ЛЕКЦИЯ 14

Типы данных , определенные пользователем

Программист может строить на основе стандартных типов данных, массивов и указателей свои собственные типы для того, чтобы адекватно учесть специфику решаемой задачи. Для этой цели он может воспользоваться одним из следующих способов:

Задание перечислимого типа

с помощью оператора

```
enum [имя_типа] {список_констант}
  [список_переменных];
```

Константы перечислимого типа считаются числовыми и целочисленными (значение первой из констант полагается равной нулю, а всех остальных – увеличивается на единицу).

Однако можно указать другое значение, если его записать после знака равенства, последующие значения будут увеличиваться начиная с этого индекса

```
enum color {red, green, blue};
cout<<red<<' '<<blue<<' '<<green; // 0 2 1
enum color2 {yellow, cyan=100, magenta};
cout<<yellow<<' '<<cyan<<' '<<magenta;
// 0 100 101
```

Перечислимые константы могут инициализироваться произвольными целыми константами:

```
enum color{r=2,g=4,b=6};
```

Допускается определение анонимных перечислений (без указания имени типа).

```
enum {off,on}; //off=0,on=1
enum {MON,TUES,WED,THURS,FRI,SAT,SUN};
```

эквивалентно описанию 7 констант-кодов дней недели:

```
const int MON = 0;
const int TUES = 1;
const int WED = 2;
const int THURS = 3;
const int FRI = 4;
const int SAT = 5;
const int SUN = 6;
```

```
enum color c;  //объявление переменной типа color
color d;       // объявление переменной типа color
enum turn {off,on} a;    //объявление типа и переменной
enum {off,on} a;  //объявление  переменной без указания имени типа
```

В языке С тип **enum** эквивалентен типу **int**.

Это позволяет присваивать целочислен-ные значения непосредственно переменным типа перечисления.

В С++ ведется более строгий контроль за типами данных, и такие присваивания не допускаются.

```
color d=0; // ошибка
```

Переменным перечисляемого типа можно присваивать только именные значения перечисляемых констант.

```
color c=r; //правильно
```

Целочисленным переменным можно присваивать значения перечисляемых констант.

```
int c=r;
enum My_Error{Err_read, Err_ write,
Err_convert};
My_Error error;
…
switch (error)
{  case Err_read: {операторы} break;
   case Err_ write: {операторы} break;
   case Err_convert: {операторы} break;
}

enum chess{pawn,knight,bishop,rook,queen,king};
   //шахматные фигуры
enum coins
{penny,nickel,dime,quarter, half_dollar,dollar};
   //монеты
enum direction
 {north,south,east,west};
```

Недостаток перечислений

Они не распознаются средствами ввода/вывода С++.

```
direction dir1 = east;
cout << dir1 << endl;
```

Будет выведено не значение **east**, а ее внутреннеее представление , т.е. число

Структуры

В отличие от массива структура может содержать элементы разных типов. Данные , входящие в структуру, называются элементами, членами или полями структуры. (В Паскале структурам соответствуют записи.)

Объявление структуры

Задает ее внутреннюю организацию, описывая поляя, входящие в состав структуры.

```
struct [имя_типа_структуры]
{тип поле;[{тип поле;}…]};
```

Само объявление структуры не создает никаких переменных, т.е. не происходит ни выделения памяти, ни объявления переменных.

Например:

```
struct emp
     {
     int empno;
     char value[10];
     double salary;
     };
```

Элементами структуры могут быть массивы или другие структуры (поле ***value*** -массив).

Определение структурной переменной

Формат:

```
имя_типа_структуры список_переменных;
```

Например:

```
emp a; //в C++
emp a1,a2,a3; //в C++
struct emp a; //в C
struct emp a1,a2,a3; //в C
```

Будет выделена память для объявленных переменных.

