if(money==quarter) printf("Денег всего четверть доллара.\n");
```

Каждый элемент перечислений представляет целое число

В таком виде элементы перечислений можно применять везде, где используются целые числа. Каждому элементу дается значение, на единицу большее, чем у его предшественника. Первый элемент перечисления имеет значение 0.

При выполнении кода

```
printf("%d %d", penny, dime);
```

на экран будет выведено 0 2.

Пример

```
enum coin { penny, nickel, dime, quarter=100, half_dollar,
   dollar};
```

вот какие значения появились у этих элементов:

```
penny          0
nickel      1
dime        2
quarter     100
half_dollar    101
dollar      102
```

Оператор sizeof

Оператор sizeof подсчитывает размер любой переменной или любого типа и может быть полезен, если в программах требуется свести к минимуму машинно-зависимый код. Этот оператор особенно полезен там, где приходится иметь дело со структурами или объединениями.

Тип Размер в байтах

char 1

int 4

double 8

Поэтому при выполнении следующего кода на экран будут выведены числа 1, 4 и 8:

```
char ch;
int i;
double f;
printf("%d", sizeof(ch));
printf("%d", sizeof(i));
printf("%d", sizeof(f));
```

Пример

Размер структуры равен сумме размеров ее членов или *больше* этой суммы

```
struct s
 {
char ch;
int i;
double f;
} s_var;
```

Здесь sizeof(s_var) равняется как минимум 13 (=8+4+1)

Средство 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

Общий вид декларации typedef (оператора typedef) такой:

typedef *тип новое_имя*;

где *тип* — это любой тип данных языка С, а *новое_имя* — новое имя этого типа.

Новое имя является дополнением к уже существующему, а не его заменой.

Например, для float можно создать новое имя с помощью

typedef float balance;

Это выражение дает компилятору указание считать balance еще одним именем float.

Затем, используя balance, можно создать переменную типа float:

balance over_due;