Типы данных в языке С,

которые может создать пользователь

Существует пять способов создания своих типов данных:

Структура - это совокупность переменных, объединенных одним именем – составной (или смешанный) тип данных.

Битовое поле – разновидность структуры, предоставляющая легкий доступ к отдельным битам.

Объединение позволяет одному участку памяти содержать два или более различных типов данных.

Перечисление – список символов.

typedef – ключевое слово, которое создает новое имя существующему типу.

Структура — объединение нескольких объектов, в том числе и различного типа, под одним именем. В качестве объектов могут выступать переменные, массивы, указатели и другие структуры.
Компоненты, образующие структуру, называются членами, элементами или полями структуры.

Структура трактует группу связанных между собой объектов не как множество отдельных элементов, а как единое целое. Фактически, структура – сложный тип данных, составленный из простых типов.

Объявление структуры приводит к образованию шаблона, который может использоваться для создания объектов структуры.

Структуру можно формировать произвольным образом, используя любые типы данных. Однако, целесообразно структуру наполнять данными, имеющими между собой

Общая форма объявления структуры:

```
struct ярлык {
    тип ИмяЭлемента1;
    тип ИмяЭлемента2;
    . . .
    тип ИмяЭлементаn;
};
```

Определён тип, переменная не определена

Ярлык – имя типа структуры, а не имя переменной.

Структурные переменные – разделенный запятыми список имен переменных. Ярлык, или структурные переменные могут отсутствовать, но не одновременно.

```
struct ярлык {
    тип ИмяЭлемента1;
    тип ИмяЭлемента2;
    . . .
    тип ИмяЭлементаn;
}
структурные_переменные;
```

Определён тип, переменная определена

```
struct {
    тип ИмяЭлемента1;
    тип ИмяЭлемента2;
    . . .
    тип ИмяЭлементаn;
}
структурные_переменные;
```

Переменная типа структура определена

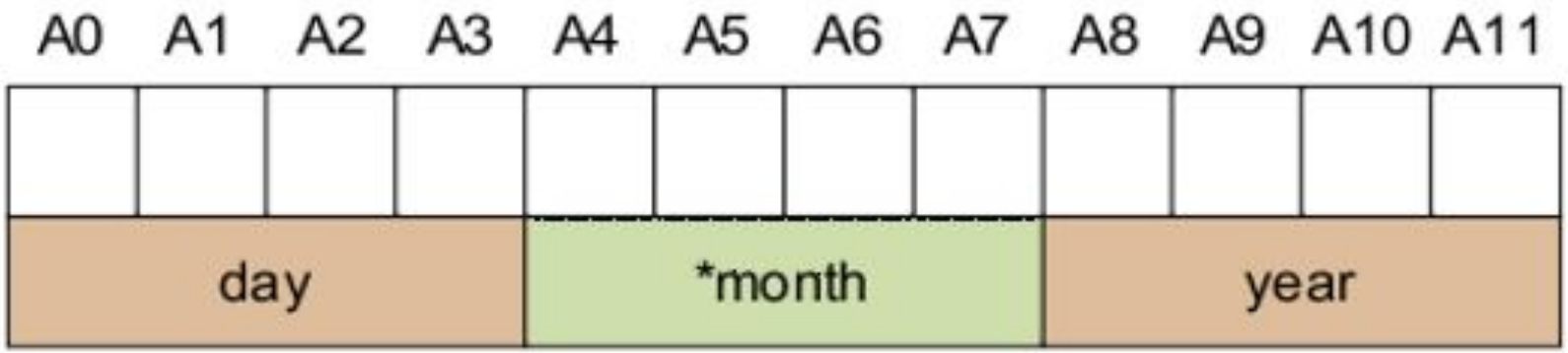
Объявление структуры - это оператор. Поэтому её объявление завершается точкой с запятой.