Опишем книгу:

```
int MAXTIT=41, MAXAUT=31;
struct book {
     char title[MAXTIT];
     char author[MAXAUT];
     float value;  //цена книги
     };
book libry; // переменная типа book
```

Имя типа структуры нужно, если структура будет использоваться неоднократно.Тогда можно объявлять переменные такого типа неоднократно:

```
book my_book; // переменная типа book
```

Но если структура используется один раз для объявле-ния переменных такого типа, то можно объединить объявление типа и определение переменных:

```
struct {
     char title[MAXTIT];
     char author[MAXAUT];
     float value;  //цена книги
     }  libry;
```

Однако если ***имя_типа_структуры*** не указывается, то структурные переменные данного типа могут быть объявлены только единожды.

```
struct {int x; int y; float length;} vector1;
```

После этого программист больше не может объявить переменные данного типа, так как не существует идентификатора типа. Следовательно, переменные безымянных структур не могут передаваться функциям.

```
struct tvector
{ int x; int y; float length; }  vector2;
tvector  vector3;
```

Описание структуры можно размещать внутри функции, тогда оно будет доступно только в этой функции, либо вне всех функций, тогда оно будет глобальным. Если описание структур используется в нескольких файлах, то это описание обычно помещают в заголовочный файл и включают в текстовый файл при помощи

```
#include "имя_файла"
```

Доступ к полям структуры

Для доступа к элементу структуры используется ***операции точка*** (**.**) . Операция точка называется ***операцией доступа к полю структуры***.

Формат доступа к полю:

```
имя_структурной_переменной.имя_поля
```

Введем данные для переменной **libry**:

```
printf("название книги\n");
gets(libry.title);
printf("автор книги\n");
gets(libry.author);
printf("цена книги\n");
scanf("%f",&libry.value);
```

Инициализация структур

Структурные переменные могут инициализироваться при их определении. Для этого после имени структурной переменной располагают оператор присваивания и в фигурных скобках перечисляют значения полей структуры.

```
emp x={10,"Paul",200.15};
book libry1 = {"Обрыв","Н.А.Гончаров",40000.};
```

Присваивание структурных переменных

```
emp y;
book libry2;
y = x;
libry2 = libry1;
```

Выполнять присваивание можно только переменным одного и того же типа.
Можно присваивать значения полям:

```
libry2.value=70000.0;
strcpy(libry2.author,"Бьерн Страуструп");
```

Обращение к элементам массива структур

Структурные переменные могут быть элементами массива.

```
book librys[100];
   // массив из 100 элементов
librys[0].value;-цена 1-ой книги
librys[4].title;-название 5-ой книги
librys.value[0] //ошибка
Но
librys[4].title[0]-верно!!! ибо это 1-ый символ названия
   5-ой книги
```

Инициализация массива структур

```
book lib[]={
{"Обрыв","Н.А.Гончаров",40000.0},
{"Овод","Э.Л.Войнич",35000.0},
{"Идиот","Ф.М.Достоевский",45000.0}};
// массив из 3 элементов
```

Пример. Обработаем массив книг.

```
#include <stdio.h>
#include <string.h>
const int MAXTIT=41;
const int MAXAUT=31;
const int MAXBOOKS=100;
```

```
const char STOP[] ="\0";
struct book {
   char title[MAXTIT];
   char author[MAXAUT];
   float value;
   };

int main()
{
book libry[MAXBOOKS];
// массив структур  типа book
int count=0;
int i;
printf("название книги или enter для завершения\n");
while
 (strcmp(gets(libry[count].title), STOP)!=0 &&
                                 count< MAXBOOKS)
 {printf("автор книги\n");
  gets(libry[count].author);
  printf("цена книги\n");
  scanf("%f",&libry[count++].value);
  while(getchar()!='\n'); //очистка
  if (count< MAXBOOKS)
   printf ("название книги или enter для завершения\n");
 }
printf("Список книг\n");
for(i=0;i<count;i++)
 printf("%s, %s:%f\n", libry[i].title,
        libry[i].author, libry[i].value);
return 0;
}
```

В программе условие

```
(strcmp(gets(libry[count].title),STOP)!=0 &&
 count< MAXBOOKS)
```

определяет конец работы цикла, константа **STOP** описывает пустую строку, т.е. строку, в которую ничего не ввели, только нажали **enter.**

Второе условие проверяет не заполнили ли весь массив.