тип_структуры имя_переменной_1,

Пример объявления структуры

```
struct date {
    int day;     // 4 байта
    char *month; // 4 байта
    int year;    // 4 байта
};
```

Поля структуры располагаются в памяти в том порядке, в котором они объявлены:



Структура date занимает в памяти 12 байт.
Указатель *month при инициализации будет началом текстовой строки с названием месяца, размещенной в памяти

```
struct addr {
    int house,
    int apartment;
    char street [40];
    char city[20];
    char state[3];
    unsigned long int zip;
};
```

```
struct pass {
    char series[10];
    int number[10];
    char issued by[100];
     struct date when;
};
```

```
struct Personal_Information {
        char lastname[40];
        char firstname[40];
        struct pass passport:
        struct addr address;
    }
```

```
struct Personal_Information Ivanov, Petrov, Sidorov;
```

Доступ к отдельным членам структуры осуществляется с помощью оператора «.» (называется «точкой»).
Стандартный вид доступа следующий:
имя_структуры.имя_члена
Вывод поля структуры на экран (например):
printf("*const char *format*", имя_структуры.имя_члена);
имя_структуры.имя_члена = 1000;

Инициализация полей структуры может осуществляться двумя способами:

- **присвоение значений элементам структуры в процессе объявления переменной, относящейся к типу структуры;**
- **присвоение начальных значений элементам структуры с использованием функций ввода-вывода (например, printf() и scanf()).**

struct ИмяСтруктуры
ИмяПеременной={ЗначениеЭлемента1, ЗначениеЭлемента_2 ..., ЗначениеЭлементаN};

struct date bd = {8, "июня", 1978};

Во втором случае реализуется процедура занесения данных в соответствующие поля структуры.

```
#include <stdio.h>
#include <stdlib.h>
struct date {
  int day;
  char month[20];
  int year;
};
int main() {
        struct date when;
        printf("Введите дату рождения\nЧисло: ");
        scanf("%d", &when.day);
        printf("Месяц: ");
        scanf("%s", &when.month);
         printf("Год: ");
        scanf("%d", &when.year);
return 0;
}
```

Таким же образом массив символов address. house может использоваться в gets():

```
gets (Ivanov .address. house);
```

Данная команда передает указатель на символ, указывающий на начало name.

Для доступа к отдельным элементам addr_info.name можно использовать индекс name. Например, можно вывести содержимое addr_info.name посимвольно с помощью следующего кода:

```
register int t;
for(t=0; Ivanov .address. house [t]; ++t)
     putchar (Ivanov .address. house [t]);
```

Присваивание структур

Информация, содержащаяся в одной структуре, может быть присвоена другой структуре того же типа с помощью одиночного оператора присваивания, то есть не нужно присваивать значение каждого члена по отдельности:

```
#include <stdio.h>
int main(void) {
    struct {
        int a;
        int b;
    } x, y;
    x.a = 10;
    x.b = 20;
    y = x;          /* присвоение одной структуры другой */
    printf ("Contents of y: %d %d.", y.a, y.b);
    return 0;
}
```

Массивы структур

Объявление массива структур заключается в следующем:

- **определяется структура;**
- **объявляется массив переменных типа созданной структуры.**

Например:

```
struct addr address[100];
```

В результате получаем набор из 100 переменных, устроенных, как объявлено в типе структуры addr.
Для доступа к отдельным структурам массива address следует проиндексировать имя массива.
Например:

```
printf("%ld", address[28]. house);
```

Как и массивы переменных, массивы структур индексируются с нуля.

Передача членов структур в функции
Фактически в функции передается значения членов структуры. Следовательно, передается обычная переменная. Например:

```
struct dt {
        char x;
        int y;
        float z;
        char     s[10];
} data;
```

```
func(data.x);         /* передача символьного значения x */
func2(data.y);        /* передача целочисленного значения y */
func3(data.z); /* передача вещественного значения z */
func4(data.s); /* передача адреса строки s */
func(data.s[2]);     /* передача символьного значения s [2] */
```

Передача адреса отдельного члена структуры осуществляется с помощью оператора & перед именем структуры. Например:

```
func(& data.x) ;    /* передача адреса символа x */
func2(& data.y);   /* передача адреса целого y */
func3(& data.z);   /* передача адреса вещественного z */
func4(& data.S) ;  /* передача адреса строки s */
func(& data. S[2]);     /* передача адреса символа s[2] */
```

Следует иметь ввиду, что оператор & стоит перед именем структуры, а не перед именем члена.

Кроме того, массив S сам по себе является адресом, поэтому не требуется оператора &. Тем не менее, когда осуществляется доступ к отдельному символу строки S, оператор & необходим.

Передача всей структуры в функцию

Структура как аргумент функции передаётся с помощью стандартной передачи по значению, т.е. изменения, внесенные внутри функции, не повлияют на исходную структуру.

При использовании структуры как параметра следует помнить, что тип аргумента должен соответствовать типу параметра. Поэтому удобно определить структуру глобально, а ее ярлык использовать для объявления необходимых структурных переменных.

```
#include <stdio.h>
struct struct_type {
        int a, b;
        char ch;
};
void f1(struct struct_type parm);

int main(void) {
        struct struct_type arg;    /* объявление arg */
        arg.a = 1000;
        f1(arg);
        return 0;
}

void f1(struct struct_type parm) {
        printf("%d", parm.a);
}
```

Указатели на структуры

Указатели на структуры создаются так же, как и на другие типы переменных.

Указатели на структуру объявляются путем помещения * перед именем структурной переменной.

Например, ранее была определена структура addr, следующая строка объявляет address как указатель на данные этого типа:

```
struct addr *addr_pointer;
```

Для получения адреса структурной переменной используется & перед именем структуры.

```
struct bal {
float balance;
char name[80];
} person;
```

```
struct bal *p; /* объявление указателя на структуру */
```

тогда

```
p = &person;
```

помещает адрес структуры person в указатель p.

Для доступа к членам структуры с помощью указателя на структуру следует использовать оператор «->». Например:

```
p->balance;
```

Для доступа к членам структуры при работе с самой структурой используется оператор «точка». При обращении к структуре с помощью указателя используется оператор «->».

Битовые поля – возможность, позволяющую работать с отдельными битами.

Некоторые причины полезности битовых полей:

- ограниченное место для хранения информации и возможность сохранить несколько логических (истина/ложь) переменных в одном байте;
- передача битовой информации, закодированной в один байт;
- процедурам кодирования необходимо получать доступ к отдельным битам в байте.

Использование битовых полей для доступа к битам основано на структурах. Битовое поле – особый тип структуры, определяющей, какую длину имеет каждый член.

Используя битовые поля, можно упаковать целочисленные компоненты очень плотно, обеспечив максимальную экономию памяти.

Набор разрядов целого числа можно разбить на битовые поля, каждое из которых выделяется для определенной переменной. При работе с битовыми полями количество битов, выделяемое для хранения каждого поля отделяется от имени двоеточием

Стандартный вид объявления битовых полей :

```
struct имя структуры {
    тип имя1: длина;
    тип имя2: длина;
    ...
    тип имяN: длина;
}
```

При работе с битовыми полями нужно внимательно следить за тем, чтобы значение переменной не потребовало памяти больше, чем под неё выделено.

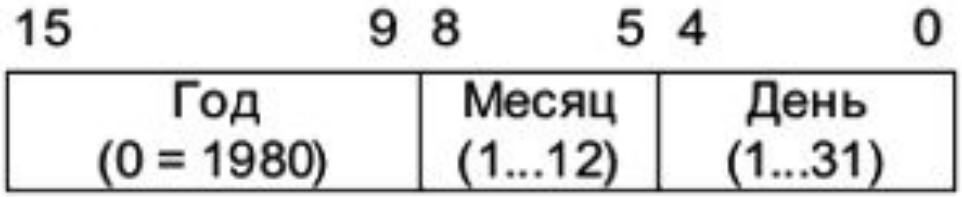
Битовые поля должны объявляться как int, unsigned или signed. Битовые поля длиной 1 должны объявляться как unsigned, поскольку 1 бит не может иметь знака. Битовые поля могут иметь длину от 1 до16 бит для 16-битных сред и от 1 до 32 бит для 32-битных сред.

Пример структуры, содержащей три переменные по одному биту кажда

```
struct device {
unsigned active : 1;
unsigned ready : 1;
unsigned xmt_error : 1
} dev_code;
```

К каждому полю структуры обращение осуществляется с помощью оператора «.». Если обращение к структуре происходит с помощью указателя, то следует использовать оператор «->».