В программе есть странная строка

```
while(getchar()!='\n');//очистка
```

Она использует особенности функции **scanf,** которая игнорирует символы

пробела и новой строки. Если ее не записать, то после ввода цены в буфере ввода останется символ новой строки и при следующем чтении названия книги первым символом будет символ новой строки и программа закончит работу, несмотря на то, что будет попытка набрать название новой книги.

Вложенные структуры

Иногда удобно вкладывать структуры друг в друга:

```
const int dl=20, dl1=50;
struct fio{
     char fam[dl];
     char imya[dl];
     char otch[dl];
     }

struct anketa{
     fio person;//вложенная структура
     char prof[dl];
     char adres[dl1];
     }
```

Инициализация переменной такого типа:

```
anketa individ = { {"Иванов","Сергей","Петрович"},
"врач","г.Минск,ул.Гая,д.2,кв.4"};
```

Доступ к полям такой структуры:

```
individ.person.fam //"Иванов"
individ.person.imya //"Сергей"
```

Указатели на структуры

Удобства применения указателей:

- Указателями часто легче пользоваться, чем самими структурами (напр, в задачах сортировки);
- Использование указателя на структуру в качестве параметра функции позволяет изменять значения элементов структуры.
- Многие представления данных используют указатели на структуру в качестве своих полей (списки, например).

Для доступа к полям структурной переменной через указатель используется оператор «->» (минус больше):

```
указатель_на_структуру -> имя _поля
```

```
struct persona
{ char name[20];
  int code;
  char region[20];
  char street [20];
}
persona rd;
persona *prd = &rd;
rd.code=12006;
strcpy(rd.name, "Иванов");
strcpy(prd->region, "Беларусь");
printf("Reader: %s, Code: %d, Region: %s\n", rd.name,
       rd.code, rd.region);
printf("Reader: %s, Code: %d, Region: %s\n",
       prd->name, prd->code, prd->region);
```

Передача структур в функции

Когда структура используется как параметр функции она передается по значению.

```
void print(struct person reader)
{
 printf(" %s, %d, %s\n",
 reader.name, reader.code,
 reader.region);
}
```

Если необходимо, чтобы функция изменяла значения полей структуры, то как параметр в функцию передается указатель на структуру.

```
void int_emp(person*r, int m, char *s, char *rg)
{ r -> code = m;
  strcpy(r -> name, s);
  strcpy(r -> region, rg);
}
//или
{
 (*r).code = m;
 strcpy((*r).name,s);
 strcpy((*r).region,rg);
}
```

круглые скобки Н., ибо операция "." имеет приритет выше, чем "*".

Объединения (union)

Объединением называется область памяти, использующаяся для хранения данных разных типов.

Причем одновременно в объединении могут храниться данные только одного определенного типа.

Данные входящие в объединение называются его полями (элементами). Тип описывающий объединение также называется объединением.

С помощью объединения к одной и той же области памяти можно обращаться как к данным различного типа.

Объединение можно рассматривать как структуру, все элементы которой имеют одинаковый начальный адрес в памяти.

Длина объединения определяется наибольшей из длин полей объединения. Объединения позволяют сэкономить память в том случае, если необходимо хранить только один из элементов объединения, но заранее неизвестно, какой именно.

Синтаксис объединений аналогичен синтаксису структур.

```
union [имя_типа_объединения]
 {
     тип_1 элемент_1;
     тип_2 элемент_2;
     …
     тип_n элемент_n;
 }
```

```
union num
{
     int n;
     double f;
}
union num d;
//d-переменная типа num (в C)
num d; // (в C++)
```

В переменной d могут храниться целые и плавающие числа, но не оба одновременно. Длина переменной d = длине max элемента (в данном случае double).

При объявлении типа объединения его можно инициализировать значением, которое имеет тип 1-го элемента объединения.

```
union num e={10}; //правильно, e=10
```

Имена элементов переменной типа объединения принадлежат локальной области видимости внутри блока. Для доступа к элементам объединения используется оператор (.).

```
num e, g;
e.n=1;
e.f=2.7;
```

Над объединениями одного типа возможна операция присваивания.

```
g=e;
```

Для доступа к элементам объединения на которые указывает указатель используется оператор(->).

```
void print(char c, union *un)
{switch(c)
  {case 'd':
     {printf("n=%d\n",un->n);
      break;}
   case 'f':
     {printf("n=%f\n",un->f);
      break;}
   default:
     printf("union\n");
   }
}
```

Объединение (как и структура) не может содержать в качестве элементов переменные собственного типа.