Пример программы упаковки даты в битовые поля




```
#include <stdio.h>
#include <stdlib.h>
#define YEAR0 1980
struct date {
      unsigned short day : 5;
      unsigned short month : 4;
      unsigned short year : 7;
};
int main() {
    struct date today;
    today.day = 16;
    today.month = 12;
    today.year = 2019 - YEAR0; //today.year = 39
    printf("\n Сегодня %u.%u.%u \n", today.day, today.month, today.year + YEAR0);
    printf("\n Размер структуры today : %d байт", sizeof(today));
    printf("\n Значение элемента today = %hu = %hx шестн.", today, today);
    return 0;
}
```

Нет необходимости называть каждое битовое поле. К полю, имеющему название, легче получить доступ:

```
struct device {
    unsigned active : 1;
    unsigned ready : 1;
    unsigned xmt_error : 1;
    unsigned : 2;
    unsigned EOT : 1;
} dev_code;
```

В битовых полях можно смешивать различные структурные переменные. Например:

```
struct emp {
    struct addr address;
    float pay;
    unsigned lay_off:1;
    unsigned hourly:1;
    unsigned deductions:3;
};
```

Битовые поля имеют некоторые ограничения. Нельзя получить адрес переменной битового поля. Переменные битового поля не могут помещаться в массив. Переходя с компьютера на компьютер нельзя быть уверенным в порядке изменения битов (слева направо или справа налево). Любой код, использующий битовые поля, зависит от компьютера.

Объединение – сложный тип данных, позволяющий размещать в одном и том же месте оперативной памяти данные различных типов.

Размер оперативной памяти объединения определяется размером памяти данных того типа, который требует максимального количества байт.

Элемент меньшей длины объединения использует только часть отведенной памяти. Все элементы объединения хранятся в одной и той же области памяти, начиная с одного адреса.

```
union ИмяОбъединения
{
  тип ИмяОбъекта1;
  тип ИмяОбъекта2;
  . . .
  тип ИмяОбъектаN;
};
```

Как и для структур, можно объявить переменную, поместив ее имя в конце определения или используя отдельный оператор объявления.

union ИмяОбъединения
структурные_переменные;

Когда объявлено объединение, компилятор автоматически создает переменную достаточного размера для хранения наибольшей переменной, присутствующей в объединении.

Для доступа к членам объединения используется синтаксис, применяемый для доступа к структурам - с помощью операторов «.» и «->».
Чтобы работать с объединением напрямую, надо использовать оператор «.».
Если к переменной объединения обращение происходит с помощью указателя, надо использовать оператор «->».

Объединения применяются для следующих целей:

- **для инициализации объекта, если в каждый момент времени только один из многих объектов является активным;**
- **для интерпретации представления одного типа данных в виде другого типа.**

Пример использования объединения для представления вещественное число типа float в виде совокупности байтов:

```
#include <stdio.h>
#include <stdlib.h>
union types {
    float f;
    unsigned char b[4];
};
int main() {
    types value;
    printf("N = ");
    scanf("%f", &value.f);
    printf("%f = %x %x %x %x", value.f, value.b[0], value.b[1], value.b[2], value.b[3]);
return 0;
}
```

Использование объединений помогает создавать машинно-независимый (переносимый) код. Поскольку компилятор отслеживает настоящие размеры переменных, образующих объединение, уменьшается зависимость от компьютера. Не нужно беспокоиться о размере целых или вещественных чисел, символов или чего-либо еще.

Объединения часто используются при необходимости преобразования типов, поскольку можно обращаться к данным, хранящимся в объединении, совершенно различными способами.

Перечисления - это набор именованных целочисленных констант, определяющий все допустимые значения, которые может принимать переменная.

Перечисления определяются с помощью ключевого слова enum, которое указывает на начало перечисляемого типа:

enum ярлык { список перечислений} список переменных;

Как имя перечисления – ярлык, так и список переменных необязательны, но один из них должен присутствовать.

Список перечислений – это разделенный запятыми список идентификаторов.

Как и в структурах, ярлык используется для объявления переменных данного типа.

Перечисления используются для объявления символических имен, которые являются целочисленными константами и улучшают читабельность кода. То есть перечисление является целочисленным типом, и их можно использоваться вместо целочисленных типов.

```
{
  enum eDirection {RIGHT, LEFT, DOWN, UP};   // создаем перечисление с
                                              // именем дескриптора eDirection
  enum eDirection dir; // создаем переменную dir
                       // с перечислимым типом eDirection
  dir = UP; // присваиваем переменной dir константу UP
 }

 {
  enum {RIGHT, LEFT, DOWN, UP}; // создаем перечисление без дескриптора,
                                // просто набор констант
  int dir;     // создаем переменную dir с перечислимым типом int
  dir = UP; // присваиваем переменной dir константу UP
 }

 {
     typedef enum {RIGHT, LEFT, DOWN, UP} eDirection; // создаем тип
                                  // перечисления с именем eDirection
     eDirection dir;    // создаем переменную dir с перечислимым типом eDirection
     dir = UP;  // присваиваем переменной dir константу UP
```

Конечный тип перечисления зависит от реализации компилятора, также все может зависеть от того какие значения имеют константы. Если значение первой константы не указано, оно по умолчанию равно 0, все следующие за ней, на 1 больше предыдущей.

```
{  enum eDirection
  {  RIGHT,  // по умолчанию = 0
    LEFT,   // = 1
    DOWN,   // = 2
    UP      // = 3
  };
}
```

```
{ enum eDirection
  {  RIGHT,    // по умолчанию =
0
    LEFT = 4, // = 4
    DOWN,     // = 5
    UP        // = 6
  };
}
```

```
{ enum eDirection
  { RIGHT = 8,  // = 8
    LEFT,       // = 9
    DOWN = 100, // = 100
    UP          // = 101
  };
}
```

Компилятор может принять решение, выделить под хранение этих констант тип unsigned char (0...255) или char (-128...127), так как все эти константы лежат в границах для этих типов, а может выделить и тип int (все зависит от реализации того или иного компилятора).

Так как перечисления используются в основном для повышение читаемости кода, то мы видим что вместо "магических значений" 1, 2, 3, ... мы используем буквенные обозначения, которые нам говорят о направлении, и в коде, когда мы видим константу DOWN мы понимает что она означает, но когда видим к примеру число 3, то можно только гадать что это (направление, цвет, коэффициент).

Если используются "магические числа", нужно везде проставлять комментарии, что это за числа. Легко сделать ошибку. Легко забыть обработать какое-то состояние.

Переменные типа enum могут использоваться в индексных выражениях и как операнды в арифметических операциях и в операциях отношения.

Использование элементов перечисления должно подчиняться следующим правилам:

1. Переменная может содержать повторяющиеся значения.

2. Идентификаторы в списке перечисления должны быть отличны от всех других идентификаторов в той же области видимости, включая имена обычных переменных и идентификаторы из других списков перечислений.

3. Имена типов перечислений должны быть отличны от других имен типов перечислений, структур и смесей в этой же области видимости.

4. Значение может следовать за последним элементом списка перечисления.

Ключевое слово typedef позволяет определять имена новых типов данных. На самом деле новый тип данных не создается, а определяется новое имя существующему типу. Оператор typedef позволяет облегчить создание машинно-независимых программ. Единственное, что потребуется при переходе на другую платформу – это изменить оператор typedef. Он также может помочь документировать код, позволяя назначать содержательные имена стандартным типам данных. Стандартный вид оператора typedef :

typedef тип имя;

тип —любой существующий тип данных, а имя – новое имя для данного типа. Новое имя определяется в дополнение к существующему имени типа, а не замещает его.

Пример. Можно создать новое имя для float:

```
typedef float balance;
```

Данный оператор сообщает компилятору о необходимости распознавать balance как другое имя для float. Далее можно создать вещественную переменную, используя balance:

```
balance past_due;
```

Здесь past_due – вещественная переменная типа balance (float). Можно использовать typedef для создания имен для более сложных типов. Например:

```
typedef struct {
      float due;
      int over_due;
      char name[40];
} client; /* здесь client - это имя нового типа */
client clist[NUM_CLIENTS]; /* массив структур типа client */
```

Использование typedef может помочь при создании более