Но в качестве элементов объединения могут использоваться указатели или ссылки на переменные собственного типа объединения.

В C++ при использовании объединений следует учитывать некоторые особенности, так объединение является классом и может содержать в себе также функции в качестве элементов.

Битовые поля

Поля битов – это особый тип полей структуры. Они используются, если нужно запомнить данные, для которых необходима ячейка памяти менее одного байта. Это могут быть, например, флажки, которым достаточно одного бита для представления значений в двоичной логике.

Битовыми полями называются элементы структуры или объединения, которые определяются как последовательность битов.

Синтаксис описания битового поля следующий:

```
Идентификатор_типа [имя_поля]: ширина_поля;
```

В качестве **идентификатора_типа** в соответствии со стандартом ANSI разрешено использовать ключевые слова **int, signed** или **unsigned**.

Ширина_поля должна быть выражена целочисленным значением, которое определяет, сколько битов необходимо выделить указанному полю.

Пример

```
struct status {
     unsigned Shadow : 1;
     unsigned Border : 1;
     unsigned Palette : 3;
     unsigned Hline : 1;
     unsigned F1Activ : 1;
}
```

К битовым полям не может быть применена операция получения адреса (**&**) и поэтому не существует указателей на поля.

Порядок размещения битовых полей памяти в значительной степени зависит от компилятора и аппаратного обеспечения. Не нужно пытаться экономить память с помощью битовых полей, так как это чаще всего не удается.

При использовании битовых полей обычно предполагается, что память для них будет выделена в виде последовательности битовых ячеек, следовательно, указанная в примере структура, состоящая из битовых полей, заняла бы 7 бит, следующих один за другим; но так ли это на самом деле - зависит, в конце концов, от компилятора.

Кроме того, адресация отдельных битов в памяти является менее рациональной, чем адресация байтов или слов, так как компилятор должен генерировать специальные коды.

Непоименованные битовые поля с шириной поля 0 задают выравнивание следующего поля по границе слова.

Инициализируются битовые поля как обычные элементы структуры.

```
status a={0,1,1,2,1};
```

Если надо обрабатывать только некоторые биты из поля, то остальные биты можно не именовать.

```
struct s
{ unsigned b1:1;
  unsigned : 8;  // не именованное поле
  unsigned b2:1;
}
```

Объявление typedef

В языке Си можно определить любые типы данных на основе встроенных типов. При определении типа указываются его идентификатор и идентификатор существующего типа. Такое определение типа происходит через ключевые

слова **struct**, **union** или **typedef**.

```
typedef  идентификатор_существующего_типа
     новое_имя_типа;
```

Пример

```
typedef int length;
// синоним названия типа
```

Функция **typedef** позволяет создать свое собственное имя типа. Например, после инструкции

```
typedef  float real;
```

можно использовать **real** для объявления переменных:

```
real x, y[50], *p;
```

Область действия такого определения зависит от расположения оператора **typedef.**

```
typedef struct person
 {char name[20];
  int post_code;
 } PERSON;
```

Теперь можно объявлять переменные типа **PERSON** и работать с ними как с типом **person.**

```
typedef void (*func_typ) (int, int)
```

Тип **func_typ** - определяется как “указатель на функцию с двумя параметрами типа **int** и не возвращающую значения".

- Описания типа используют, чтобы создать легко запоминающиеся имена.
- Описания типа используют, чтобы создать сокращения для обширных типов (**struct**, **class**, указатель на функцию).
- Описания типов позволяют решить проблему при переносе программного обеспечения. Если с помощью **typedef** объявить типы данных, которые являются машинно-зависимыми, то при переносе программы на другую машину потребуется внести изменения только в определения **typedef**.

КОНЕЦ ЛЕКЦИИ